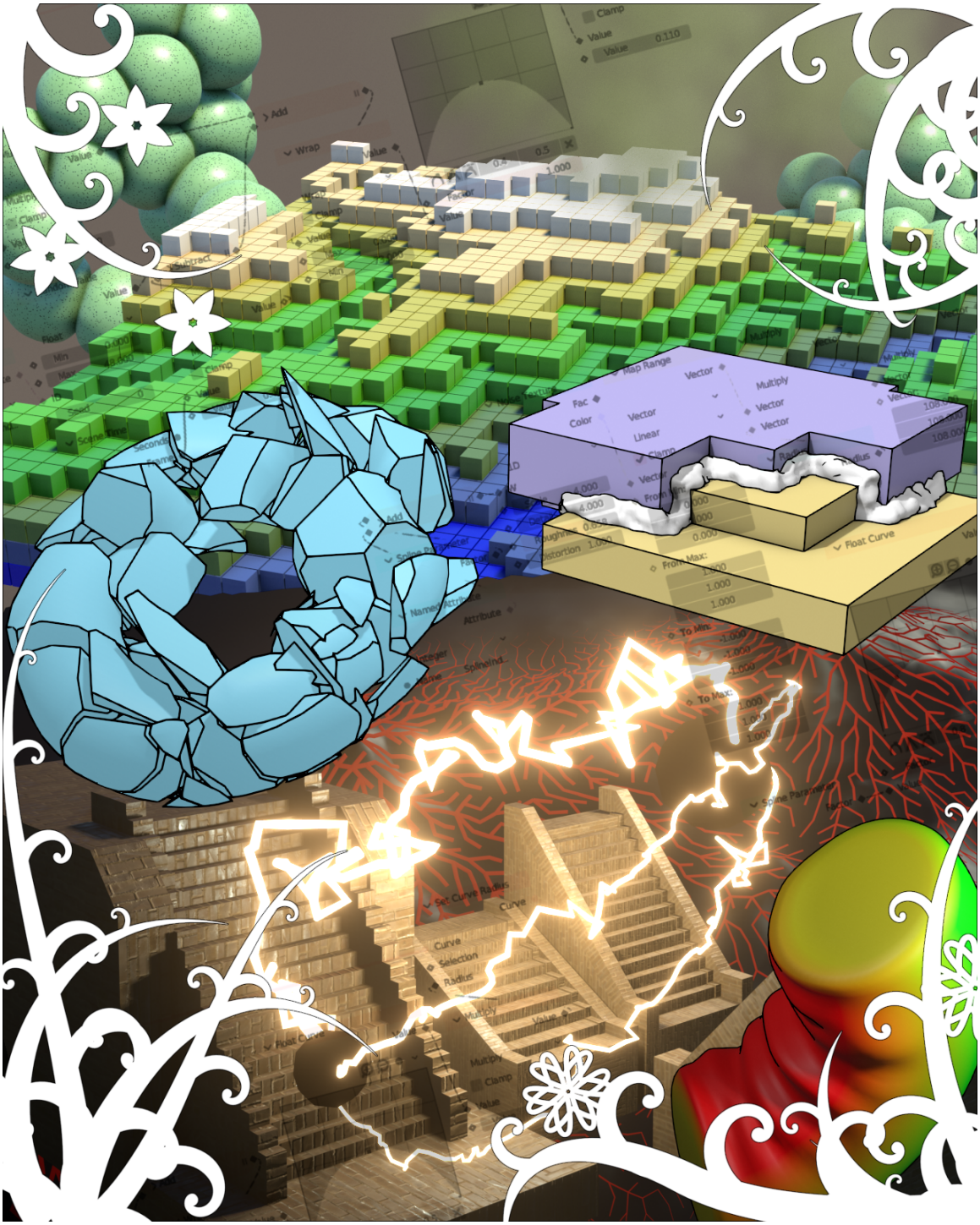
Blender ジオメトリノード 解説＆作例集 Vol.2

Geometry Nodes (for Blender 3.2 - 3.3)

Version 1.6

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. 2

Geometry Nodes (for Blender 3.2 and 3.3)

作者 Twitter アカウント
Q@スタジオぽぷり

2022.09.04 ver 1.0 2022.11.07 ver 1.2 Blender 3.4 での変化に関して追記と更新。一部サンプルの変更。 2022.12.23 ver 1.3 Blender 3.4 での変化に関して追加と更新。一部サンプルの追加。	2023.04.11 ver 1.4 Blender 3.5 での変化に関して追加と更新。一部サンプルの追加。 2023.07.06 ver 1.5 Blender 3.6 での変化に関して追加と更新。 2023.11.27 ver 1.6 Blender 4.0 での変化に関して追加と更新。一部サンプルの追加。

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。
前回の本、Vol. 1 は Blender 3.0 の時に最初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。

Vol. 1 では、できるだけノードを網羅したかったので、一種類のノードに付き、大体 1 つずつ作例を記載しました。
しかし、時間が経ってくると、「他にもこういう別方向の作例があるなぁ」とか、
「この作例は、ちょっと変化球過ぎたのでは…（もっと素直な使い方の例を載せた方が良かったのかも…）」などといった点に気が付き始めました。
また、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。

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 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。

Blender 3.4 になると、新たな機能が追加されるのと同時に、いくつかのノードが統廃合されるような変更が行われています。
そのため、以前のノードの組み方とは使うノードの組み方などが変化している箇所があるので、注意が必要な箇所などがあります。
また、ver 1.0 のサンプルが一部非効率な作りになっているところの修正などを加え、Version 1.2, Version 1.3 そして Blender 3.5と3.6と4.0 へ対応した更新を行い Version 1.4, 1.5, 1.6 としました。

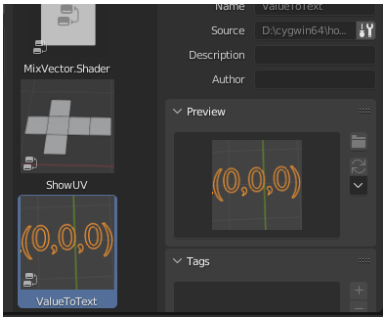
※本書に埋め込んである画像の一部は GIF アニメーションになっています。
.pdfとして書き出したファイルは、残念ながら静止画として最初のフレームが使われているだけなのですが、html版はブラウザーで見れば動いて見えるはずです。
また、**サンプルファイルは Blender 4.0 で保存をしました。**互換性に関する仕様として Blender 3.6 で開くことはできますが、それより前の Blender では開くことはできません。
Blender 3.6 で保存をすることで、Blender 3.5 以前の Blender で開くことが出来ます。(但し、必ずしも正常に動作するかは保証できないのですが…)

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 3.1 以降で起きた、ジオメトリノードに関係する変化の中で、単純に「ノードが増えて機能が増えた！やった！」ということの他に、
周辺も含めて気になるものを幾つか挙げてみます。

アセットブラウザでノードグループの登録が可能に

Blender 3.1 から、ノードグループがアセットブラウザに登録可能になり、再利用が容易になっています。

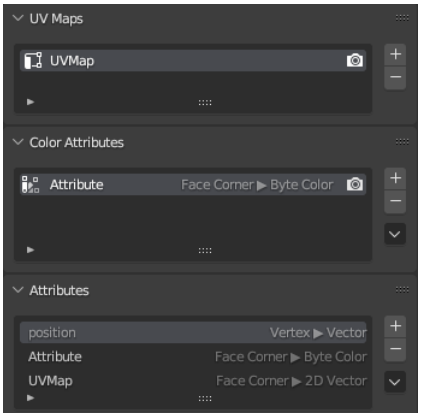


これは、自分が登録したジオメトリノード用のノードグループの一例です。
「UVの形をしたメッシュを作成して、UVの形を画面内で確認するノード」や、
「数値やベクトルの値を、テキストとして画面内で表示ができるようにしたノード」を、
ノードグループ化して登録し、アセットブラウザから再利用ができるようにしています。
その際、アイコン用に画面をキャプチャした小さな画像を用意して、カスタムアイコンとして使っています。

アセットブラウザに関しては、後で簡単に使い方を説明します。
Blender 3.4 からはアセットブラウザからだけではなく、メニューからも利用できるようになっています。

頂点カラー、UVマップ等のアトリビュートへの統合

ジオメトリノードでは、各頂点について位置や色のパラメーターを弄ります。
それらのパラメーターを Attribute (アトリビュート) と呼んでいます。



昔からある Vertex Color (頂点カラー) の色データや UVMap の情報は、
「Attribute とは別の似て非なるデータ」だったわけですが、
それぞれ Blender 3.2 と 3.5 で Attribute に統合されています。
旧頂点カラーは名目上廃止になっています。

頂点カラーを編集しようと Vertex Paint(頂点ペイント) の編集モードにして、追加されるデータを見てみると…
項目自体が Vertex Color だったのが Color Attributes になっているのが分かります。

同様にUVマップ情報もアトリビュートとしても存在していることが分かります。
旧頂点カラー(Color Attribute)は Face Corner(面コーナー)が持っている Byte Color(RGBが256段階ある色データ)、
UVマップ情報は、Face Corner が持っている 2Dベクトルであることが分かります。

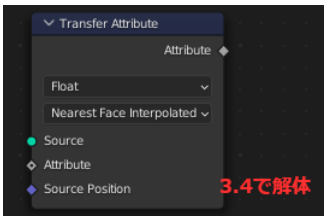
さらに、Blender 4.0 では、ベベルモディファイアで使うベベルのパラメーターが、bevel_weight_edge と bevel_weight_vertex としてアトリビュートに統合されています。

ノードがいくつか変更

ノードのいくつかは、バージョンアップの際に仕様が変更されたり、統廃合がされています。
Blender 3.3 から、Separate RGB(RGB分離) と Combine RGB(RGB合成) が Separate Color(カラー分離) と Combine RGB(カラー合成)に変更になっています。
また、Blender 3.4 から、Mix RGB ノードが Mix ノード(ミックス) に変更されています。
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。

3.2 以前に作った .blend ファイルを 3.3 に持って行く場合には自動で変換がされるのですが、
3.3 で保存した .blend ファイルは、3.2 ではノードが見つからないことになってしまいます。
(Mix ノードの 3.3 から 3.4での扱いも同様です)

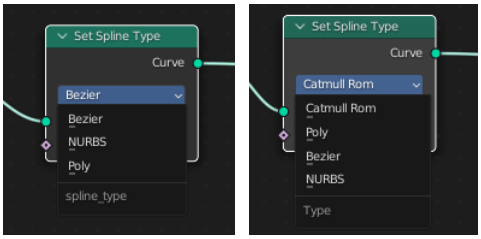
前のバージョンにさかのぼってファイルを読み込むのは、もとからあまり推奨されることではないのですが、
色関係のノードとして使う頻度が高いノードでの変更なので、要注意です。



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。
元から少し扱いの難しめのノードなので、利用機会はそんなに多くないノードなのですが、注意が必要です。
本書の最後に、簡単な解説を掲載しました。

新しいカーブシステムとカーブの種類の追加

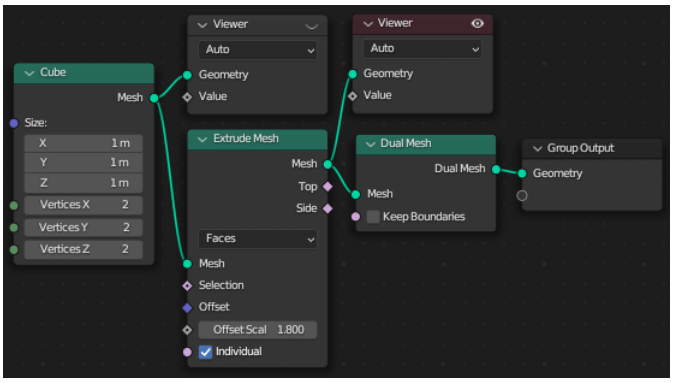
Blender 3.3 で、主に髪の毛のような多数のカーブを制御する目的で Curves というオブジェクトの種類が追加されています。
恐らくその関係でしょう、カーブのスプラインのタイプに Catmull Rom が追加されています。
Catmull Rom は、制御点の位置（前後入れて４点）でカーブの曲線を表現する方法で、Bezier カーブのようにハンドルを使わずに曲線を表現します。



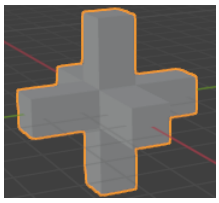
これに伴いスプラインの種類を表す内部の番号がつけなおしになっているようです。
（メニューの順番も変わっていますね）
Curves オブジェクトに関したジオメトリノード関係の解説は Vol.3 で扱っています。

ビューワーノードの強化

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。

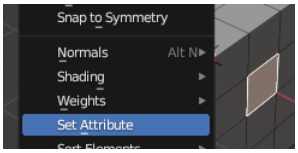
ジオメトリノードで編集中の状態が視認できるので、ややこしい処理をして「だんだん分からなくなってきた」ときに、とても役に立ちます。
これはあくまでプレビュー用なので、レンダリングしたり、別のジオメトリノードを追加したりすると、最後まで処理が進んだ Group Output での形が表示されます。

メニューと名前の更新

Blender 3.5 では、機能(ノードの種類)の増加に伴って、メニューが再編成されています。また、一部ノードの名前やソケット名が変更になっています。
Transform > Transform Geometry
Field at Index > Evaluate at Index
Interpolate Domain > Evaluate on Domain
Face Set Boundary > Face Group Boundary
また、Group Index という名のソケットは、Group ID という名前に変更されています。

アトリビュート編集

また、Blender 3.5での機能追加として、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。

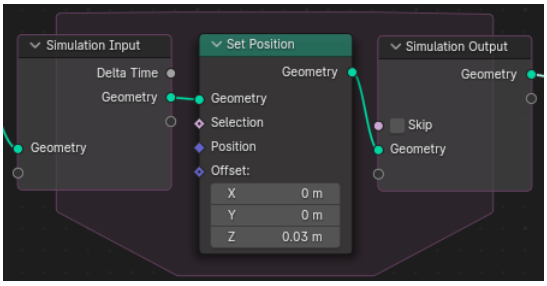


メッシュ編集時のメニューから、Mesh - Set Attribute(属性を設定)にあります。 現在アクティブなアトリビュートに対して、数値やそのほか色などの設定をすることができます。

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は、将来の目標になっているようです。

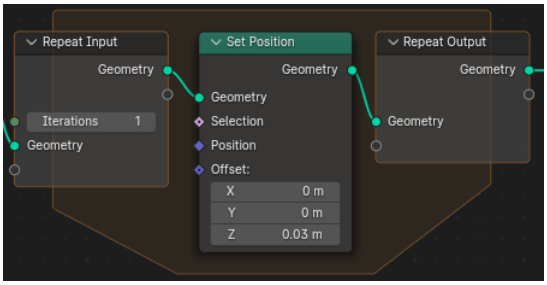
シミュレーションノード と Repeat Zone(リピートゾーン)

Blender 3.6 と 4.0 ではそれぞれ、Simulation nodes(シミュレーションノード) と Repeat Zone(リピートゾーン)機能が追加されました。
これにより、時間的に進化していくような効果や、繰り返しをともなう処理の作成機能が大幅に強化されています。
Simulation Input と Simulation Output 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。

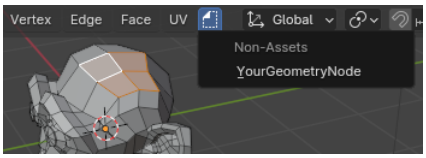
シミュレーション機能にはこのシリーズでは詳しくは踏み込みませんが、非常に強力な機能なので、ジオメトリノードに慣れたら是非シミュレーションにも手を出してみて欲しいと思います。



Repeat Zone も似たような仕組みで、Repeat Input と Repeat Output 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。

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、編集中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）



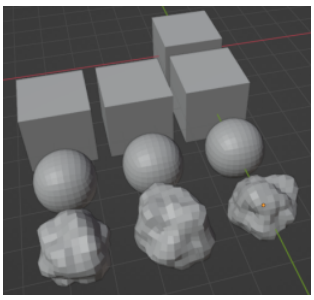
適切に設定をすると、メニューの中から自分の作成したジオメトリノードを直接呼び出して編集作業に使うことが出来ます。
この Tool タイプのジオメトリノードに関しては、Vol.4 の本で詳しく解説をします。

ジオメトリノードの基本

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。



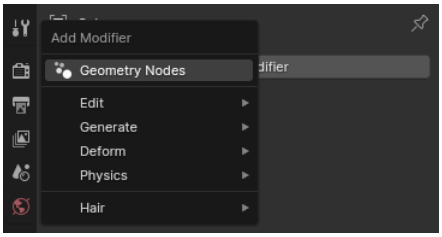
例えば、デフォルトキューブに対して

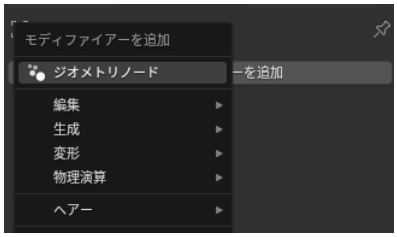
○ Array(配列)
○ Subdivision Surface(サブディビジョンサーフェス)
○ Displace(ディスプレイス)

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。

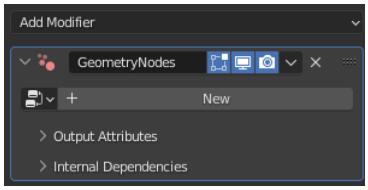
ジオメトリノードは、非常に高機能でカスタマイズが可能なモディファイア、と考えることができます。
モディファイアを追加するメニューの中に GeometryNodes の項目があるので追加をします。

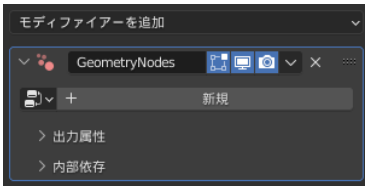




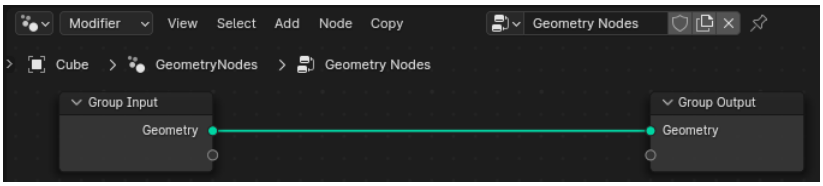
モディファイアのメニュー表示は、Blender 4.0 で更新されています。
ジオメトリノードは、メニュートップの選択しやすい位置に移動しました。

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。
これは、新規ノードをすぐに作ってしまうと、既存のノードを使いまわしたいときに「必要ない新規のノード」ができて邪魔なるのを回避する仕様です。
もし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。

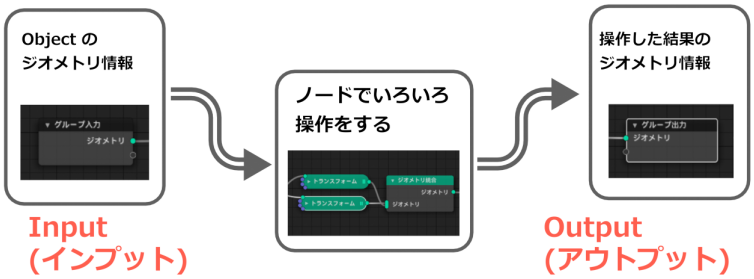


作成したノードの編集は、ジオメトリノードエディタで行います。
モディファイアとしてジオメトリノードを作成したので、
アイコン隣に表示されているモード選択が Modifier となっているのが目につきます。
（Blender 4.0 でモード選択が追加されました）

ジオメトリノードの編集

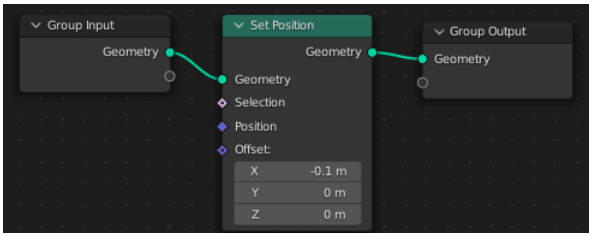
最初は緑の線が Group Input(グループ入力) と Group Output(グループ出力) のノード間をつないだ状態になっています。
ジオメトリノードの基本形は、Group Input から Group Output の間に、様々な操作を挟み込む形になります。
Group Input (グループ入力) の Geometry (ジオメトリ) ソケットは、元々のメッシュ形状の情報になっています。
今は何もせずに直接 Input から Output に繋がれているので「何もしないでそのまま」という状態です。

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。



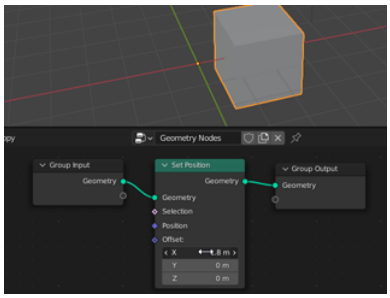
もっとも単純なジオメトリノード編集例

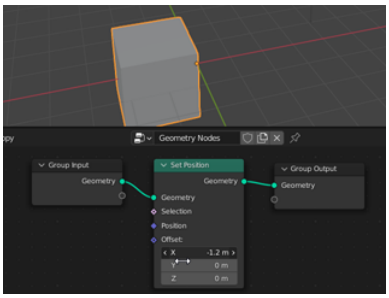
もっとも簡単なノード編集として、間に Add - Geometry - Write(書込) - Set Position(位置設定)を挟み込んでみます。



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。

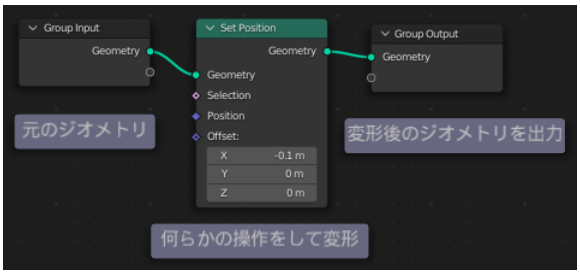




オブジェクト自体が動いているように見えますが、よく見ればオブジェクトの原点の位置は動いていません。
オブジェクトの位置はそのままで、メッシュが変形していることが分かります。

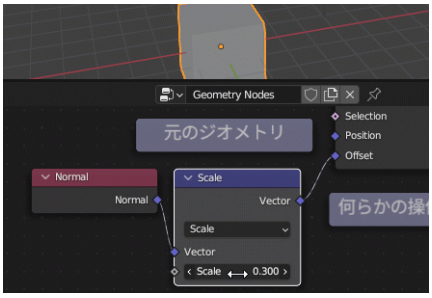
データの流れ(Geometry Node Fields)

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 - Utilities(ユーティリティ) - 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、法線ベクトルとパラメータの値を掛け算したものが 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、

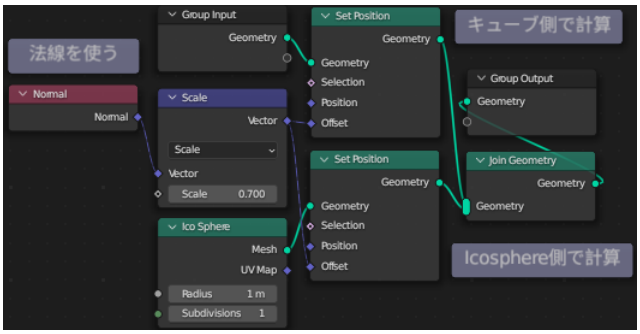
0. Set Position ノードで Offset のソケットに何かつながっている。
1. つながっているのはベクトルの乗算だ。AかけるBの結果になっている。AとBは何だ？
2. Aをたどると Normal(法線)だ。
3. Bをたどると 値(Value)だ。

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。

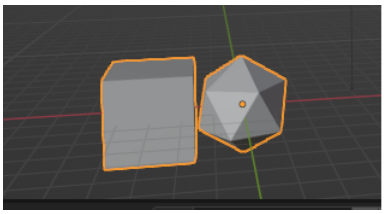
（公式サイトの Fileds に関する解説記事では、Callback という用語で説明がされています。
プログラマーであれば、コールバック関数という例えがしっくりくるかもしれません。
使うジオメトリの、「法線情報を得るという関数」が渡されている理解になります）

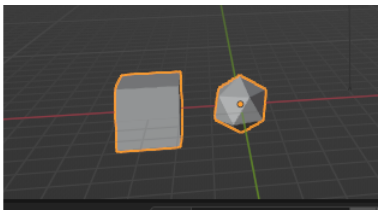
少しわかりづらい考え方なので、例を見てみます。
複数のメッシュを１つに合成する Join Geometry を使ってみましょう。
（重なってしまうので）デフォルトキューブを少し動かして脇にどけておきます。



ノードを使って Icosphere を原点に配置して、こちらも SetPosition を使って編集をします。
そして、Join Geometry でもともとのキューブと合成をします。

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。



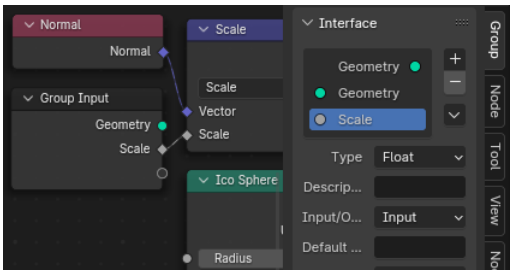


キューブと Ico球 それぞれで法線方向に膨らんでいます。
これは、Normal ノードが法線のデータそのものではなく「法線を使う」
という情報を表すためです。
（法線が実際に評価されるのは、緑のライン(ジオメトリ)がつながっている Set Position のノードです。
そこで、「キューブの法線」「Ico球の法線」がそれぞれ別々に評価されているわけですね。）

アトリビュートの入出力

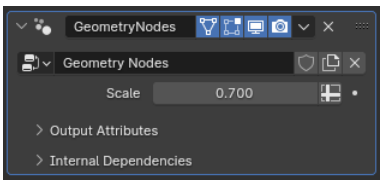
今までの例では、膨らませ方のパラメーターなどは、ノードの中で直接数値を編集していました。
それでは、「パラメータだけちょっと違う変形をする」ような場合などでもバリエーションの数だけノードを用意しないといけません。
そういう場合に、パラメータだけ変更する方法も用意されています。

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。

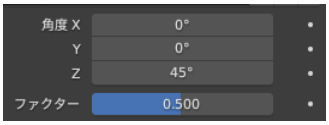
Group Input のノードの空のソケットにラインを繋ぐか、パネルの(+)ボタンメニューから、入力用のソケットを増やすことができます。
Scale のソケットからラインをつなぐと、同名(Scale)という入力ソケットが追加されました。



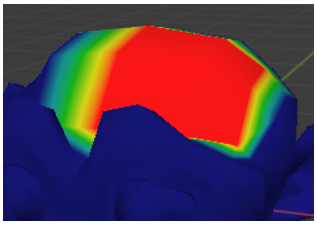
Group Input のソケットを増やすと、モディファイアのパネルにも対応する入力欄が現れます。
ここで、数値を入力することで、パラメータ違いのエフェクトなどを作ることができるわけです。

（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）

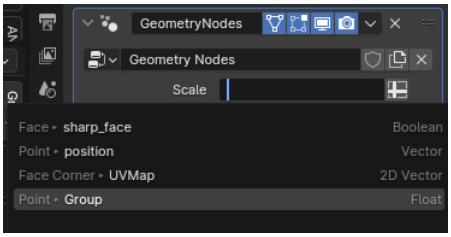
ソケットの種類は、デフォルトでは Float(つまり普通の数値) ですが、用途に合わせて別のタイプ、
整数であったり、オブジェクトであったり、画像であったり…に変更することができます。



また、Blender 3.6 では SubType(サブタイプ)の設定が追加されました。
サブタイプを適切に設定することで、角度を(ラジアンではなく)角度で入力したり、スライダー付きの入力をすることが可能になっています。

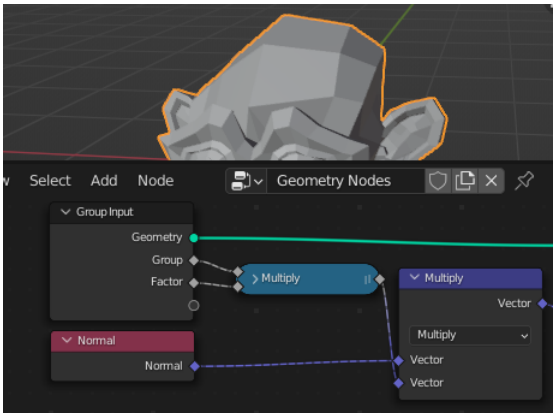


単一の数値だけではなく、頂点ウェイトやカラー属性のように、元のメッシュが持っている属性を入力にすることができます。
例えば、頂点ウェイト(デフォルト名 Group)を作って、一部だけ値を持たせてみます。



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。



入力のソケットで、頂点グループと、ファクターの数値を入力できるようにしました。
これをかけ合わせた量だけ、法線方向に頂点を移動するようにすれば、頂点ウェイトにしたがって膨らむようになります。

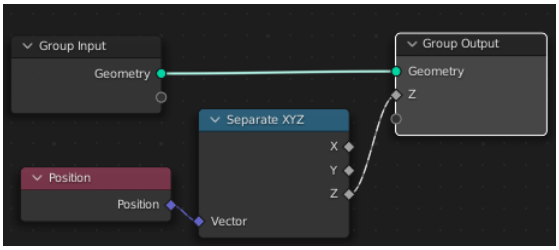
これは、つまり Displace モディファイアの機能をほぼそのままジオメトリノードで再現したことになりますね。
入力のソケットを複数使うような場合には、ソケットのラベル名がわかりやすくなるように、きちんと名前を編集しておくのがおすすめです。
（再利用時のわかりやすさが大きく違いますから…）

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。

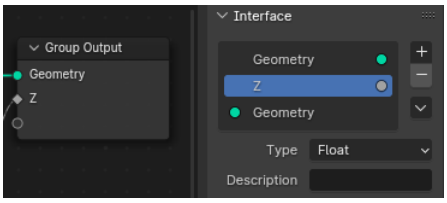
今までUVはUV、頂点ウェイトは頂点ウェイト、というように blender内部の仕組みはバラバラに作られていました。
ジオメトリノードの登場以降、こうしたパラメータなどはアトリビュート（属性）の一種として扱いが統一されつつあります。
ジオメトリノードで様々な操作を統一して行うことができるように改造の最中なのでしょう。
（Blender 3.x あたりのバージョンは、そうした統一の過程の過渡期になっているようです）
今後も、この流れが続いて、ジオメトリノードで扱うことの出来る属性が増えていくのではないか思います。

アトリビュートの出力

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。

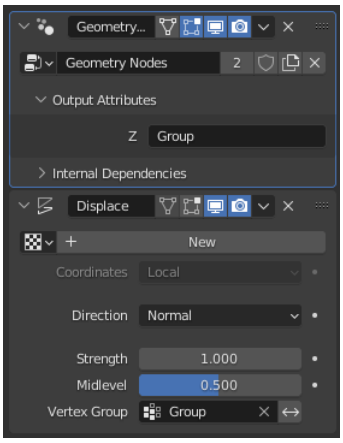


頂点の位置(Position)ノードから位置を得て、
Separate XYZ(XYZ分離)ノードで Z 成分を取り出しました。
Group Outuput の空ソケットにつなげると、出力用のソケットが出現します。



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。

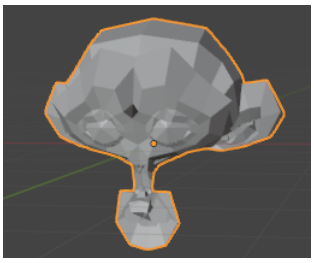


今、ラベル名 Z という名前で出力のソケットを作りました。

モディファイアのパネル側には、そのラベル名で
「この値を保存するためのアトリビュート名」を指定する項目が現れます。
ここで、名前を指定すれば「その名前のアトリビュート」に Z 成分が保持されるようになます。

ここでは、頂点グループ「Group」に Z 成分を保持するようにしました。

Displace モディファイアを追加して、そこで「Group」を利用するようにします。
このようにして、ジオメトリノードで計算した値を「ジオメトリノード以外」で使うことができます。



その結果、下の方では凹み、上の方では膨らむような変形が実現できました。

頂点グループは本来[0-1]の値しか取れないはずですが、
ジオメトリノードで書き換えたことによって、下半分ではマイナスの値を取ることもできています。

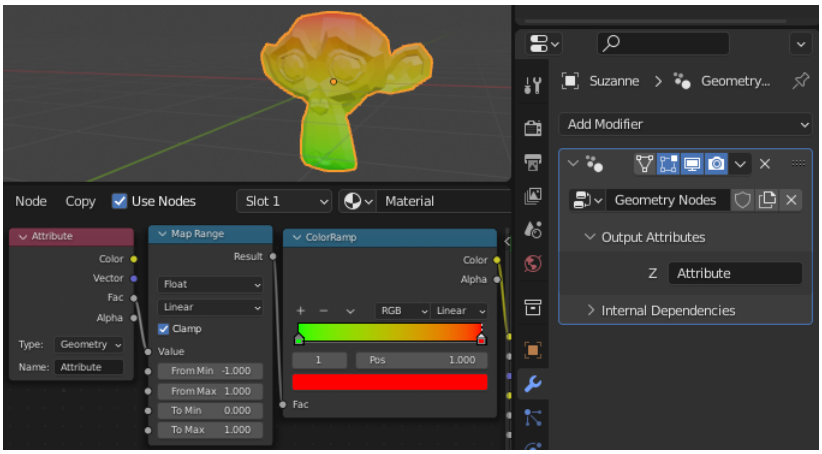
ここではあらかじめ登録された頂点グループ「Group」に対して出力をしています。 本来は頂点グループとは関係なく、新たに好きな名前のアトリビュートに出力をすることもできます。

ただし、(Blender 4.0時点で) Displace Modifier 側は「頂点グループ」しか使うことができません。
「頂点グループではないアトリビュート」を使おうとするとエラーになります。(というか入力できません)

モディファイアの方に、「頂点グループではない」一般のアトリビュートを指定ができる仕組みが（まだ）無いというわけですね。
そこで、「あらかじめ用意した頂点グループに対して出力をする」という設定をしたわけです。

シェーダーでのアトリビュート利用

ジオメトリノードから出力をしたアトリビュートは、シェーダーから利用することができます。



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るので
非常に強力な組み合わせです。

Blender 4.0 の時点で（以前からの挙動と同じように）シェーダーから頂点グループの情報を直接得ることはできません。
Attribute の方が(新しく作られてシェーダーへの対応がされたので)利用範囲が広いという逆転現象が起きているわけです。
ですので、ジオメトリノードからの情報の出力先を頂点グループにしてしまうとシェーダーで使うことができない問題が起きます。
いずれ内部的に完全に統合が進めばそうした問題は解決されそうですが、
現時点では例えば同じ出力を、新規のアトリビュートと、頂点グループの両方に出力しておく、などの回避手段が必要になることもあります。

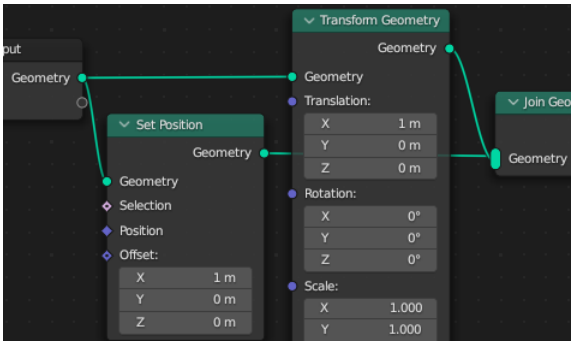
ところで、Group Input と Group Output を介してアトリビュートの入出力（データの読み込みと書き込み）をするのは、すこしまどろっこしいですね。
他の方法として、Blender 3.2 からNamed Attribute(名前付き属性)という別の入出力方法が実装されています。

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

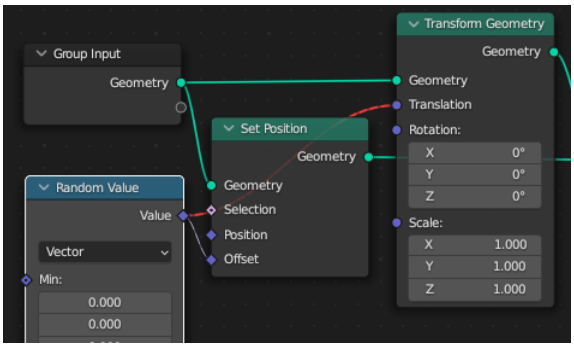
メッシュを平行移動で動かしたいときに、Transform Geometry (ジオメトリをトランスフォーム)を使うこともできますし、Set Position(位置設定)を使うこともできます。



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…

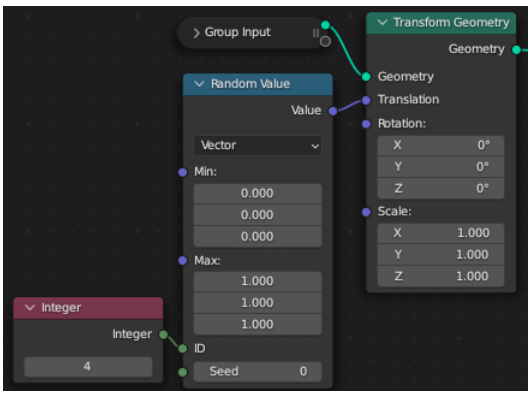


Random Value(ランダム値)などをつなぐと、各頂点に対してばらばらのランダムの値が与えられて、
頂点がそれぞれランダムに移動します。
このように、ただ１つだけの値ではなく「各頂点ごとのパラメータ」をやり取りする、
ということを表すのが菱形のソケットです。

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。
Set Position の Offset のソケットは、各頂点ごとにバラバラの値も受け取れますが、２つ上の図では (1,0,0)という単一のベクトルを受け取って平行移動をしているわけです。
その下の図のようにランダムな（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。

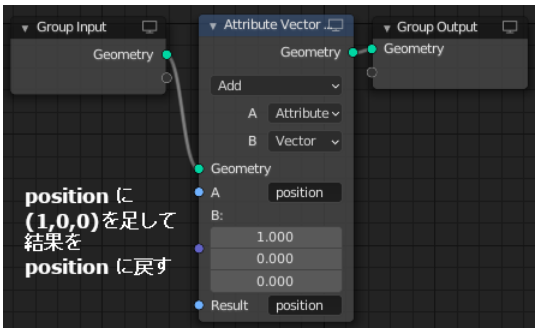
そういう場合は、Transform Geometry の丸いソケットにランダムをつなげることができて、 エラーにはならない…というわけですね。

Blender 3.2, 3.3 で追加されたノード

この章では、Blender 3.2 と 3.2 で追加されたノードを使った作例を主に見てみます。

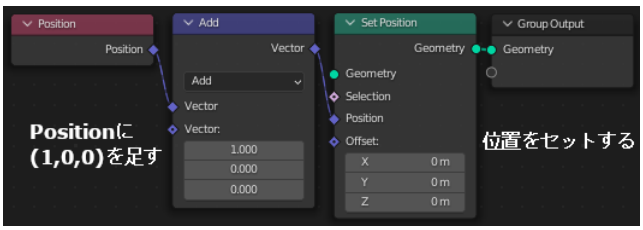
Named Attribute (名前付き属性)と Capture Attribute(属性キャプチャ)

Blender 3.0になる前、つまり Geometry Node Fields と呼ばれる方式になる前は、位置や頂点ウェイトなどのパラメーターの解決には、名前を使っていました。



2.93で平行移動を実現するノードの例です。
今のやり方だと、Set Position ノードなどを使ってパラメータを指定するところですが、
思いっきり文字列"postion"を使った名前解決をしています。

"position" という属性に (1,0,0)を足す、という操作が平行移動になっているわけですね。



今の Fields 方式であれば、このようなノードのつなぎになります。
(Offsetを使わずに、位置を直接計算してセットする場合のやり方です)

パラメーターの名前をユーザーが意識しなくて良いようになっています。

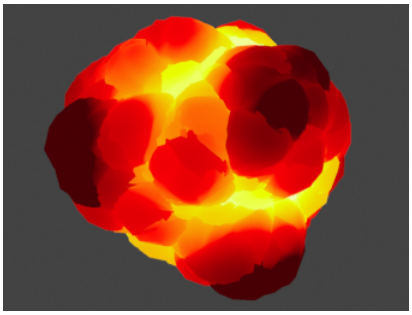
実は「使うノードの数が増えてしまう」という弱点もあるのですが、新方式の方がノードを組みやすいです。

使い比べてみないと、なぜ新方式の方が組みやすいのか？というのが実感ができないかもしれないですが… 旧方式は、仕組み上ひたすらノードを並べて順番に処理をするというノードの組み方になってしまうのです。 2.93までのジオメトリノードを使った経験のある人は、ノードが横に一列にどんどん伸びていく経験をしたのではないでしょうか？

しかし、実は利点と欠点は表裏一体で、
旧方式にあった「**ノードの間を線で結ばなくても名前で解決ができる**」いう利点が新方式では失われてしまいました。

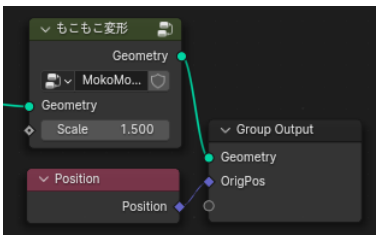
3.2 で導入された Named Attribute は、旧方式と同じように、名前で解決をするためのノードです。
新方式のノードの組み方に、旧方式の利点を持ち込めるものです。
（導入というか復活というべきかもしれません。）

とはいっても、実例が無いと分かりにくいメリットなので例を見てみましょう。
Vol. 1 の本にある作例で、爆炎のように色を付けたもくもくした形を作りました。



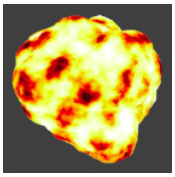
単純な Ico 球からノイズテクスチャを使って変形を行い、
同じテクスチャを使ってシェーダーでも色を付ける仕組みです。

しかし素直にシェーダーを組むと、ノイズテクスチャの評価に「変形後」の位置を使ってしまいます。
変形を制御するのは、「変形前」の位置で評価したノイズテクスチャです。
そのため、変形と色の関係がずれてしまうのです。
「変形前」の位置情報を、Attributeとしてシェーダーに渡す必要があります。

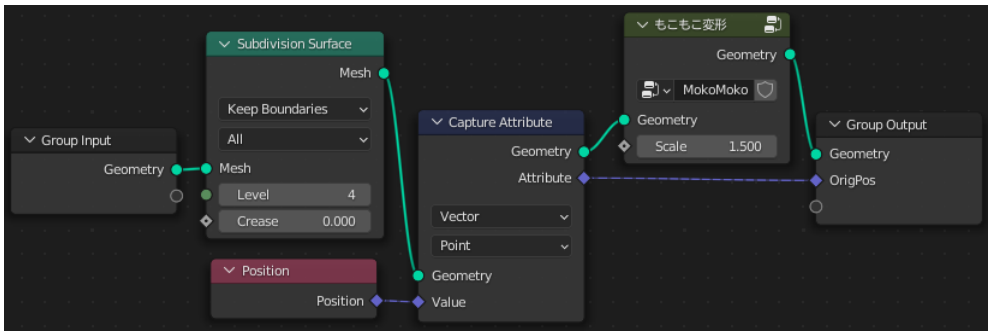


まず、もこもこと変形させる部分は、ノードグループ「もこもこ変形」としてまとめて、
肝心の部分だけはっきりと分かるようにしておきます。

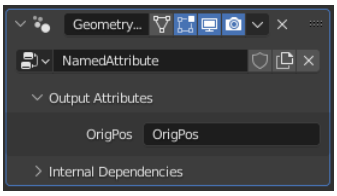
Group Output のソケットに、普通の Position ノードを繋いでしまうと、
「もこもこ変形」によって変形した**後**の位置情報が渡されることになります。



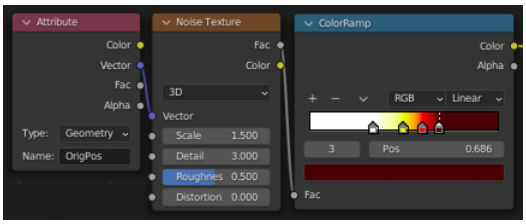
この位置情報を使ってシェーダーで色を付けても、単にノイズテクスチャが乗るだけです。
変形で大きく盛り上がった場所、そうでもない場所、に応じた着色をすることが出来ません。



そこで、Capture Attribute(属性キャプチャ) というノードを使って、変形前の「元の位置」の情報を先に保存しています。
この情報は「名前の無い一時利用のAttribute」として保存されていて、利用するときには Capture Attribute の出力ソケットを使います。
最後に Group Output と線を結ぶことによって、この「元の位置」情報を出力して、シェーダーに渡す仕組みです。



Group Output で出力した「元の位置」情報の名前を OrigPos にしました。
ラベル表示と実際の名前は別々に設定できることに気を付けます。
（便利といえば便利ですが、忘れやすいので要注意のところでもあります）



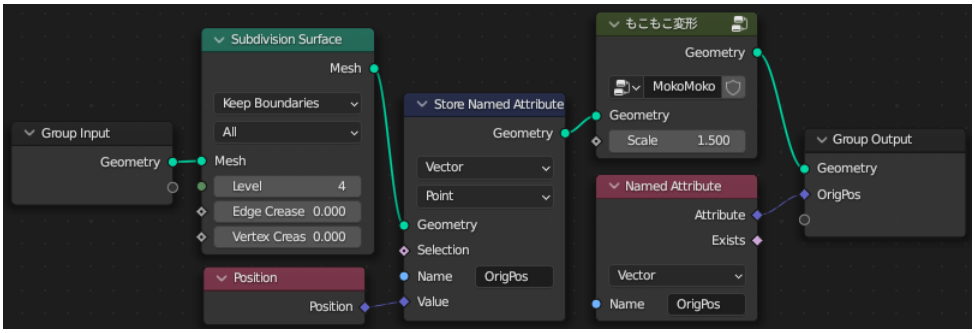
シェーダーの方で、この「元の位置」情報 OrigPos を使って、色を付けているわけです。
この時使うノイズテクスチャのパラメーターなどはもちろん合わせておく必要はあります。
（それらもAttributeとしてシェーダーに渡してしまう手もあります）

Store Named Attribute(名前付き属性収納) と Named Attribute(名前付き属性)

Capture Attribute(属性キャプチャ) の代わりに、

○ Attribute - Store Named Attribute(名前付き属性収納)
○ Input - Named Attribute(名前付き属性)

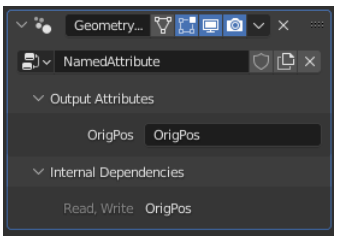
の2つのノードを使って作り直してみます。



名前を入力する欄には、OrigPos という文字を入力しました。
これで先のCapture Attribute(属性キャプチャ)を使った例と同じことが行えます。
OrigPos という名前の属性に、「元の位置」情報を収納しておき、
最後に Named Attribute で（OrigPosという名前を使って）呼び出しているわけです。

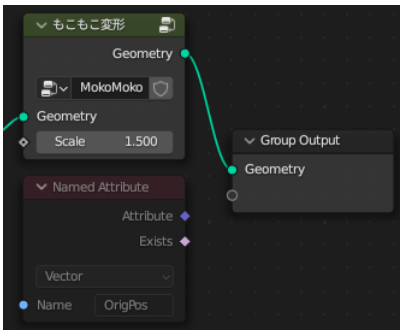
Capture Attribute を使ったときは名前を考える必要が無かった代わりに、ノードの間を線で結ばなければなりませんでした。
長大なノードを組んでいるような時に、距離の離れたノードを長ーい線で繋げなくてはならないので、整理が難しくなります。
今回は「もこもこ変形」部分をグループ化してまとめているので、距離が近いので良いのですが…

Named Attribute(名前付き属性)を使えば、名前で解決ができるので、ノード間をつなぐ必要が無くなりノードの整理がしやすくなります。



複数の Named Attriute を使って長大なノードを組むこともあるかもしれません。
内部で使われている Attribute についての簡単な情報は、Internal Dependencies のパネルに表示されて確認ができます。

Named Attribute にはもうひとつ利点があります。
上の図で見たように、ジオメトリノードから Group Output ノードを使って出力するアトリビュートは、モディファイアの設定として名前を付けないといけません。
上の図の例では「OrigPos という名前のアトリビュート」として出力をしたわけです。
これは、ノードを使いまわしたときに「別の名前のアトリビュートに出力できる」という利点もあるのですが、必要な操作が多くて多少まどろっこしい点があります。
実は、Named Attribute で設定したアトリビュートは、Group Output で出力しなくても、その名前のアトリビュートとして出力がされています。

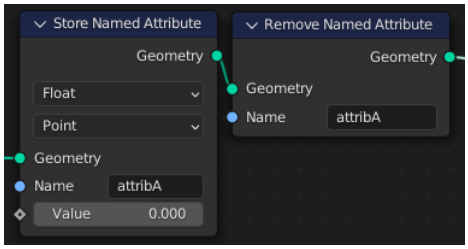


先の例では、今までのやり方を踏襲して最後に Group Output で出力をしたのですが、
実は最後の部分は無くても大丈夫だったのです！
（OrigPos を OrigPos に代入する…という2度手間を行っていたのです）

「アトリビュートの名前は固定で充分、ユーザーが指定をする必要が無い」
というような場合には、Named Attribute を利用した方が全体的に必要な設定が少なくなるので、
慣れるとお勧めの方法です。

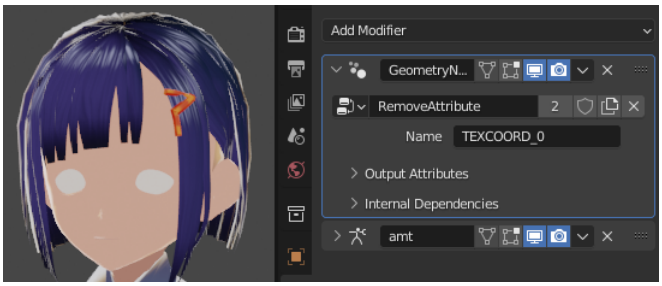
Remove Named Attribute(名前付き属性削除)

こうして作成された名前付き属性(Attribute)を削除するノードも一緒に用意されています。
Attribute が追加されたメッシュは、その後様々な変形などをする際に、その Attribute に関しても計算が行われます。
重ーい処理などで、少しでも演算量を減らしたい場合には、既に必要がなくなった Attribute を削除しておいた方が、パフォーマンスが良くなることになります。



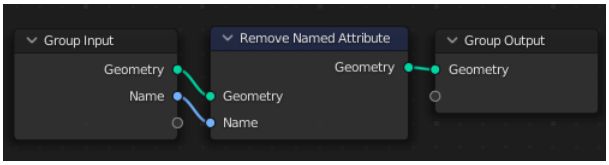
ただ、逆に言えばパフォーマンスに問題が無いような場合には Attribute の削除をしたい状況というのがあまり思い浮かびません。
ほったらかしにしておいても（ユーザーとしては）問題が無いので、このノードの使い道はそんなに無いかな…？

とも思えたのですが、ジオメトリノードの枠から外れて考えてみると、UV や 頂点ウェイトなども属性として扱うことができます。
そのため、それらの属性を（名前を指定するだけで、元データを弄らすに）クリアするために使うことができます。



3Dモデルから UV座標を 取り除いてみました。
単色の表示になっています。(0,0)位置でのテクスチャ色の表示になっているのでしょう。
Group Input での入力は文字列も使うことができるので、
別の文字列を指定をすれば別の属性をクリアすることもできます。

(※ このモデルは VROID STUDIO から出力したモデルを使わせていただきました。
データの中身を見てみると、UV座標の名前が TEXCOORD_0 となっていたので、それを削除してみたわけです)



Group Input の文字列でアトリビュート名を指定するように
ノードをつないでみました。

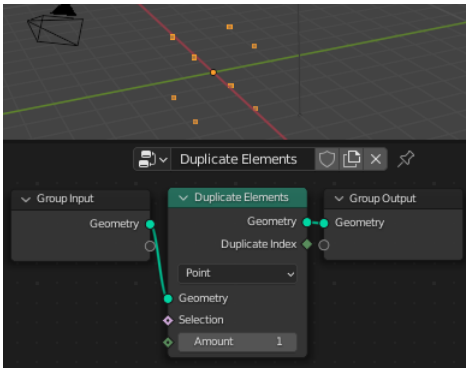
このほか、Attribute を取り除くことで出来るような用途を考えてみると…

○例えばカラー属性を取り除いて素の状態(=黒を設定したとき)の表示を確認する
○アーマチュア用の頂点ウェイトの一部を取り除いて（例えばひじパーツから先を動かなくして）故障したパーツがある表現をする

といったことを（元データを弄らずに）モディファイアを追加することで行えることになります。

Duplicate Element(要素複製)

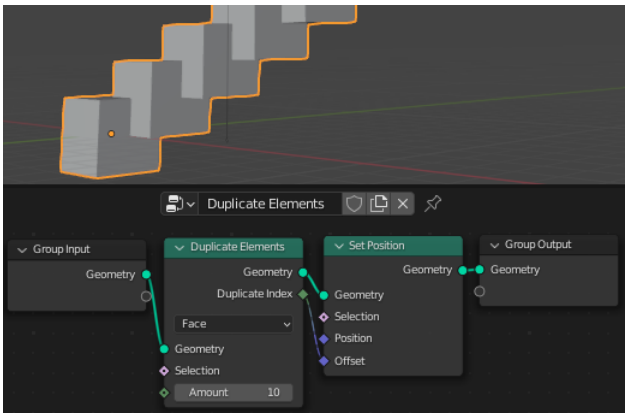
3.2 で導入された Geometry - Operations - Duplicate Elements(要素複製)は、その名の通り複製を作成します。



面白いのは、メッシュそのものを複製するだけではなくて、
Point(ポイント) や Edge(辺) といった要素のみの複製も可能なところでしょう。

デフォルトキューブを１つ複製した時、ポイントを指定していれば、このように点だけが複製されることになります。
単純にメッシュ全体を複製したいときはFace(面)を選べばよいですし、
カーブで使用するときには Spline(スプライン)単位での複製などができるわけです。

※) 面を複製するときは、面ごとにばらばらに複製されるので、Edge Split されたようなメッシュになる点に注意です。
スムーズシェーディングなどが意味をなさなくなります。

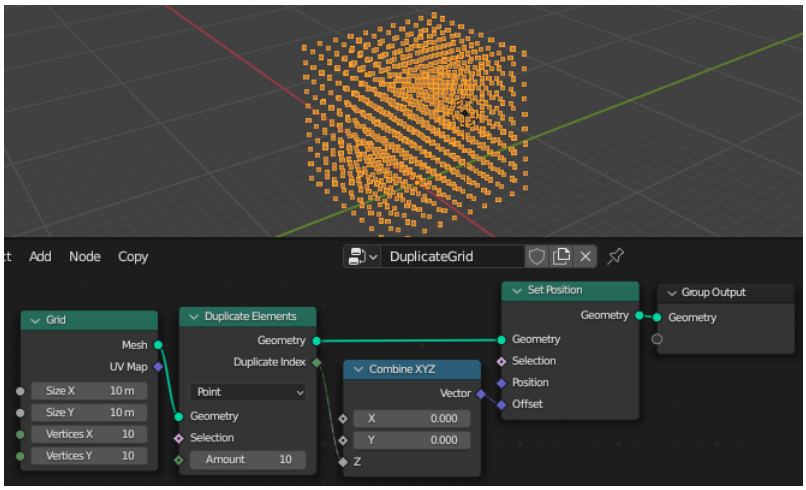


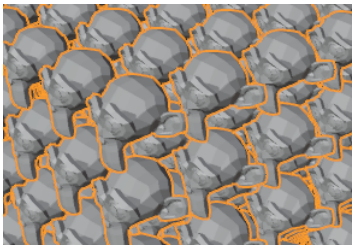
このノードの機能は「複製だけ」です。
１０個複製しても、そのままだと全部同じ場所に重なって表示されるだけで、見た目が変化しません。

通常は、Set Position(位置設定)などを使ってずらして配置をして使うことになります。
この時、もう一つの出力ソケット Duplicate Index(複製のインデックス)が、「何番目の複製か」の情報を持つことになります。

数値をそのまま ベクトル入力の Offset につなぐと、x,y,z成分全部に対して効くので、
(1,1,1) (2,2,2) …
のようにずれて配置されて、斜めの階段状になります。

他と組み合わせて使う素材として、3次元グリッド状に配置を作成してみましょう。
X-Y軸方向に広がる2次元のGridを作成して複製し、Z軸方向に移動させるだけです。



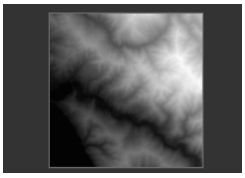


Instance on Points(ポイントにインスタンス)を使って、
頂点の位置にスザンヌでも配置をすればこの通りです。

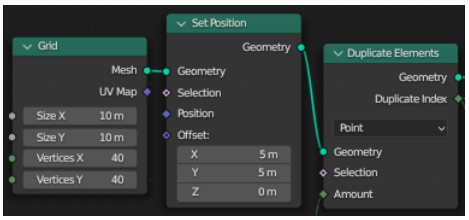
格子状の配置のような基本的な形状は、他のノードなどと組み合わせて
様々な使い道があるでしょう。

Blender 3.4 では、Distribute Points in Volume が導入され、ボリューム内にグリッド状（またはランダム）に点を配置することができます。
上のように自力でポイントを配置するようなノードを組まずに、キューブからボリュームを作り、ボリューム内にポイントを配置する手順で、
もっと簡単に三次元グリッド状の配置をすることができます。

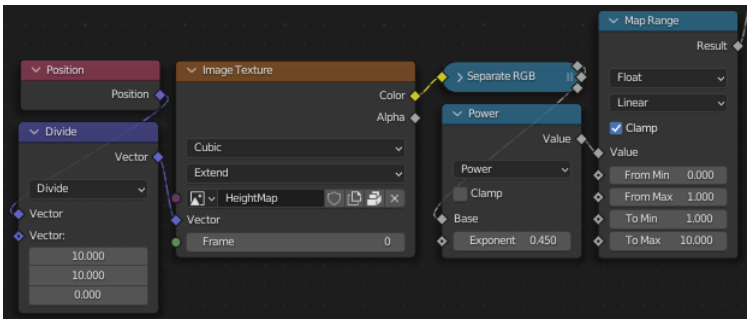
マインクラフト風味の地形



複製で作れる形の例として、
キューブを複製してマインクラフト風の地形を作成してみます。
元になるのは、高さ情報を得るための地形図テクスチャです。

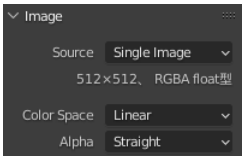


10m幅の40x40のグリッド状に配置をする元になるグリッドを作りました。
テクスチャは[0-1]の範囲なのでそのままだと対応がつけずらいです。
5mだけずらして[0-10]の範囲になるようにします。10で割れば[0-1]の範囲になります。

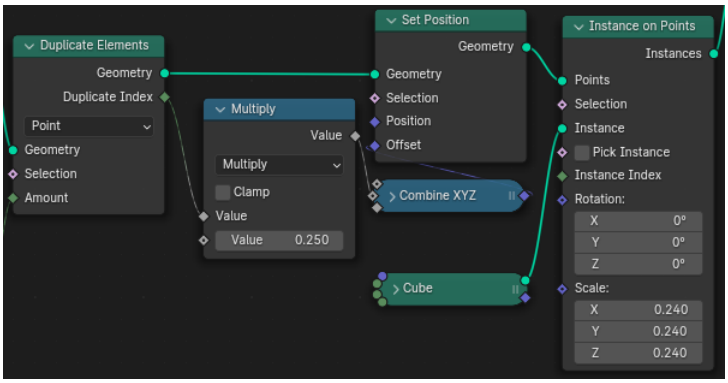


テクスチャに従って複製する数を決めます。
明るいほど多く頂点が複製されます。

ところで、画像テクスチャの明るさは、RBGの数値に比例しているのではなく、
sRGBの変換式に従って変換されているはずです。
一方高さ情報は、RGBの数値そのものに対応しているのが普通です。
近似式による変換で、明るさをRGBの数値に戻すように 1/2.2 (大体 0.45)
のべき乗で変換をしました。



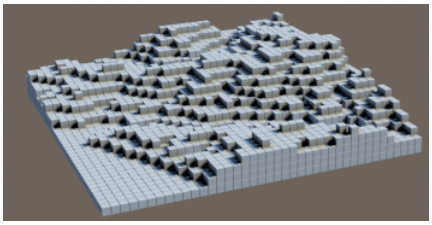
もしくは、画像の方の Color Space(色空間）の設定を
sRGB ではなく Linear(リニア) にしておくという手もあります。



複製された頂点を、複製のインデックスに従って上に移動します。
そして、グリッド位置に Instance on Pointsを使ってキューブを配置します。

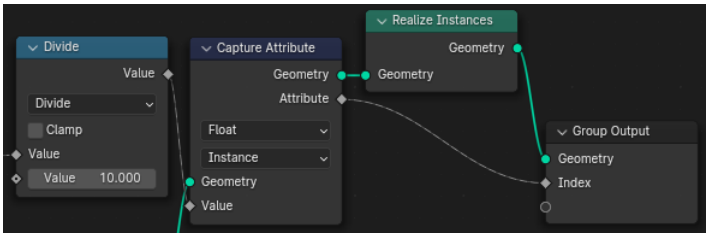
10mで40グリッドになるように作ったので、ひとグリッドのサイズは0.25です。

移動のスケールや、キューブのサイズがそれに従うように、
パラメータを調整します。



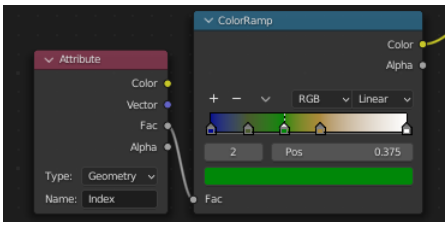
テクスチャの明るさに対応した高さにブロックが積まれました。
単色なので、少々寂しい感じです。

高さ(つまり複製のインデックス)に応じてブロックに色を付けてみます。
地形の**各頂点**が、ブロックの高さ(インデックス)情報のアトリビュートを持つようにしてみます。



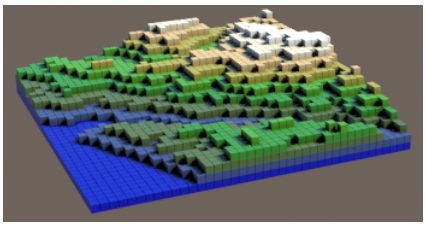
複製のインデックス情報をシェーダーに渡すためのノードを組みます。
最大で高さ10ブロックになるようにしているので、
0-1の範囲になるように10で割っています。

Capture Attribute で情報を保持した後で、インスタンスを実体化します。
(これで、実体化したメッシュに Attribute が引き継がれます)



シェーダーの設定で、渡された Attribute の値にしたがって色をつけます。

（モディファイアの設定としても、Group Output で出力した Attributeの名前を設定する工程を忘れないように…）

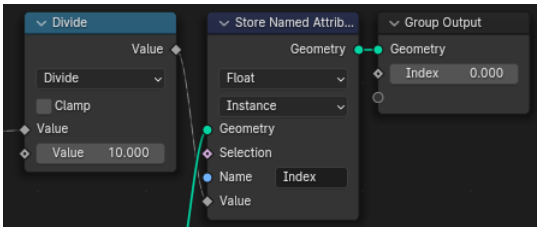


マインクラフト風味の地形図になりました。

この例は、01_DUPLICATEELEMENTS/01_DuplicateElement_Map.blend として同封しました。

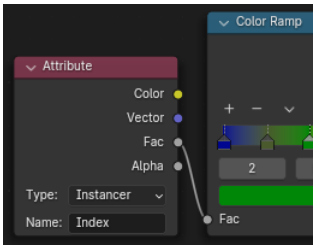
Blender 3.3 までは、メッシュの各頂点（や面コーナーなど）が持つアトリビュートをシェーダーが使うことができるという仕組みでした。
そのため、Realize Instance(インスタンス実体化) を使用してインスタンスからメッシュに変更し、そのメッシュの各頂点が色情報をもつという仕組みが必要でした。

Blender 3.4 からは、シェーダーが Instance の持つアトリビュートを直接使うことができるようになっています。

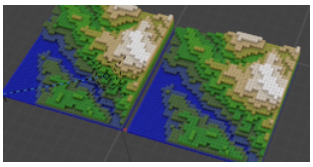


Store Named Attribute でドメインのタイプを Instance にします。
これで、インスタンスごとに持つアトリビュートとして、
高さ情報(複製のインデックス情報)を保持できます。

インスタンスの実体化をする必要もなくなっています。



シェーダー側のアトリビュートノードで、Instancer(インスタンサー)を選ぶことで、
インスタンスごとのアトリビュートが利用できます。
言葉の意味（インスタンスでは無くインスタンサー）で考えると、インスタンスを配置する配置元、
今回は複製をしたグリッドの持っていたアトリビュート、というニュアンスでしょうか。

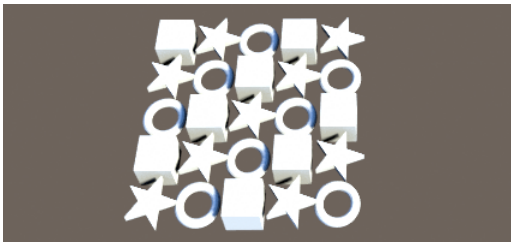


サンプルファイルには、２つの場合についてのサンプルを並べて配置しました。
理屈の上では、構成するキューブをインスタンスのまま扱っている後者の方がパフォーマンスは高いはずです。
（この例程度の表示では、差は実感できないと思いますが…）

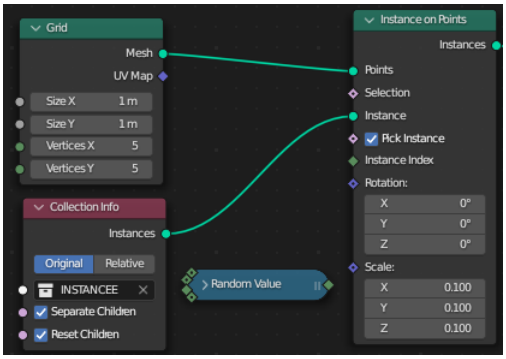
インスタンスの数調整

数種類のオブジェクトから選んでインスタンス複製をしたいことがあります。
そういう場合にはコレクションを利用して、コレクション内にあるオブジェクトを複製する手段がありました。

3種類のオブジェクトを作成して、コレクション内にまとめておきます。



グリッドを作成して、Instance on Points を使ってグリッド上にインスタンスを作成します。
この時に、Collection Info(コレクション情報)を使って3種類のオブジェクトの複製をするようにします。



ノードの設定をきちんと仕込む必要があるのですが、
Separate Children(子を分離) コレクション内のオブジェクトをバラバラに扱うようにする。
Reset Children(子をリセット) コピー元のオブジェクトは普通は（原点に重ねて置かずに）並べて置いておくので、位置情報が邪魔になるのを、原点に置いてあるものとして扱う
Pick Instance(インスタンス選択) バラバラのオブジェクトを順番に使う

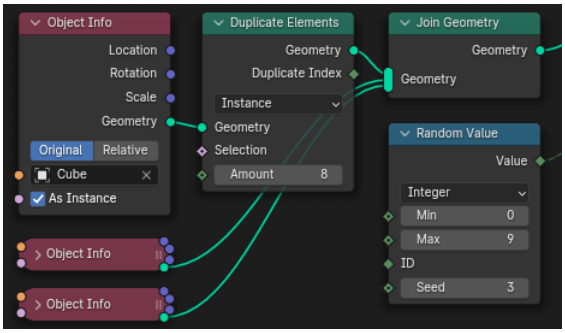
という3つの設定をチェックしすれば、上の図のような配置になります。

上の図では順番に複製されるので、規則正しく並んでいますが、
Instance Index(インスタンスインデックス)のソケットにランダムを繋げることで、
ランダムに配置することもできます。

この時、オブジェクト１を3個、オブジェクト２を２個、というようにランダムの比率を調整することは簡単にはできません。

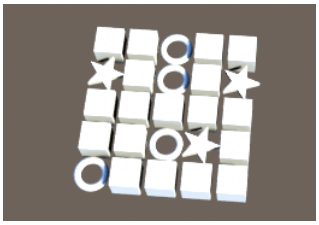
何番目のオブジェクトを使うか、という設定は Instance Index のソケットに数字を送ることによってできるので、
ノードを使ってうまく組み立てればできるのですが…かなり面倒くさいのです。
Utility - Float Curve(Float カーブ) を使ってグラフで編集もできますが、直感的に数値で指定することが難しくなります。
テーブルなどを使って、「0 の時は 0 を、 1 の時も 0 を、2 の時は 1 を…」
というような対応関係を作ることが出来れば簡単そうなのですが、ノードでそれをするのはとても面倒くさいのですね。

そのため、少しだけ星のオブジェクトが混ざっている、というような表現が難しいことになります。
（コレクション内にキューブを10個、星を１個置いておくという手はありますが…カずくという感じでスマートではないですね）
そこで、Duplicat Element(要素複製)の、インスタンスを複製するモードを使うことにしましょう。



Object Info(オブジェクト情報) を使って、オブジェクトを１つを用意します。
この時、As Instance(インスタンスとして)にチェックを入れると、文字通りインスタンス扱いになります。
そのため Duplicate Elemens(要素複製)を Instance で使うと複製で数を増やせます。

それらは、Join Geometry で統合することで、コレクション内の複数のオブジェクトと同じように複製ができます。
図の場合は、キューブが8個、星が1つ、トーラスが１つの計10個になるわけです。
0-9までのランダムを Instance Index に使えば、少しだけ星とトーラスが混じった配置が実現できます。



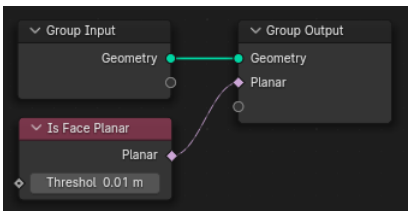
Join Geometry で複数のインスタンスをまとめたジオメトリは、
Collection info の代わりに Instance on Points のノードにつなげて使うことができます。

星やドーナッツが少量混じった配置になりました。
この例は、01_DUPLICATEELEMENTS/01_DuplicateElement_Instance.blend として同封しました。

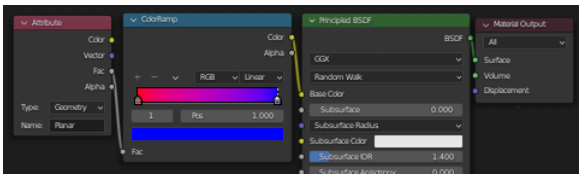
この仕組みができる前は、インスタンスの比率を変えて配置するようなノードを組むのは煩雑で面倒くさいものだったのですが、
Duplicate Elements でそうしたノードをシンプルにすることができます。

Is Face Planar (平面判定)

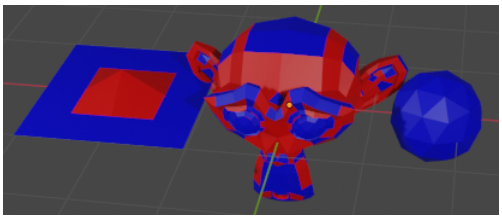
Mesh - Read - Is Face Planer (平面判定) は、文字通り面が平面で構成されているかどうかを知ることができるノードです。



三角形のポリゴンは数学的に必ず平面になるのですが、4角以上のポリゴンは平面になっているとは限りません。
ワザと非平面にすることもあるかもしれませんが、大抵の場合はうっかり非平面にしてしまったという場合です。
平面かどうかをチェックすることができます。



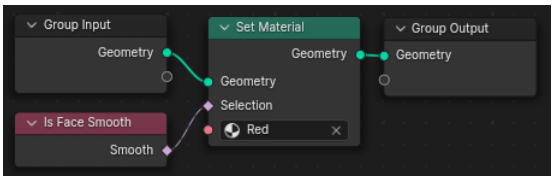
Planar という名前のアトリビュートとして出力をして、色分けをするという単純な仕組みを作ってみました。
今回の作例では、Named Attribute を使わずに、Group Output ノードを使って情報の出力をしています。



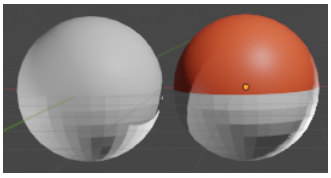
平面ではない面が赤く着色されています。
この場合は、(頂点ごとではなく)面ごとに情報が欲しいところですから、
Attribute Domain の設定を Face(面)にしています。
（頂点にすると、補間された感じの着色がされることになります）

Is Smooth (スムース判定)

Mesh - Read - Is Smooth ノードは、新たに追加されたノードではないのですが、Is Face Planer と似ているのでここで解説をすることにします。
その名の通り、レンダリングのスムース設定がされているかどうかを判定するためのノードですが、Blender 4.0 でさらに、
Is Face Smooth(面スムーズ) と Is Edge Smooth(辺スムーズ) という2つのノードに変更されました。

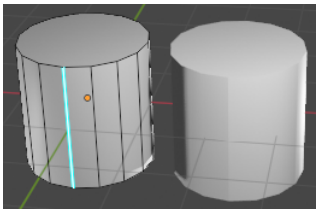


Is Face Smooth(面スムーズ) は、面がスムーズシェードかフラットシェードかを判定します。
例えば、Smooth シェードされている面を別のマテリアルに変更するようなノードであれば、単純にこのように組めます。



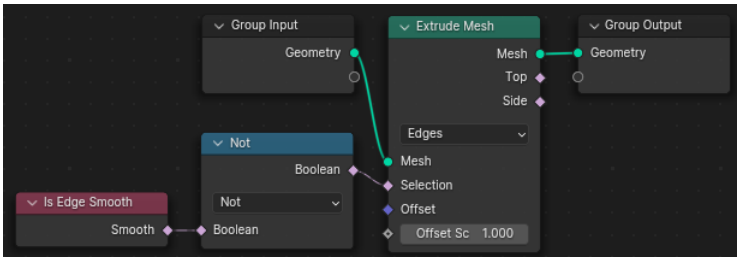
上半分をスムーズなシェーディング、下半分をフラットなシェーディングにした球です。
上のジオメトリノードを右側の球に使いました。
結果は、まぁ予想できる通り、スムーズな部分に違う色のマテリアルが付きました。

Is Edge Smooth(辺スムーズ)の方は、少しややこしいのですが、Auto Smooth が有効になっている時に使う属性です。

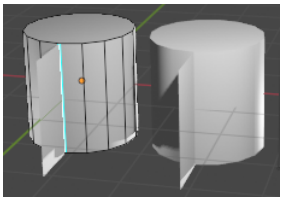


例えば円筒に対して Auto Smooth を有効にしてしきい値の角度をうまく設定すると、側面は滑らかに、上下の面はフラットに描画されます。
この時、特定のエッジに Sharp 設定をすると、そのエッジをフラットに描画することができます。
(この図は、分かりやすくするために16角形のかなり粗めの円筒を使っています)

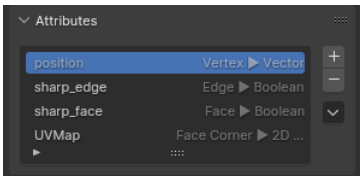
複製した右側のオブジェクトでは、Sharp 設定した辺が、フラットな描画で明るさの変化が滑らかではない表示になっています。



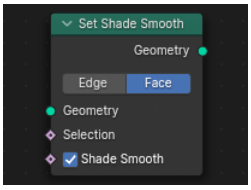
Is Edge Smooth は Sharp 設定した辺の情報を得ることができます。
といっても、Smooth かどうかを得るノードなので、Sharp 設定は逆です。
Sharp かどうかをの判定のために、ブーリアン演算の Not を使って逆転させました。
Sharp な辺を Extrude Mesh の Edge モードで伸ばしてみます。



Sharp 指定された辺から、Extrude Mesh によってひれが伸びました。



実は、これらの属性は汎用アトリビュートとして組み込まれているので、
sharp_edge および sharp_face として名前付き属性などでも利用することができますし、書きこみも可能です。



Is Smooth ノードが Face と Edge の区別をきっちりするようになったのに伴って、
Set Shade Smooth ノードも、Face と Edge の2つのモードの切り替えができるようになりました。
それぞれ、sharp_face と sharp_edge のアトリビュートに書きこみをするのと、内部的な処理は同じことになります。

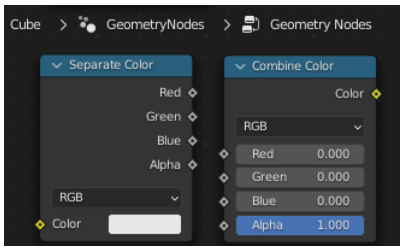
ノードは、スムーズかどうか、アトリビュートは逆にシャープかどうか、という名前になっているのも注意点です。
読み書きで使う True/False の値も当然逆になるので、True/False を反転させる ブーリアンの Not ノードの出番が増えます…！

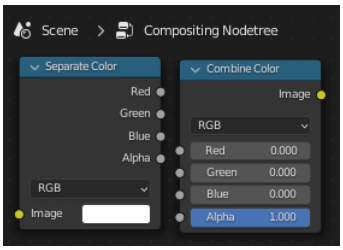
sharp_edge の属性は、Blender 4.0 の時点では**Auto Smooth が有効になっている時に**使われる属性だということに注意してください。
この Auto Smooth の振る舞いは、今後変わることが計画されています。
Auto Smooth が廃止になり、sharp_face と sharp_edge のアトリビュートでスムーズとフラットをすべて管理するようになる予定です。

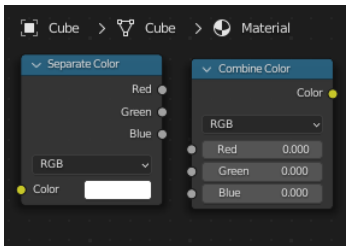
※ Auto Smooth の自動でシャープとフラットを割り振る機能が無くなってしまったら不便では…？
という疑問が当然湧きますが、角度に応じて sharp_face と sharp_edge を設定するためのノードが追加され、Auto Smooth の代わりになる予定のようです。

Separate Color, Combine Color(カラー分離、カラー合成)

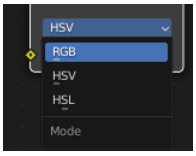
ここから先は Blender 3.3 で増えた機能について見てみます。
色を成分ごとに分けたり(Utilities - Color - Separate Color)、分離した成分を合成したりする(Combine Color)ノードが新しくなっています。
地味な更新ですが、今まで マテリアルノードやコンポジットにおいても、似た機能のノードがあり、
それぞれ少しづつ違ったりとばらばらに実装されていたものが、統一されてすっきりしました。
一応スナップショットを並べて見てみます。（ほぼ）同じノードになっていますね。





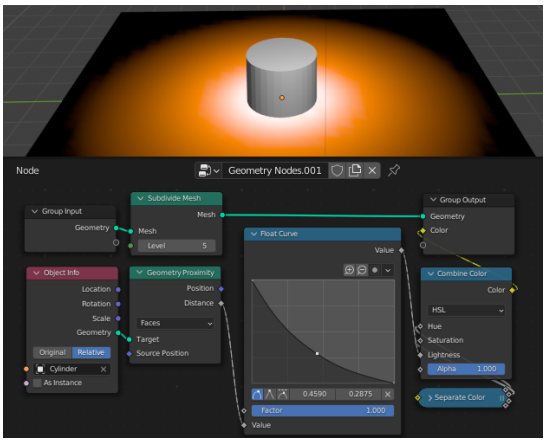


こうしてみるとマテリアルのシェーダーだけ Alpha 成分が無いですね。
（むしろアルファ値も同時に扱っている コンポジションとジオメトリノードのほうが、ちょっと拡張された仕組みになっている、と考えるべきなのでしょうか）



モードとして、RGB, HSV, HSL が用意され、どのノードでも色相や彩度での色の調整が可能になっています。

Blender 3.2 まで、ジオメトリノードでは RGB成分での分離合成しか用意されていなかったので、種類が豊富になったのは純粋な機能の拡張です。



HSV, HSL はともに色相と彩度を使って色を調整することができるモードです。
HSVとHSLでは、HSVの方がより広く使われる方法です。
色調補正などでも、HSV, つまり Hue, Saturation, Value の3つのパラメーターを使って色補正をします。

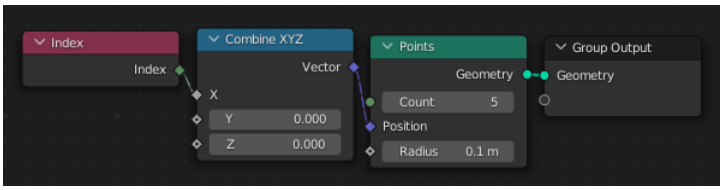
HSLモードは、L(Lightness)が 1 の時に白になる(0.5の時に原色)という特徴があります。
明るすぎて色が白にサチっていくような表現を値 1 つで表現ができます。
試しに、距離によってカラー属性が変わるようなノードを HSL の L が変化する場合で組んでみました。

こういう中心のエネルギーが高いようなグラデーション表現は、
色相、彩度を固定させて、明るさを変化させたいような場合はあまりなく、
温度変化で黒-赤-黄色-白、と黒体輻射の色変化のように変えたい場合が多いでしょうが…

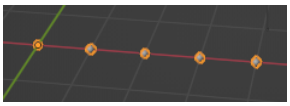
今回の更新で自由度が増えて統一感が出てきたので、地味に使い勝手が良くなったと思えばよいと思います。

Points(ポイント)

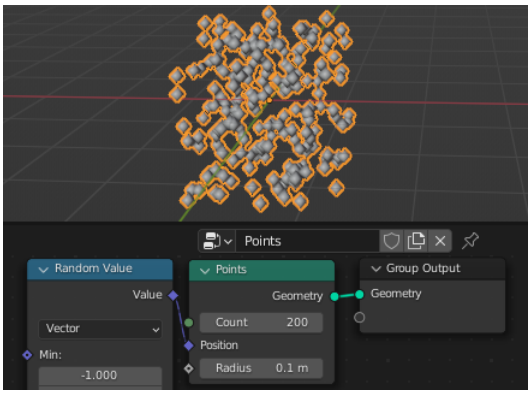
Point - Points(ポイント)は、文字通り「点」を直接作成するノードです。
これまでポイントが欲しい時にはメッシュの頂点にポイントを置いたり、メッシュの表面にランダムにポイントを配置することをしてきました。
これによって直接的に位置を指定して点を配置することができるようになりました。



Count(数)で指定しただけのポイントを作成します。
とはいえ、各点の位置を指定しなければ全部原点に重なるだけです。
ポイントの区別は Index でつくだけですから、Index で位置が指定できるようにノードを組みます。

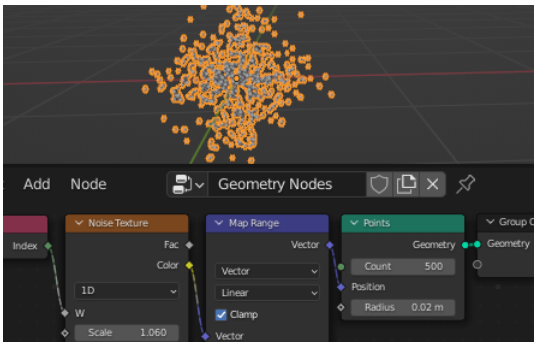


例えば上の図のように Index と Combine XYZ ノードをつなげば、直線状に点を配置できます。



わかりやすい使い方としては、Random Value を Vector として作成して、
[-1,-1,-1]から[+1, +1, +1]の範囲を指定すればランダムな空間配置を作れます。
ただし、このやり方は見ての通り四角い箱の中に一様にランダム配置したようなことになります。

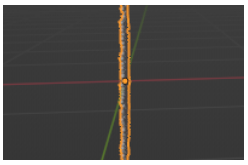
四角ではなく、球状で中心に密度が集まっているような分布の方が自然な感じです。
このランダムをもとに変形して、空間的な正規分布になるようにノードで計算する、
といった数学的に正しそうなアプローチをとることも可能でしょうが…
１次元のノイズテクスチャで、簡単にもう少し自然なランダム的分布にすることもできます。



ノイズテクスチャの1次元の入力に Index の値を使い、出力に色ソケットを使います。
色は当然[0-1]の範囲なので、Map Range (範囲マッピング)で [0-1] の範囲を[-1, +1] の範囲に（ベクトルで）変換をします。

四角い形状ではなくなり、だいぶ自然になりました。

ただし、このノイズテクスチャは、本当にランダムなのではなく、Perlin(パーリン)ノイズといわれる、少ない計算回数で自然な雰囲気のノイズになるように作られている疑似ランダムです。
周期性があって、その周期性のスケールと入力(今回はIndex)がうまくはまるとパターンがばっちり見えることになります。



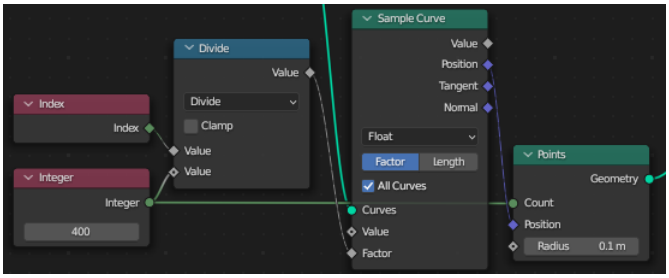
スケールのパラメーターを動かして見ると、あるところでばっちり噛み合って、
完全に平面的な分布になってしまいました。
(入力にIndex(整数)を使うと、スケールが丁度整数の時にこうなります)
こういうことが起きることがあると知っておいて、少し注意をして使う必要はあります。

回避方法は、スケールを少し変えるか、ノイズテクスチャのノードにDistortion(歪み)のパラメーターを与えると、
さらに別のルールで結果にランダム性が乗るので、 それを利用する手もあります。

ポイントとカーブに沿った配置

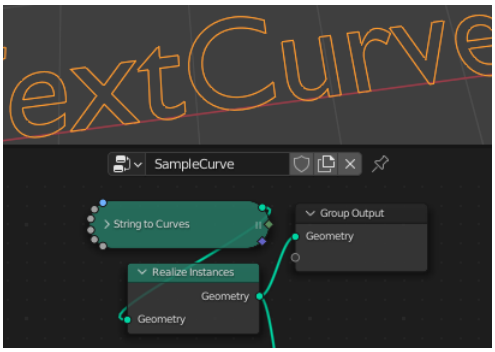
ポイントノードはノード１つで「任意個の点を作れる」のが利点です。
利点を生かした使い道を考えてみると、「カーブに沿った配置」が相性が良いようです。
Vol.1 の本でカーブを使った作例としても追加したのですが、ここでも軽く触れてみます。

Sample Curve のノードを使うと、カーブ上の任意の位置を得ることができます。



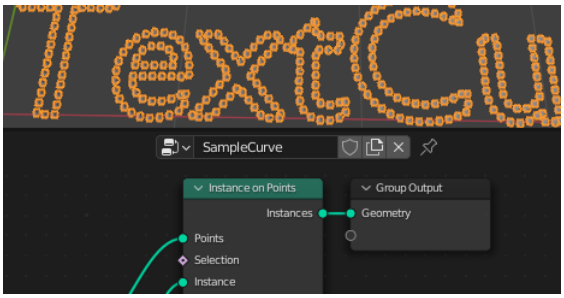
位置は、全カーブをまとめて[0-1]の範囲で表現すると都合がよいので、All Curves にチェックを入れました。
ポイントの数に応じてインデックスを総数で割ります。

indexに応じて Sample Curve ノードで評価した点の位置を指定することで、
カーブ上で等間隔に配置した点が表現できます。



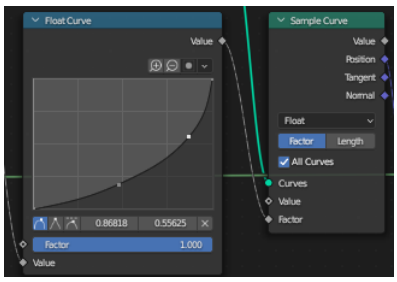
配置に使うカーブは例えば Strings to Curves などを使えば複雑な形もすぐに作成できます。
もちろん、フォントに応じて頂点の間隔もまちまちでしょうから、これを利用して配置を行いたい場合には、Sample Curve や Resample Curve などを利用して間隔を調整することが必要です。

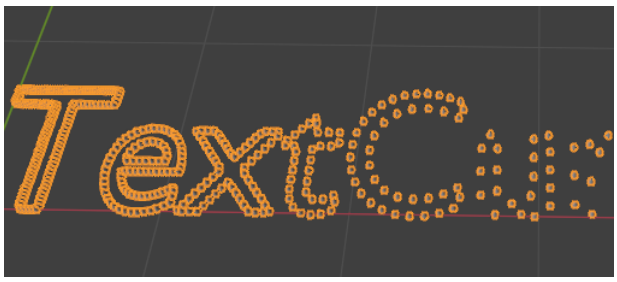
作成されたカーブは、最初はインスタンス扱いです。
（テキストなので同じ文字を複数回使ったりするので、そのほうが効率的です）
Realize Instances を使って実体化をしておきます。



点に沿ってインスタンスを配置すれば、カーブ上の等間隔配置ができます。
ただし「等間隔にする」だけであれば、Resample Curve ノードを使ってカーブ自体を等間隔化する方が簡単です。
わざわざ Sample Curve の方のノードを使った利点は、入力するFactorのソケットを調整すれば、分布の偏りを作れることです。

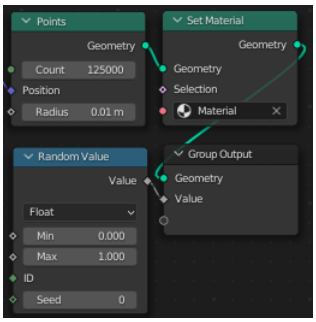
Factor につなげる前に Float Curve を使って分布に偏りを入れてみます。





密度分布に偏りを持たせてインスタンスの配置をすることができました。
この例は、02_POINTS/02_SampleCurve.blend として同封しました。

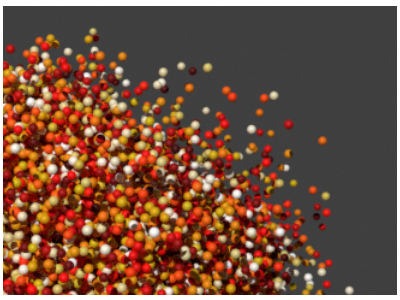
ポイントの表示



ところで、ポイントは、Eeveeでは表示がされません。
そのため、Eeveeを主体に使っているとポイントそのものを使う機会が無く、
ポイントを元にさらにインスタンスを配置して…というような間接的な使い方しかしないのですが、

一応ポイントオブジェクトの本当の使い方は、
点群としてスキャンしたデータや、砂の様に大量に小さな粒を表示するときに、
（インスタンスを大量に置いて表示するよりも）高速に表示するというものです。

アトリビュートとして、点毎にパラメータを持たせることもできます。
例としてランダムな値を与えてみました。



Cycles を使って、アトリビュートに従って着色をして表示しています。
点は球として表示されます。
表示したいのは「大量の球である」と予め分かっているので、その表示に最適化した表示がされ、
球のインスタンスを配置するよりもさらに速く表示がされることになっています。

この程度の表示だと１、２割速くなる程度で「あまり劇的な差はなかったな…」というぐらいですが、
数万以上の大量の点を表示すると差が顕著になってくるようです。

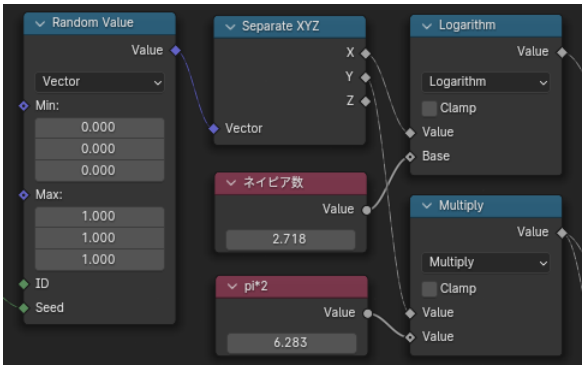
標準正規分布

点をランダムに分布させるときに、Random Value ノードを使って分布をさせると、四角い箱の中に一様分布になります。
標準正規分布などを使って、球形にもやっと広がった分布にさせたい時もあるでしょう。
一様分布から標準正規分布に変換する時には Box-Muller 法という方法が有名です。
数式ノードを使って比較的簡単に実現できるので、実装してみましょう。

Wikipedia の記述を引用すると、
「確率変数 X 及び Y が互いに独立で、ともに（0, 1）上での一様分布に従うものとする。このとき、

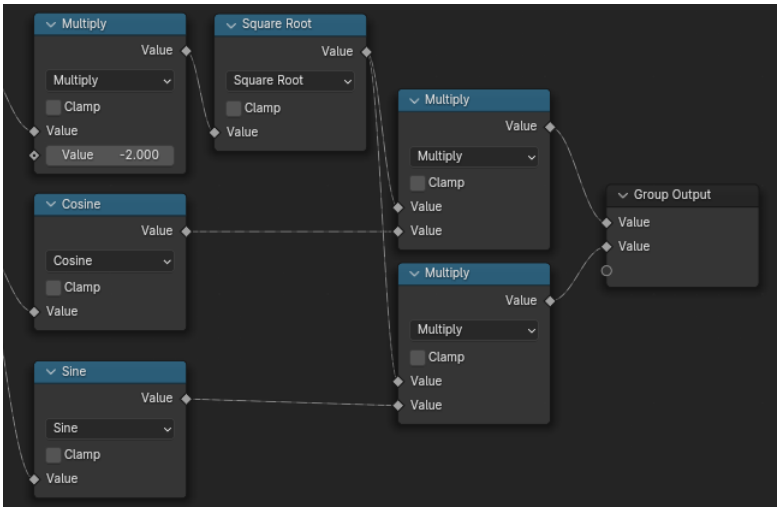
$$Z_1 = \sqrt{-2\log X}\cos 2\pi Y,$$
$$Z_2 = \sqrt{-2\log X}\sin 2\pi Y$$

で定義される Z1, Z2 は、平均 0、分散 1 の標準正規分布N(0,1)に従う互いに独立な確率変数となる。」
となっています。
この式に従って標準積分布を作成してみます。



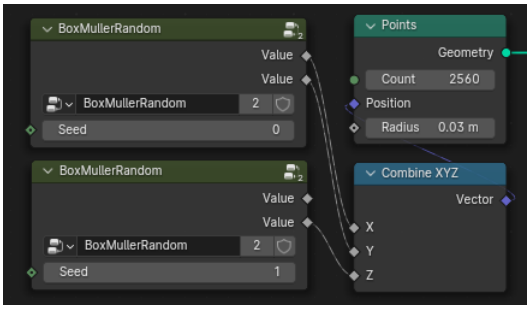
ランダムな分布を(0,0)から(1,1)まで分布するようにして、XYZ を分離します。
Box-Muller 法 は、2次元のXYの分布から２つの正規分布を作る方法です。
そのため、Zが余ってしまいますし、３次元の分布を作るには正規分布が１つ足りません。
3次元の正規分布を作るには、もう一つ Box-Muller 法を使うことになります。
（なので、ノードグループにまとめると便利です）

Logarithm(Log) ノードは、Base(底)もパラメーターを与えるタイプなので、自然対数の場合はネイピア数(2.71828...)を数値で与えます。
同じように、2π の部分なども、数値を与えます。

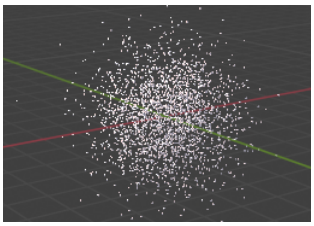


後半です。
前半で定数のファクター部分などを計算したので、
それらを使って数式通りに計算するように、ノードを構成していきます。
最終的に、２つの正規分布を出力します。

この部分までをまとめてノードグループ化します。



ノードグループを2つ使って、XYZ の3成分分のランダム分布を作ります。
この時、ランダムは別々に独立していないといけないので、Seed 値は違ったものを与えてやらないといけません。



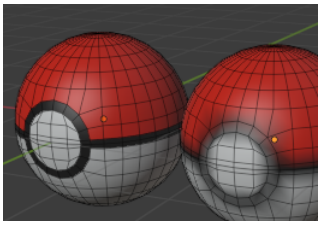
点の位置に球のインスタンスを配置して、このようになりました。
パーリンノイズで代用していた時は、ちょっと怪しいランダム感でしたが、正しく正規分布の丸いランダムになっています。

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

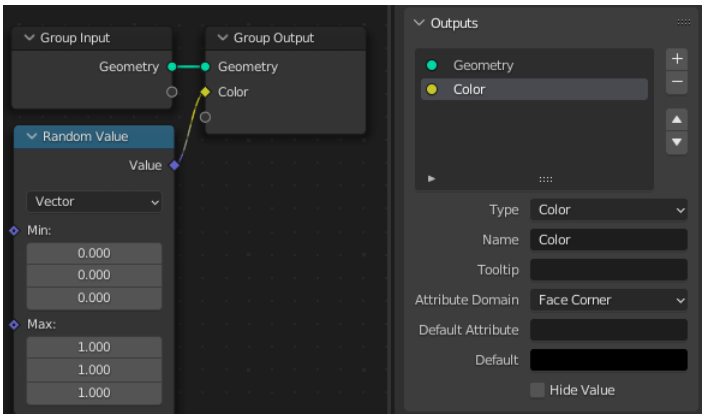
Evaluate on Domain (ドメインでの評価)

Utilities - Field - Evaluate on Domain は、もともと Interpolate Domain(ドメイン補間)という名前だったものが
Blender 3.5 で Evaluate on Domain(ドメインでの評価) と改名されたものです。
メッシュなどで、どの要素がアトリビュートを持っているのか。頂点なのか、面なのか、というのが Domain(ドメイン) 設定です。
普段はあまり意識しないのですが、UV や Color Attriibute (カラー属性)を使う時には意識する必要があります。

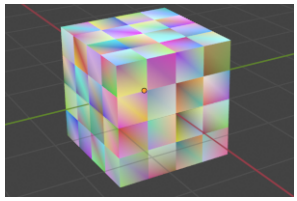
特にカラー属性は、昔の名前（頂点カラー）の印象もあって頂点ごとに値を持っているように思えますが、実はそうではありません。
Face Corner(面コーナー)が値を持っています。



「各頂点ごと」に１色づつしか色が持てなければ、ポリゴンごとのパキッとした色分けができませんね。
赤と黒の境界の頂点には本当は赤と黒両方の色情報が必要です。
さもなければ、せいぜい補間をして右の様な滲んだポケモンボールになってしまいます。
ドメインが頂点であることと、面コーナーであることの違いはこういう時に効いていきます。

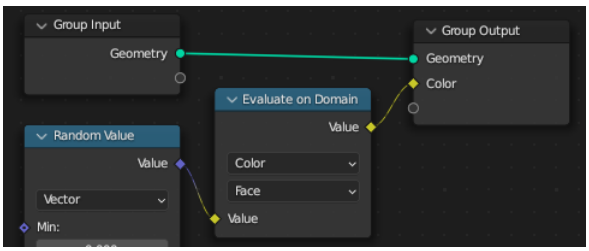


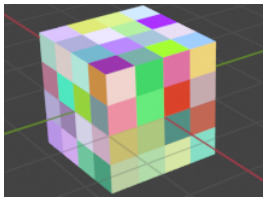
という事で、ドメインの設定に気を使ってきちんと Face Corner を設定して、
ランダムなカラーを出力するようにします。
これで各面ごとにランダムな色を出すことができるか…？と思いきや、



ランダム色は各面の頂点ごとに設定されます。
（デフォルトキューブの各面を4分割した形状です）

各ポリゴン単位で色をランダムにするのに、Utilities - Field - Evaluate on Domain(ドメインでの評価) ノードが使えます。
このノードによって、色情報を持つのが、頂点なのか、面なのか、（その他辺やインスタンスなのか）、といった設定の変更をすることができます。

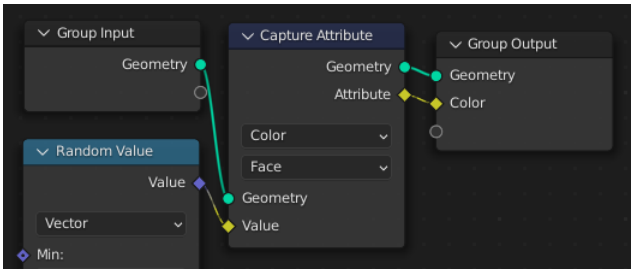




間にノードを挟み込んで設定を Face(面) にすることで、
ドメインが「面」で評価され、ランダム色は各面が持つ情報になります。
（正確には、同じ面に所属する４つの面コーナーが、同じ色の属性を持つようになります）

面単位で色がランダムに表示されるようになりました。

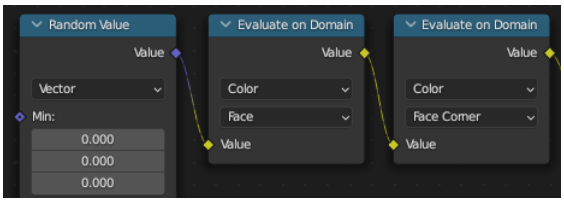
実は、この状態を実現するだけなら今までのノードでも実行する方法はいくつかあります。
その意味では、Evaluate on Domain (ドメインでの評価) は単にシンプルな代用ノードになります。



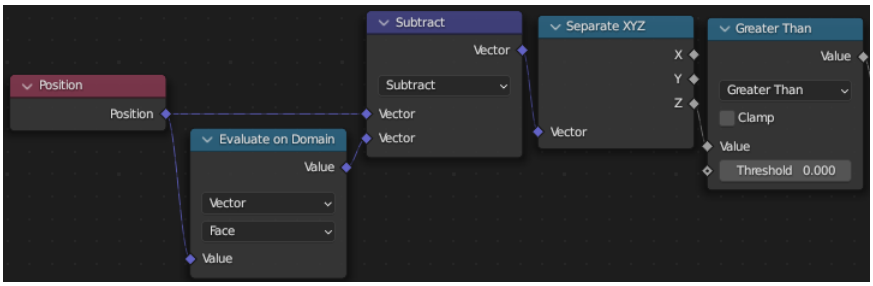
Capture Attribut(属性キャプチャ) を使って「面ごとの属性として色を持つ」と
明示的に指示をすれば、まったく同じことをすることが可能です。

もっと根本的に、Group Output で指定するドメインの設定を Face(面)にすれば、
そもそも出力するときに面単位でしか色を持てなくなるので、
シェーダーでそのアトリビュートを表示すればおなじ見た目が達成できます。

そこで、もう少しだけ Evaluate on Domain を駆使してややこしいノード構成を考えてみましょう。

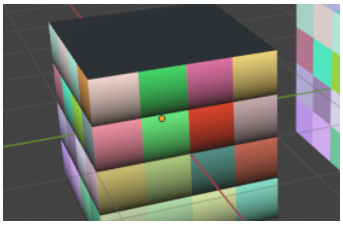


Face 毎にランダムに色情報を持たせた後で、Face Corner が色を持つように戻してみました。
（Faceが1色の情報を持ち、それが Face Corener で4点に分散されるので、
各ポリゴンの4隅が同じ色を持つことになります)
さらに、これを元に色を変化させれば、面ごとにグラデーション付きで色を指定するようなことが可能になります。



少し技巧的ですが「面ごとの Position 情報」を作ります。
これは、ポリゴンの中心位置になります。
各頂点の位置から、ポリゴン中心の位置を引けば、面ごとに相対的な頂点の位置が出てきます。

そこで、その Z 成分を使って、上側は 1、下側は 0 になるようにして、
元の色と掛け合わせてみます。



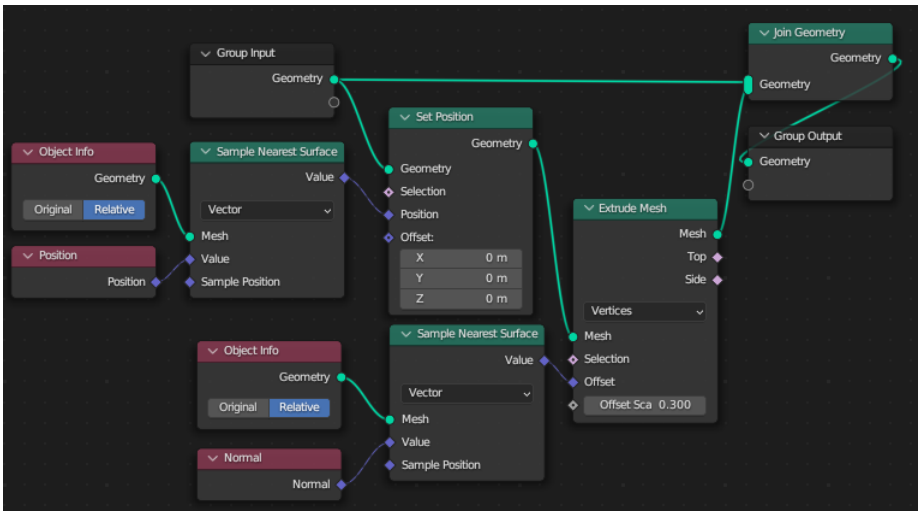
縦方向にグラデーション付きで色を指定することができました。

各面ごとの4頂点で、
「面中心より上であれば、元の色」
「面中心より下であれば、黒」となるようにしたわけです。

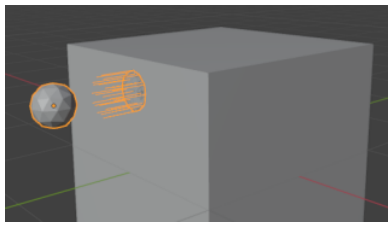
単純な例ですが、03_EVALUATEDOMAIN/03_EvaluateOnDomain.blend として同封しました。

法線の評価

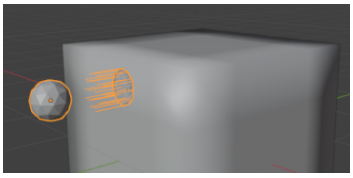
法線情報も、頂点が持つ属性か面が持つ属性かの差が重要なことがあります。
そこで、Sample Nearest Surface を使ってオブジェクトの法線を評価するときに、Evaluate on Domain を使う場合の例を挙げてみます。
法線は直接目に見えないので、法線情報を表示するノードを工夫してみます。



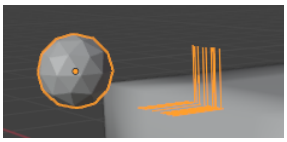
Sample Nearest Surface を使うと
他のオブジェクトの最も近い表面での情報を得ることが出来ます。
そこで、球の頂点を立方体の最も近い表面に張り付け、同時に法線の情報も得て、
Extrude Mesh で頂点からエッジを伸ばし、法線方向を目で見えるようにします。



少しややこしいですが、左のようなノード組みになります。

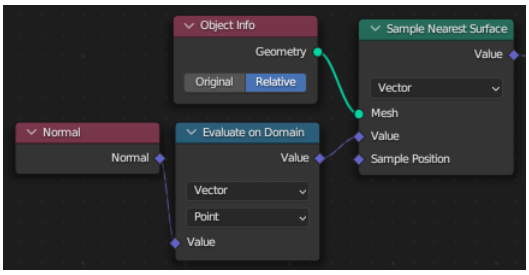


この法線の向きは、ジオメトリーから計算される法線なので、
スムーズシェードなどで見かけを滑らかに処理した場合でも、
法線の向きは面に垂直になります。

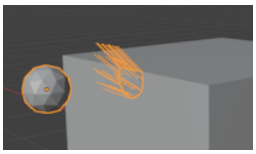


そのため、最近点が角に当たるような場合だと、線が横を向いたり上を向いたりで角のどちらかの法線情報が得られます。
しかも、どちらの面が採用されるかは不定で（数値計算の誤差などに依存して）
球を動かすと線が横を向いたり上を向いたりして表示がちかちかします。

Sample Nearest Surface で、表面の法線情報を使うような処理をするときには、少し困ります。
こういった場合に Evaluate on Domain を使って点としての法線を評価すると、角が滑らかな連続的な評価ができます。



法線を評価する時に、Evaluate on Domain ノードを挟みます。
これによって、キューブの角での頂点の持つ法線（斜め45度を向いています）を
位置に応じて補間された法線情報が Sample Nearest Surface で評価されます。



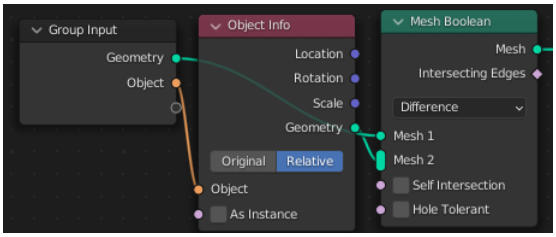
角で丁度45度になり、スムーズシェードなどで見かけ上滑らかにした法線に相当する法線情報が得られました。
法線の評価が連続的になっているので、角の所で球を動かしても法線の向きがチラついたりはしません。

このサンプルは 03_EVALUATEONDOMAIN/03_SampleNormal.blend として同封をしました。

また、他にもトリッキーな理由で Evaluate On Domain を利用する別の例を後半の作例の中に収録しました。

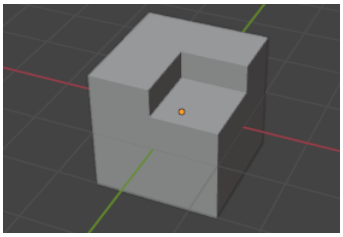
Intersecting Edge(交差する辺)

Mesh - Operations - Mesh Boolean (メッシュブーリアン)の機能に、Intersecting Edges(交差する辺) が追加をされました。
この機能を使うと、ブーリアンを実行した際に、交差部分として作成されたエッジがどの部分かを知ることができます。



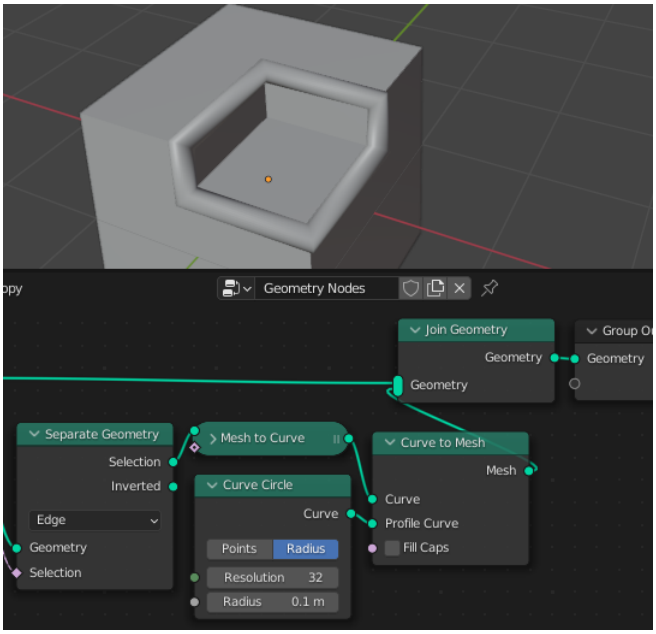
簡単な例として、デフォルトキューブを２つ配置して、片方をブーリアン処理してみます。
(Cube と Cube.001)

Difference(差分) を使うと、片方のキューブで片方のキューブを削ったような形になります。



ブーリアンを使ったときのお約束ですが、片方は見えないように隠しておかないと、変化後の形状が隠れてしまってよくわかりません。

そのあたりをきちんと設定して形が見えるようにします。
キューブでキューブを削り取った形が見えてきました。



この時、Intersecting Edges のソケットから、
「どのエッジがブーリアン処理で削られた部分のエッジになのか」の情報が得られます。

情報が得られただけでは見た目で分からないので、形状で見えるようにします。
Separate Geometry(ジオメトリ分離)を利用してそこだけエッジを抜き出します。
Mesh To Curve(メッシュからカーブへ) と Curve to Mesh(カーブからメッシュへ)を使い、
太さを与えてエッジを強調して表示してみました。

削られた部分のエッジだけが抜き出されていることが分かります。
Difference(差分)だけではなく、Union(合成)やIntersect(交差)などでも同様に使うことができます。

これを元に加工をすれば、２つのオブジェクトを溶接したようなパーツなどを簡単に作れるのですが…
今、交差部分に作った形状は、カーブを元に作った円筒形なので、変形などには少し使いずらい形です。
ここから、Remesh(リメッシュ)モディファイアなどを使えば、等間隔のメッシュに変換ができるので加工などが簡単になるのですが、
このままの状態だと元のCubeの形状と、溶接用のパーツが2つで1オブジェクトなので、少し扱いにくいです。
そこで、２つのキューブとは別に溶接部になるオブジェクトを用意して、それが Cube と Cube.001 の交差部分のパーツになるように改造してみます。