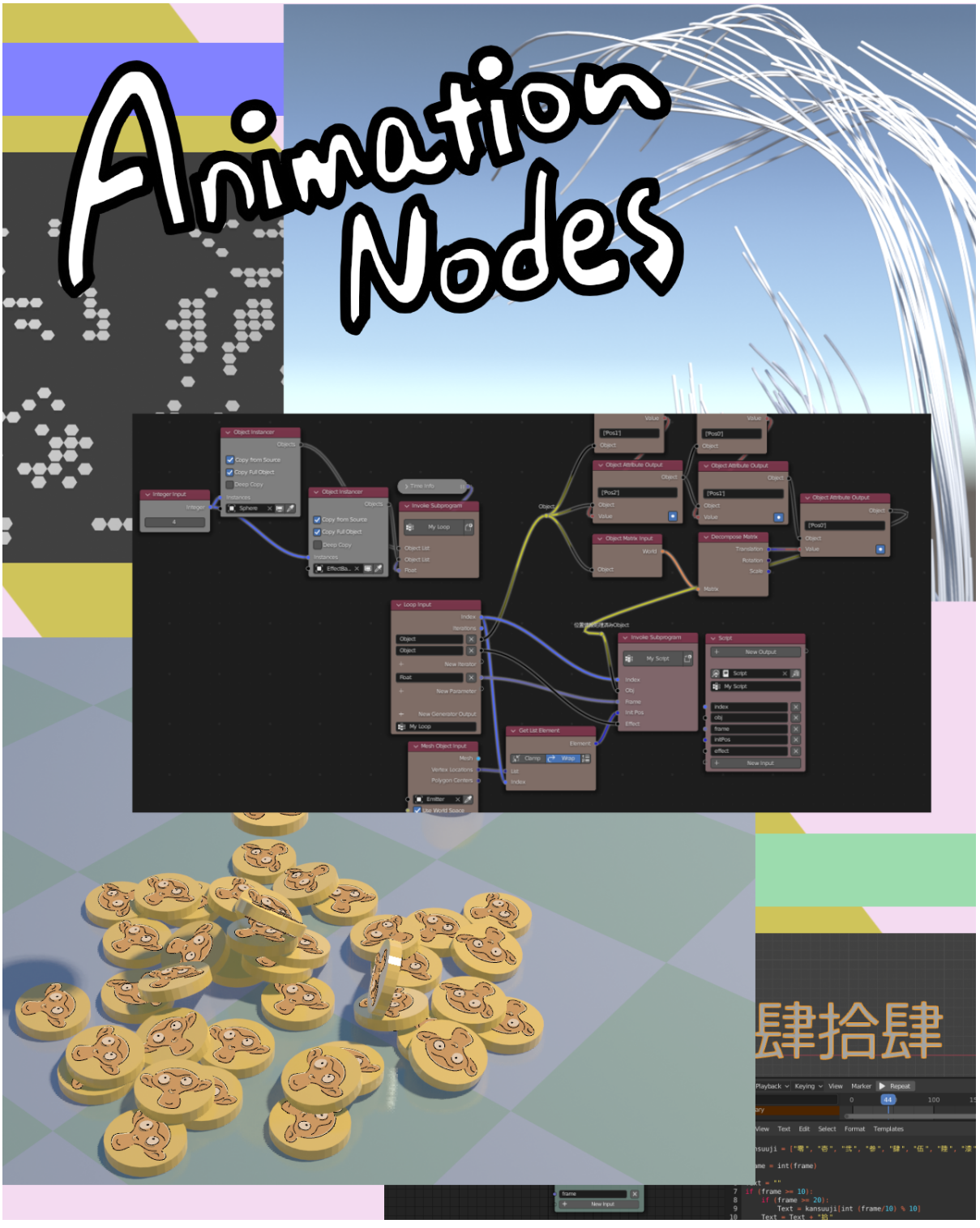
アニメーションノードを使ってみよう

Animation Nodes Addon 入門 (Blender 3.3 and Animation Node 2.3)

Version 2.0

Q@スタジオぽぷり
@popqjp

作者 Twitter アカウント
Q@スタジオぽぷり



目次

初めに … 3
インストール … 3
アニメーションノードの基本 … 5
新規ノードツリーの作成 … 5
インプットとアウトプット … 5
連動して動くオブジェクト … 7
ノードの配色 … 8
オブジェクトの複製 … 8
インスタンスを並べる1 … 9
インスタンスを並べる2 … 10
エクスプレッションとスクリプト … 11
エクスプレッション … 11
スクリプト … 12
ループ処理 … 14
リストについて … 17
マテリアルのリスト … 18
ノードのキーフレームアニメーション … 19
メッシュの操作 … 20
メッシュデータの基礎 … 20
メッシュデータの変更、配置 … 21
Falloff … 24
もくもく変形 … 25
メッシュの配置とオブジェクトによる制御 … 27
モディファイアとの併用 … 28
ループを使ったメッシュデータの変更 … 30
任意のアトリビュート編集 … 32
ジオメトリノードの係数 … 32
カーブ(Spline)の操作 … 34
カーブからメッシュを作成する … 35
カーブに沿って動かす … 36
カーブに沿って動かす（プロジェクション） … 37
テキスト情報 … 38
アニメーション情報（F-Curves） … 39
アニメーション情報（アクション） … 41
時間差アクション … 43
頂点ウェイトのアニメーション … 44
頂点ウェイトの時間進化 … 45
ライフゲーム … 48
UV、頂点カラー(面コーナーにあるデータ) … 50
グリースペンシルの操作 … 52
レイヤー操作 … 52
フレーム操作 … 53
オブジェクトで GP アニメーションのコントロール … 54
グリースペンシルからカーブを作成して袋文字 … 57
パーティクル情報の取得 … 58
着弾エフェクトの作成 … 59
グリースペンシルを使った着弾エフェクト … 60
グリースペンシルでの煙 … 62
パーティクルヘア情報の取得 … 63
ヘアを元にしたスプラインを伸ばす … 63
補間(Interpolation) … 64
マテリアルノードの制御 … 66
Geometry … 67
カメラ上の座標 … 67
KD and BVH Tree … 68
一番近いオブジェクトを探す … 68
メッシュの内側外側を判定する … 68
Sound 情報を利用する … 70
剛体シミュレーションを制御する … 71
初速度の制御 … 72
剛体シミュレーションの結果のベイク … 73
剛体オブジェクトの位置情報 … 74
複製と親子関係 … 75
剛体シミュレーションとエフェクト … 75
グループ化 … 78
Import と Export …80
Importでスクリプトからデータを渡す … 80
リストを渡してグラフにしてみる … 81
Exportでスクリプトにデータを渡す … 82
Expression ノードと Python のワンライナー …83
サンプル.blend ファイル … 85
終わりに … 86

アニメーションノードを使ってみよう

Animation Nodes Addon 入門 (Blender 3.3 and Animation Node 2.3)

作者 Twitter アカウント
Q@スタジオぽぷり

2020.4.25 ver 1.0
2020.5.08 ver 1.2
2020.5.31 ver 1.4
2022.10.03 ver 2.0

初めに

アニメーションノードは、Blender の有名なアドオン(Add-on)です。
ノードベースで複雑なアニメーションを制御する機能があります。
メッシュの編集に関しても幾つか機能があるので、アニメーションに関係なくある種のモデリングにも使える部分があります。

この本では Animation Node を用いて様々な制御をおこなう手順を紹介します。
題名は入門と大きく出ましたが、内容としては blender の初心者でも扱えるように一から手順を解説していく、というほど丁寧な解説ではありません。
ある程度 blender の操作に慣れ、コンポジットノードや、マテリアルノードを使ったことがあり、ノードによる処理がどういうものかわかっている人を対象に考えています。
この本の ver 1.4 を書いた時には、使った Blender のバージョンは 2.82, Animation Nodes のバージョンは 2.1.7 でした。

アニメーションノードは長らく、ノードを使って形状やアニメーションの制御をする代表的なアドオンとして有名ですが、
Blender 2.92で、**ジオメトリノード**が導入されたことで、メッシュの形状やインスタンスの配置などをノードを使って制御することができるようにました。
アニメーションノードが担っていたような機能の一部は、ジオメトリノードを使って実現することができます。

しかし、ジオメトリノードよりも細かい操作が行えたり、ジオメトリノードの持たない機能をまだ数多く持っていて、アニメーションノードを利用するメリットは依然として健在です。
（むしろジオメトリノードの方が得意そうな分野はジオメトリノードに任せて、アニメーションノードの特異な操作だけを行うような分業ができるので、より使いやすくなったという面があるとも言えそうです。）

ver 1.4 のあと、だいぶ時間が経って情報が古くなってきた部分もあり、Blender 3.3 を用いた新しい版として ver 2.0 にしました。
説明が古くなった部分の修正の他、幾つかのサブセクションや作成を追加し、スナップショット画像なども、Blender 3.3 での画像に更新しています。
その他、説明が足りなかった部分に説明を追加したり、説明はあってもサンプルファイルが無かった作例などに関してもサンプルファイルを大幅に追加をしました。

画面のスナップショットは、最初の方の一部は日本語表記で撮っていますが、その後のほとんどの画像は、英語の状態で撮っています。
※アニメーションノードは公式アドオンではないので、日本語翻訳は一部しかされておらず、多くは英語のままなので、英語環境の方が使いやすいかと思います。

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

インストール

Animation Nodes の公式ページは以下のサイトです。
https://animation-nodes.com

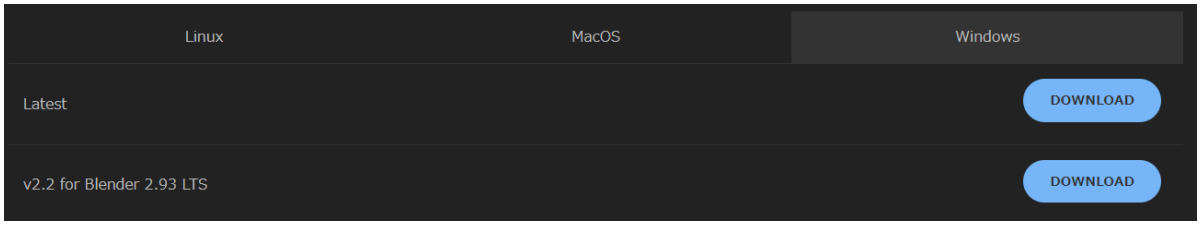
最新の Aniation Nodes をダウンロードして、blender の Add-on インストール手順に従ってインストールと有効化を行います。
公式ガイドの場所も、ver1.4で書いた場所はリンク切れになっており、現在は以下のようになっています。

公式ガイド（英語）のURL:
https://docs.animation-nodes.com/
※公式サイトから、"GET STARTED" のボタンでリンクがつながっています。

ガイドのインストール手順の項目は以下のサイトです。
https://docs.animation-nodes.com/documentation/installation/

また、インストールのトラブルシューティングとして、以下のページが公式からリンクでつながっています。
https://github.com/JacquesLucke/animation_nodes/issues/1240

ダウンロードできるアニメーションノードのバージョンは、安定版と開発版です。

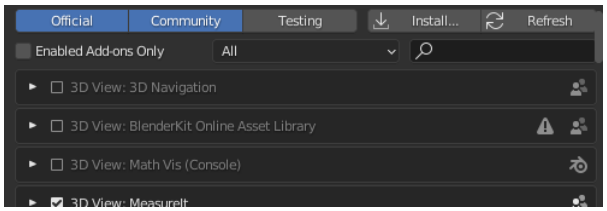


ダウンロード画面から明らかなように、安定版は Blender 2.93 LTS に対応した版になっています。
Blender 2.93 LTS と現時点で最新の Blender 3.3 では内部で利用している Python のバージョンが異なり、安定版は Blender 3.3 では動きません。

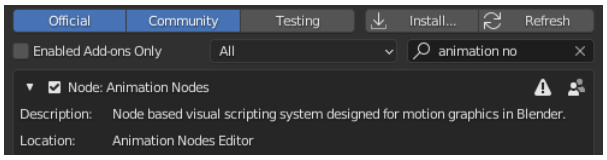
本書では、Blender 3.3 との組み合わせでアニメーションノードを使うとして、開発版の方を使用していきます。

アニメーションノードの旧バージョンが入っている場合はアンインストールします。

アンインストール時の注意点
「アニメーションノードを無効にした状態で起動した Blender 」の状態でないと、アンインストールはできません。
（一旦アドオンが有効な状態になると、アドオンの一部のファイルを Blender が掴んだ状態になるので、ファイルの削除が Blender 自身からは行えないことが関係しています）



Preference (プリファレンス) 画面のインストールボタンでファイルを選びイントールします。
この Add-on は、.zip で固まったファイル群をまとめてインストールするタイプの Add-on になります。
（解凍の必要はありません）



Add-on を有効にすれば、使用する準備は整います。

インストールのトラブルシューティングのページに記載されていますが、
Windows 版は vc_redist.x64が必要です。
他にもこの環境が必要なソフトは多いので、既に導入済みである可能性も高いのですが、もし runtimes 関係のエラーが出るようなら導入します。

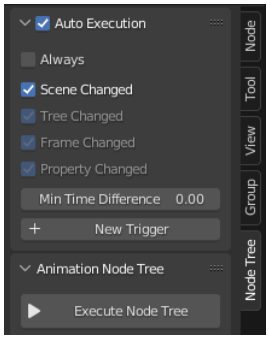
アニメーションノードの基本

まずは実際に簡単なアニメーションノードの使用例を見てみましょう。

新規ノードツリーの作成

ノードエディターの種類に Animation Nodes が追加されています。

Animation Nodes 画面にして新規ノードツリーを作成します

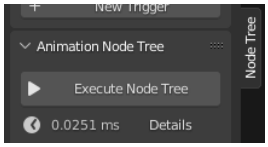


アニメーションノードを組むと、
ある条件が整ったときにノードで組んだ操作を実行します。

実際にノードを組む前に、まず設定を確認してみます。
画面右側のパネル(N) にノードツリーの設定の項目があります。

ここの Auto Execution(自動実行)の項目がデフォルトの Scene Changed の場合は、シーンに何らかの更新が入ったときに計算が実行されるモードになっています。

以前はAlways (常時)がデフォルトになっていました。
文字通り、なにも操作していない待機状態の時でも常に繰り返しアニメーションノードの計算を実行を繰り返すモードです。
情報の更新が確実なのですが、いささか計算の無駄が多い設定なので、このモードはデフォルトでは off にするようになったようです。
問題が無ければ、「Always(常時)」にはせずにデフォルトにしておくのが良いと思います。
(その他、Tree, Frame, Property で、それぞれが変更された時にのみ計算が実行するように設定もできます)

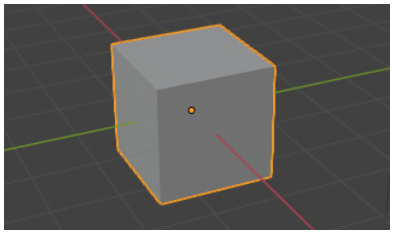


場合によっては、実行を一度だけ行えばよいことがあります。
一度計算すれば良く、ほとんど再計算をする必要がないような場合です。

そうした場合には自動実行を切り、必要に応じて Execute Node Tree ボタンで手動で実行を行うのが効率の良い方法になります。

インプットとアウトプット

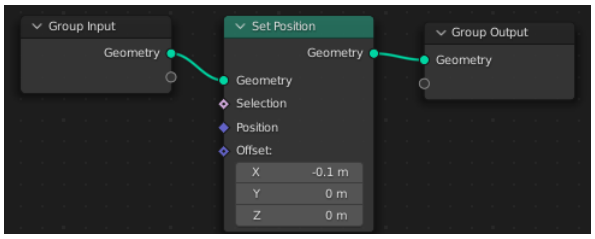
最初に簡単な例として、デフォルトキューブをアニメーションノードを使って操作してみます。
ところで、ノードによる操作にはジオメトリノードが既にあるので、**ジオメトリノードとの違い**を気にしながらアニメーションノードの操作を見てみましょう。



アニメーションノードを使って、
デフォルトキューブを動かすという操作をしてみます。

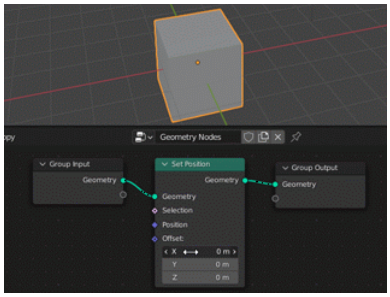
ジオメトリノードの場合

標準機能にジオメトリノードのある今、アニメーションノードを使ってみようという人であれば、
ジオメトリノードは軽くは弄ったことがあると思います…ありますよね？
まずはジオメトリノードとの比較で考えてみます。



基本の操作…モディファイアにジオメトリノードを追加して、
新規のノードを作成する…あたりは説明は省略します。
即ノードを組んでみます。

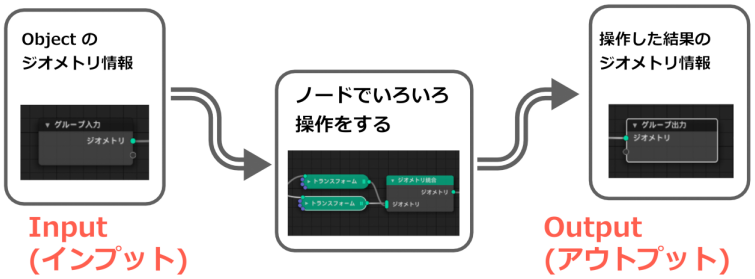
Group Input から、Set Position(位置設定) で位置を変更するノードにつなぎ、Group Output へとつなぎます。



Offset のソケットの値を変化させると、すべての頂点の位置がそれに応じて変化して、
結果的にオブジェクトが動いたように見えます。
動画)SetPosition01.gif

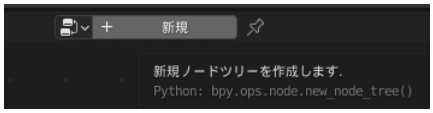
オブジェクト自体が動いているように見えますが、よく見ればオブジェクトの原点の位置は動いていません。
「オブジェクトの位置はそのままで」メッシュが変形していることが分かります。
これは、Set Position の代わりに Transform ノードを使った場合でも同様です。

インプットで元のオブジェクトのジオメトリ情報が入ってきて、ノードの中でいろいろな操作をして、その結果をアウトプットする、というイメージになります。

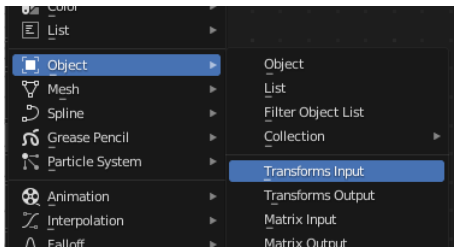


この、「ノードでいろいろ操作する」となっている部分を中心に考えるので、
「元のオブジェクトの情報」が、Input(入力)
「色々操作した結果」が Output(出力)
になるのですね。
アニメーションノードでも、この基本概念に沿って考えることになります。

アニメーションノードの場合

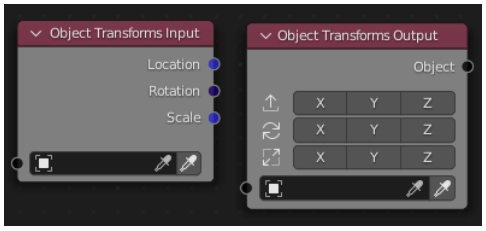


Animation Nodes の編集画面で新規ノードツリーを作成します



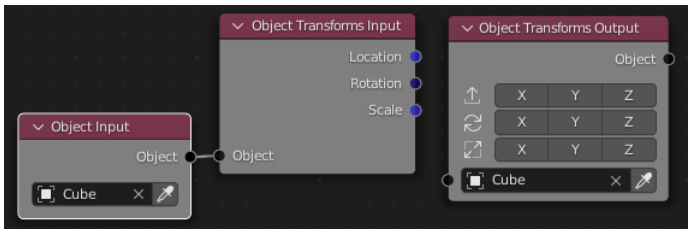
Addメニュー(Shift-A のショートカット)からノードを追加できます。
オブジェクトのカテゴリーから、Transform Input と Transform Output を作成します。

基本は、Input が元オブジェクトの情報を得るためのノードで、
Output が操作をする（操作した結果を出力する）ためのノードです。

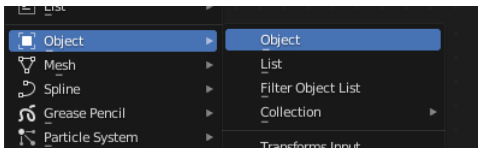


Object Transform Input のノードには、対象のオブジェクトの情報を得るソケットがありますし、
Object Transform Output には、対称のオブジェクトの位置、回転、拡大縮小を操作する項目があることが分かります。

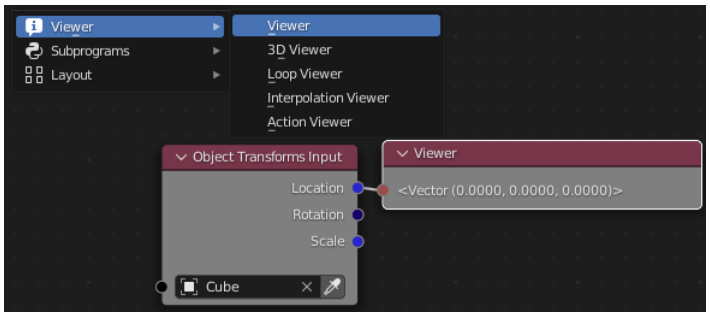
ジオメトリノードと違って、操作対象になるオブジェクトが決まっている分けではないので、
オブジェクトを決めるソケットもついています。
（最初は空です）



オブジェクトを決めるソケットに対して、
オブジェクトを指定するノード(Object - Input)からつなぐか、
ソケットで直接オブジェクトを選択して、対象のオブジェクトを指定します。

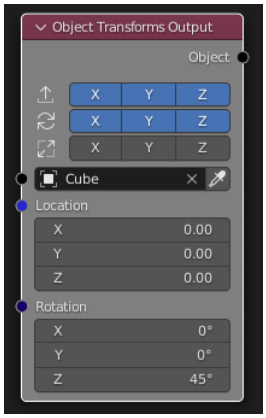


オブジェクトを指定する Object Input のノードは、
やはりオブジェクトのカテゴリーの中の Object で作成します。
（※メニューの表記の名前とノードの名前が一致していない場合があります）

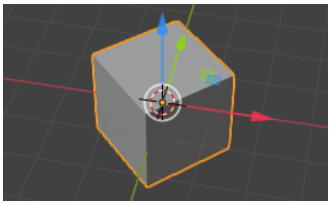


Transform Input の位置のソケットにビューワーを繋ぐと情報を見ることができます。
追加メニューの Viwer のカテゴリーに入っています。
デフォルトキューブの位置情報はVector(0,0,0)で、キューブが原点にいることが分かります。

情報を気軽に見ることができるビューワーノード、ジオメトリノードにも欲しいですね。
現時点(3.3)では、アニメーションノードにはあるけれども、
ジオメトリノードにはない便利な機能はまだ多数存在しています



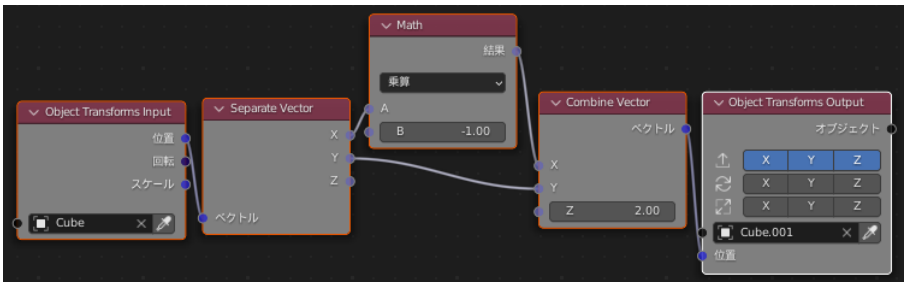
Object Transform Output ノードで位置や回転を操作する入力ソケットは、
最初は無効化されて、隠れています。
アイコンで操作を有効にすると現れるような仕様になっています。



例えば、Rotation の Z を45度に設定をすると、
Z軸回りに 45 度回転した状態になります。

連動して動くオブジェクト

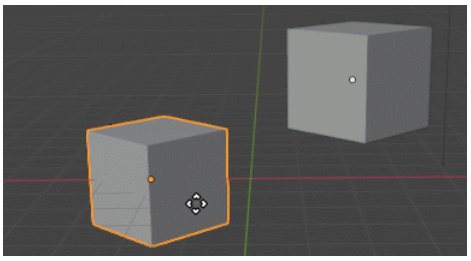
Input と Output をつないで、オブジェクトの位置情報を使って別のオブジェクトを動かすということをしてみます。
Cube を複製して、Cube と Cube.001 の２オブジェクトを用意します。
下の例のように ベクトル の分離(Separate)や結合(Combine)、数式ノードを組み立ててみます。



Separate Vector(分離)や Combine Vector などのベクトル成分のノードは、
Vector(ベクトル) のカテゴリーから選択します。
同様にMath(数式) のノードはNumber(数)のカテゴリーから Math を選択します。

基本的な計算ノードなのですが、現時点でジオメトリノードなどとメニューのカテゴリー分けのルールなどが違うので、少し混乱します。
（歴史的な経緯が違うものなので、やむを得ないのですが、本当は統一化してほしいですね）

ノードの構成の流れは以下のようになっています。
1.Cube の位置の情報をInputノードで取得して、Separate Vector で成分をxyz成分に分離。
2.x座標を逆にして、 Combine Vector で再度ベクトルに戻す。
3.Output ノードで Cube.001 を動かします。
（位置が完全に重なってしまうと良く分からないので、z座標をずらしています。）



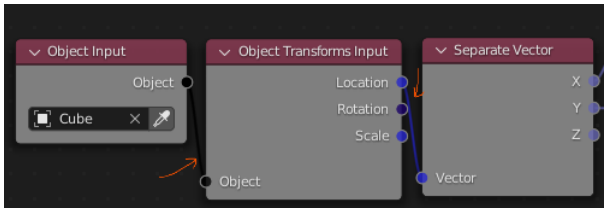
Cube を操作すると、連動して Cube.001 が動くようになりました。
動画)Comoving01.gif

オブジェクトの重心も表示されているので、位置の確認ができます。
ジオメトリノードでのキューブの移動との大きな違いは、
「オブジェクトの位置を」動かしているということです。

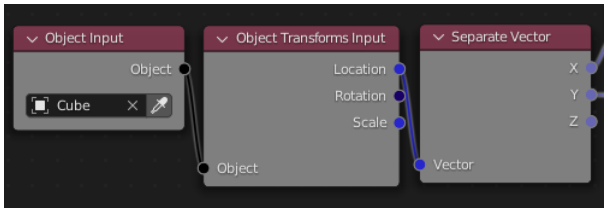
アニメーションノードはオブジェクトそのものを動かすというのが、ジオメトリノードとの大きな違いになります。
ジオメトリノードでも「インスタンスを配置する」と考えれば、変形ではなく実質的に「オブジェクトを動かしたような」効果を得ることができます。
逆に、アニメーションノードでも、メッシュを変形することは可能です。
ですから、この差が絶対のルールというわけではないのですが、
この辺の基本的な操作の違いが、ジオメトリノードとアニメーションノードを使い分ける場合には重要な差になってきます。

ノードの配色

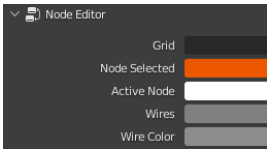
ところで、今までに作ったスナップショット画像は、必要のないノードも選択された状態になったスナップショットでした。
（ノードの枠が赤くなったりしています）
Blender のバージョンが進んで行く間に、デフォルトカラーが微調整されたり、各部品の色付けルールが調整されたのですが、
その結果どうもアニメーションノードのワイヤーの色設定が、（デフォルトの暗い Blender Dark のテーマだと）見づらくなってしまったようです。



(Animation Nodes の開発版の配色なので、この先調整が入るかもしれませんが)
スナップショット画面に使うには厳しいので、この後の画像は
テーマのワイヤー色の設定を明るめにしたものを使うことにします。



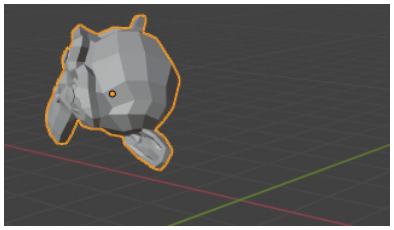
Wires の項目の色を明るさ0.5の灰色まで上げて、線の輪郭が明るく見えるようにしました。



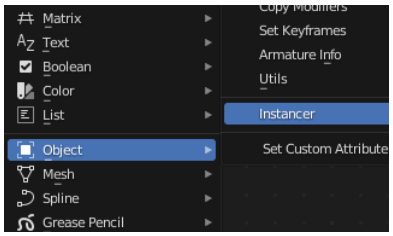
オブジェクトの複製

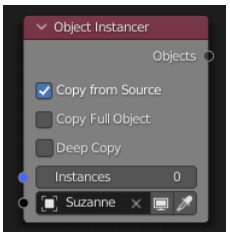
既にあるオブジェクトを動かすだけではなく、アニメーションノードを使って新たにオブジェクトを作成して、それを管理することができます。
オブジェクトを作成するノードは Instancer(インスタンサー) と呼びます。

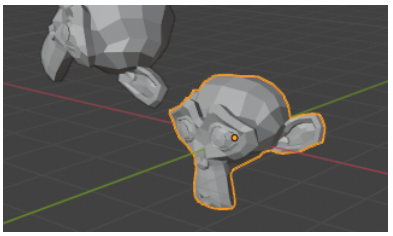
まずは、オブジェクトの複製をしてみます。



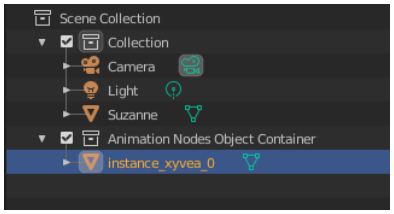
まず、複製の元になるオブジェクトを普通に作成しておきます。
コピーと重なってしまうと分かりづらいので、スザンヌを、あらぬ場所、あらぬ方向に置きました。



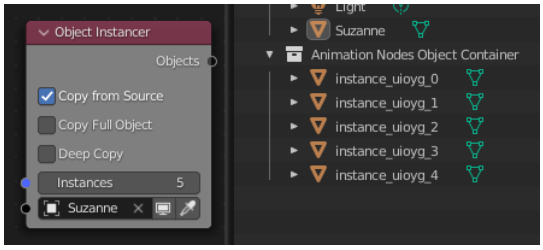




追加メニューから、Object - Instancer を選択します。
オブジェクトにスザンヌを選択して、Instance 数を 0 から1 にします。
原点にスザンヌのコピーが作成されました。



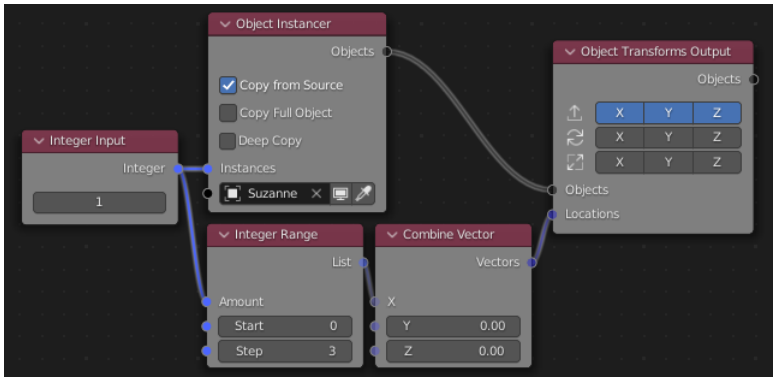
アウトライナーで確認をすると、
Animation Nodes Object Container という名前のコレクションの中に、
作成されたコピーが入っているのが分かります。
名前は内部でタグとして使用するためのランダムな（？）文字列になるようです。



インスタンスの数を増やせば、複数のスザンヌがコピーされることが分かります。
しかし、このままだとすべて原点で重なってしまってあまり意味がありません。
作ったコピーを整列させてみます。

インスタンスを並べる 1

複数されたスザンヌを並べてみます。
Object Instancer から出力される Objects データの位置を Object Transforms Output で変更することになります。



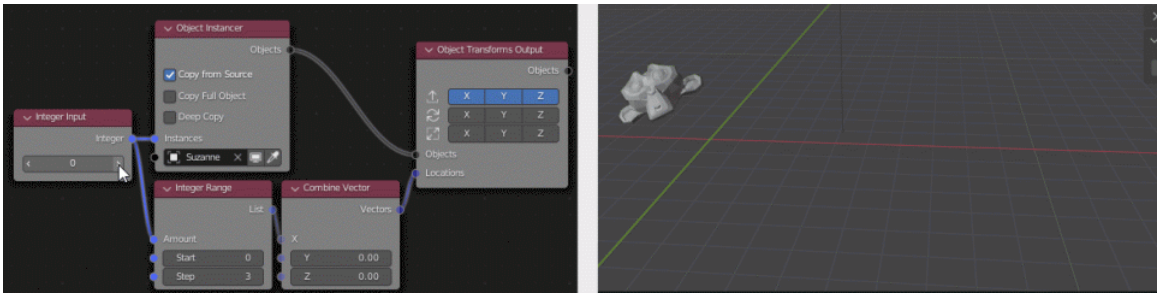
追加メニューから Number(数) - Integer、Number - Integer Range と Vector - Combine を使って上のようにノードを組み立てます。

上の段がスザンヌのコピー作成、下の段が並べるための位置ベクトル情報の作成です。
Integer Input ノードを使って共通の整数を指定することで、コピーの数とベクトルの数が一緒になるようにしています。

Integer Rande ノードは、Amount の数だけの整数のリストを作ります。
リストの概念は、Geometry Nodes には無い、Animation Nodes 特有のものです。
名前の通り複数の情報を並べたもので、この場合は 0, 3, 6, ... という数字のリストが作られることになります。
詳しくは後で確認することにして、いまはそういうものとして話を進めていきましょう。

数字のリストを Vector - Combine ノードを通すことで、ベクトルのリスト(0,0,0), (3,0,0), (6,0,0), …になります。

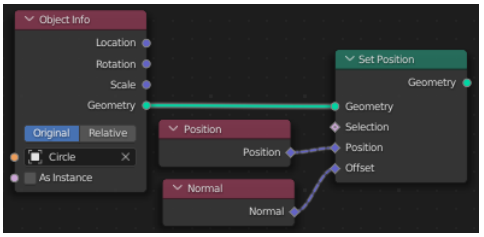
最後に Object Transform Output に、コピーされたオブジェクトのリストと、位置ベクトルのリストを繋ぎます。



Integer Inputの数を増やすと等間隔にスザンヌが並びます。
動画)IntancerB01.gif

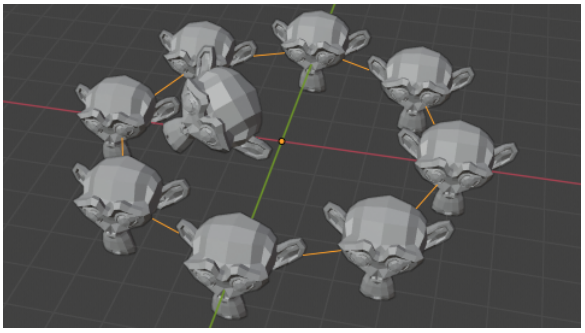
インスタンスを並べる 2

別のオブジェクトのメッシュ情報を利用して、スザンヌをメッシュで作った円の頂点に配置して、整列してみます。

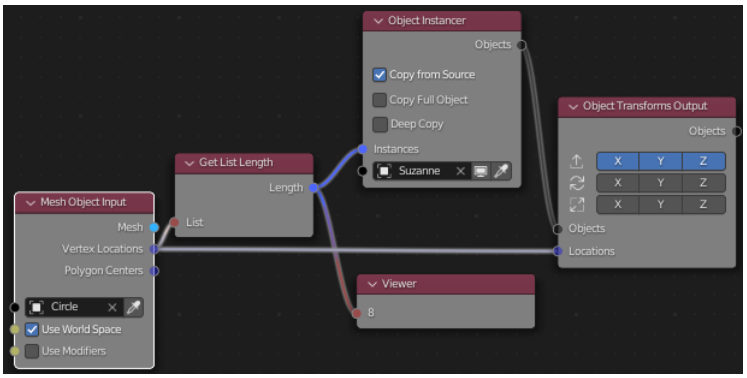


ジオメトリノードでは、Object Info ノードを使って任意の別のオブジェクトのジオメトリ情報を取得できました。
その後、そのジオメトリのPosition(位置)やNormal(法線)等のノードにつなげば、
それらの情報を取り出すことができます。

そうしたジオメトリ情報を得る方法のアニメーションノード版になります。



例として、8点で作った円(８角形)の(Circle)を作成し、
そのメッシュの情報を取得して、
頂点の位置にスザンヌを並べてみます。

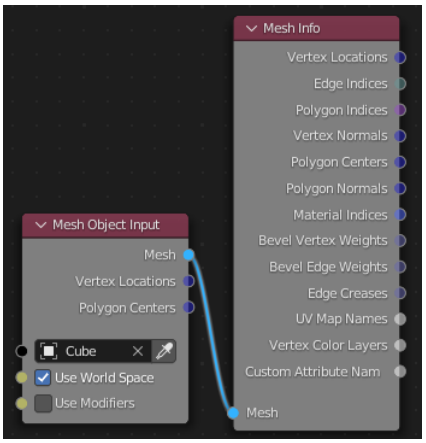


ノードの全体配置図です。
Mesh - Mesh Input で Mesh Object Input ノードが配置できます。
これは、オブジェクトのメッシュ情報を得るノードです。
オブジェクトに Circle を選びます。

頂点位置（のリスト）は Vertex Locations のソケットから得られます。

⚠
メニューでの表記(Mesh Input)とノード名(Mesh Object Input)が微妙に違うところに気を付けます。

List - Get Length につなぐと、リストの長さ(頂点数)が得られます。
Viewerをつなげば頂点数が8だと確認ができます。
頂点の数だけインスタンサーで複製をして、頂点位置のリストを Transform Output につなげば、8角形の頂点の位置にスザンヌが配置できるわけです。



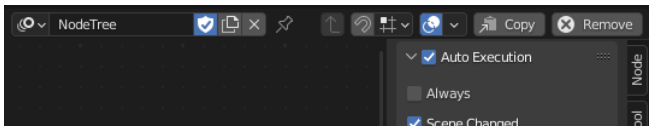
水色のソケットは、メッシュ情報全体を意味しています。
水色のソケットから Mesh - Mesn Info ノードにつなぐと、頂点(Vertices)の他にも様々な情報が得られます。
例えば法線のリストや、その他メッシュのエッジやポリゴンが結んでいる頂点の番号リストなどです。

（最もよく使われる、Vertex Locations(頂点の位置）と Polygon Cenetrs(ポリゴンの中心位置)のリストだけは、
Mesh Info を使わなくても得られるように Input ノードにも
（デフォルト状態で）ソケットがあるわけですね。

アニメーションノードの削除

ところで、要らなくなったノード全体のデータはどうやって消すのでしょう。
ジオメトリノードなどは、オブジェクトやモディファイアを削除すると、「誰からも利用されていない状態になれば、保存の時にデータが消えるのでユーザーは特に気にしなくて良い」というルールで、削除をあまり気にしなくて済んでいます。

アニメーションノードはどこかのオブジェクトやモディファイアに結び付いているわけではなく、シーンに結び付いています。
普通はシーンを消すわけには行かないので、アニメーションノードが要らなくなっても残ります。
（×ボタンでシーンからの利用を解除という操作もできません）



そういう訳で、専用の削除ボタンがヘッダーに追加されいて、このボタンでノード全体を削除することになります。

このほか、ノードの複製をつくる Copy ボタンなども用意されています。
（コピーというより、むしろDuplicate（複製）の動作です）

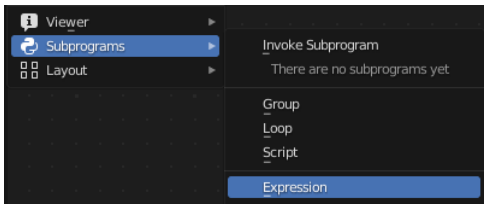
エクスプレッションとスクリプト

インスタンスを任意の位置に配置する…ような操作の場合は、ジオメトリノードが充実してきたために、アニメーションノードを使う必要は薄れてきました。
そんな中で、アニメーションノードが優位を持つ大きな特徴は、アニメーションノードの中で、スクリプトを走らせることができることです。
スクリプトというと、普通は基礎を学んだあとで最後の応用として学ぶ、というような印象がありますが…

簡単な計算をさせたい、というときにわざわざ足し算や掛け算のノードを幾つも組み合わせて計算させるよりも、数式をパパっと入力できれば良いのにということが良くあります。
アニメーションノードは、そうしたちょっとした数式をスクリプトで書いてノードの中に組み込む、というようなことが簡単にできるようになっています。
折角ですので早いうちに使えるようになった方が楽ができると思います。
一足飛びに、スクリプト周りの使い方を説明してみます。

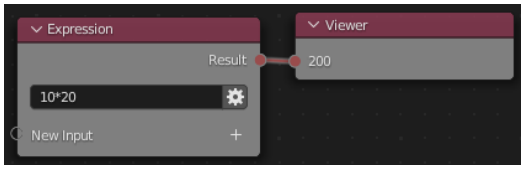
エクスプレッション

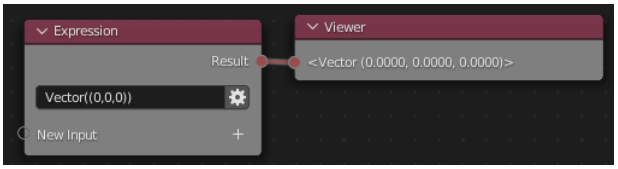
一番簡単にスクリプトを実行できるノードは、エクスプレッション(Expression)です。



Subprograms - Expression というメニューになっています。

エクスプレッションノードには、1行分の入力欄があります。
一行に収まる範囲で様々な計算をすることができます。（Pythonのコマンドとして実行されます）
Resultのソケットにビューワーノードを繋げば、その場ですぐに実行結果を見ることができます。

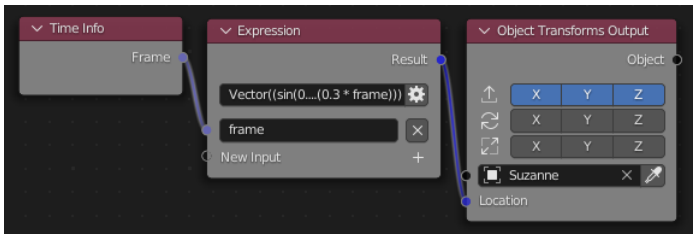




エクスプレッションの出力は、数値だけに限りません。
ベクトルなどもエクスプレッションで計算して出力させることができます。

⚠blender で python を使っていればおなじみですが、Vector の表記には括弧が 2 つ必要なのに注意してください。

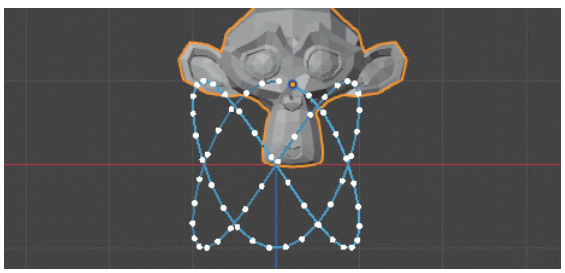
Expression の Input のソケットには様々なデータをつなぐことができます。

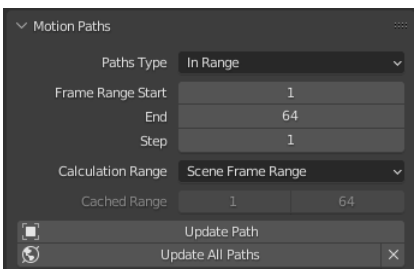


例えば、Animation - Time Info から時間情報をつなぐことができます。
入力した情報には名前を付けてやらなければいけません。
デフォルトでは変数名 "x" の名前になるのですが、
任意の名前に変更できるので、この場合は入力が時間なので frame に変更しました。

これで、時間情報は変数 "frame" として利用ができます。
例えば三角関数とVectorを使って、オブジェクトの位置情報につなげば、数式でアニメーションさせることができます。

```
Vector((sin(0.2 * frame), 0, cos(0.3 * frame)))
```



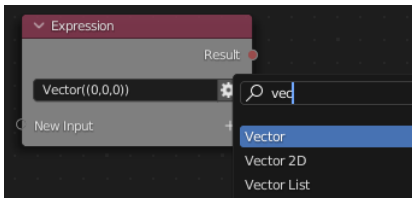


数式で動くスザンヌの軌跡です。
動画)Comoving01.gif

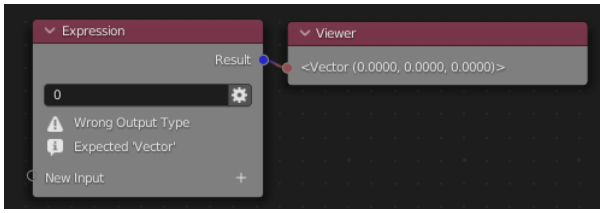
(軌跡の表示にはモーションパス(Motion Paths)の表示機能を使っています)

出力の型

式の入力欄の右側になる歯車アイコンは、出力するデータの型を設定するボタンです。



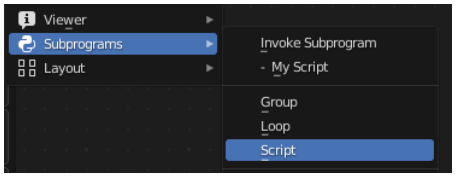
出力する型を例えば「 Vector 型」など明示的に指定することができます。
デフォルトでは generic 型として何でも出力できる設定になっています。



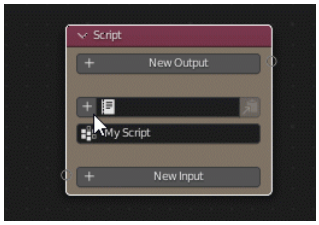
出力する型を設定すると、間違ったタイプの式を書いたときに警告が出るようになります。
安全を重視した慎重なノードの組み方をするなら、設定しておいた方が良いものです。
(変換が可能であれば変換は行われます。テストでサクサク実験するような時には無視することも多いですが…)

スクリプト

エクスプレッションの入力欄は１行しかないので、ちょっとした計算式の入力には良いのですが、複雑な処理には向いていません。
スクリプトは、一行に収まりきらないような計算や処理をさせる時に使います。
エクスプレッションよりも多少大掛かりな準備が必要です。

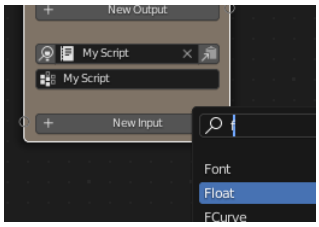


Subprograms - Script というメニューからノードを作成します。



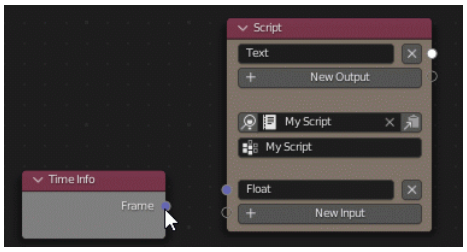
スクリプトとして実行するテキストを用意しないといけません。

スクリプトは、blenderファイル内の(もしくは外部のテキストファイルの)テキストを使います。
すでにあるテキストを選ぶか、＋ボタンで新規テキストを作成できます。
動画)Script01.gif

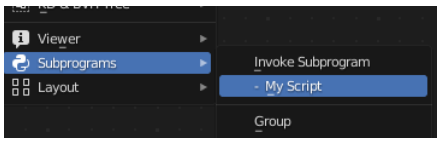


New Input と New Output ボタンから、入出力のための変数の種類を設定します。
最初に種類をリストから選び、その後で変数名を決めます。

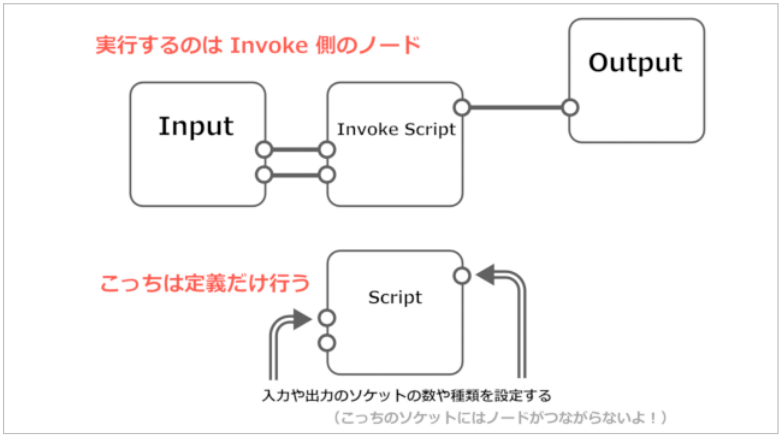
単純なオブジェクトの移動の例はエクスプレッションでやったので、
ちょっと変わった例として、「入力を Float、出力をテキスト」にして、時間で文字を変えるアニメーションを作ってみます。



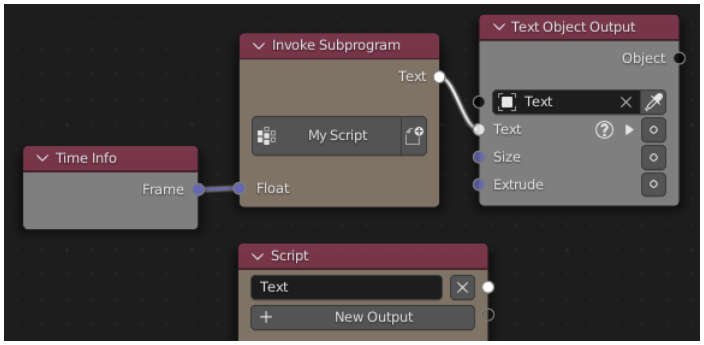
ところが、入力ソケットに時間情報を繋ごうとしてもつながりません。
実は、このノードはあくまで「スクリプトの設定をする」だけの機能しか持たないで、実際にスクリプトを実行するのは別のノードになります。
動画)Script02.gif



Subprograms - Invoke Subprogram のメニューで、今作成したスクリプトノード（My Script）を選ぶと、実行用のノードが作成されます。
Invoke は、普段はあまり見かけない英単語ですが、呼び出すという意味です。
関数を呼び出す(Call)に近い意味合いのようです。

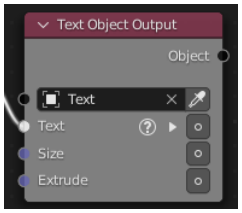


図のようにノード全体はスクリプトの定義と実行の２つのパートに分かれることになります。



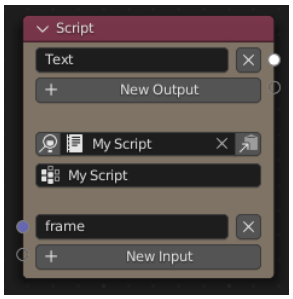
Invoke Subprogram のノードの方のソケットに時間情報を接続します。
出力されるテキストを利用する Text - Object Output ノードにも接続します。

また、出力するテキスト情報を利用するために、テキストオブジェクト(Text)を一つ作成して用意しておきます。
Text - Object Output で出力先のオブジェクトに指定しておきます。



Output 系のノードの一部は、機能を有効にするかどうか…が白い丸でON/OFFが表示されます。
Text はデフォルトで OFF なので、クリックしてONにしておきます。

ON状態であれば、テキストの内容をスクリプトからの出力によって変更できます。
（最初の状態では、ソケットがつながっているのにOFFになっているので、**注意喚起に"？"マークがついています**）



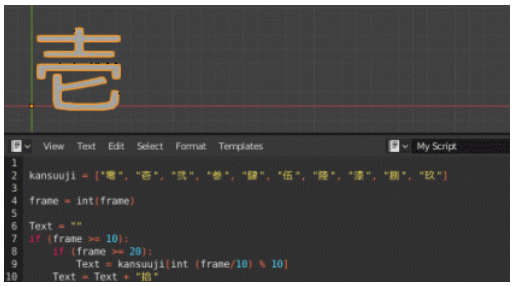
スクリプトの定義側のノードも下準備をします。
入力ソケットや、出力ソケットの名前を必要に合わせて変更をします。

この名前が、スクリプト中で使う変数の名前になります。
例えば、Float 型の入力ソケットのデフォルト名は "Float" なのですが、
スクリプト内の変数名として "Float" という名前は混乱を招きそうで良くないですね。
入力するのが時間情報なので、frame と名前を変えました。

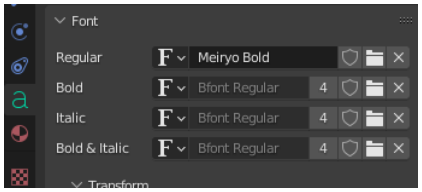
スクリプトの中身は、折角ですのであまり単純な例ではなく、少しややこしいスクリプトにしましょう。
次のようなものにしました。

```
kansuuji = ["零", "壱", "弐", "参", "肆", "伍", "陸", "漆", "捌", "玖"]

frame = int(frame)

Text = ""
if (frame >= 10):
    if (frame >= 20):
        Text = kansuuji[int (frame/10) % 10]
    Text = Text + "拾"

if (frame % 10 != 0):
    Text = Text + kansuuji[frame % 10]

if (frame == 0):
    Text = "零"
```

入力されたフレーム情報を、漢字の数字に置き換えてみます。（９９までの対応です）
Time Info ノードから得られた時間情報は、ソケットの名前"frame"としてスクリプトに渡されています。
その値に応じて 出力用の変数"Text"を変更すると、それがアニメーションノードへと出力されます。



漢字表記でカウントが進むアニメーションです。
動画)Script03.gif

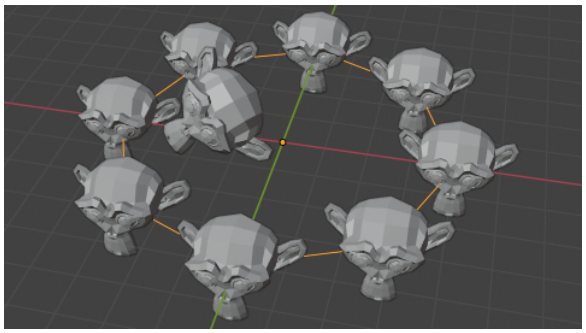


⚠
テキストオブジェクトの設定で、フォントなどをあらかじめ日本語対応のものを選んでいないと、
日本語表示できないことに気を付けてください。

この例を、01_SCRIPT/01_Script.blend として同封しました。

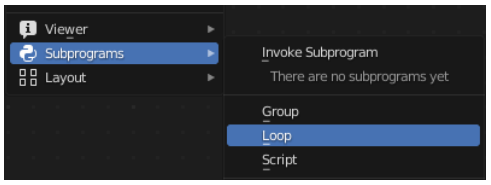
ループ処理

アニメーションノードは、数個のオブジェクトの管理をすることにも使えますが、はやり多数のオブジェクトを一括管理などをするのに使うのが便利です。
そうした際にはループ処理を行います。

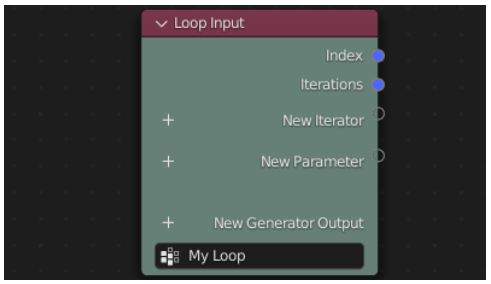


ここで、例として先ほど作ったこのセットアップを ループ処理を使うように拡張してみます。

ループ処理も、上のスクリプト処理と似たところがあります。
ループの設定をするノードと、それを実際に実行する Invoke のノードの2つを使います。

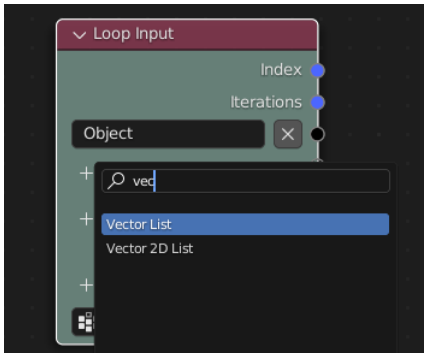


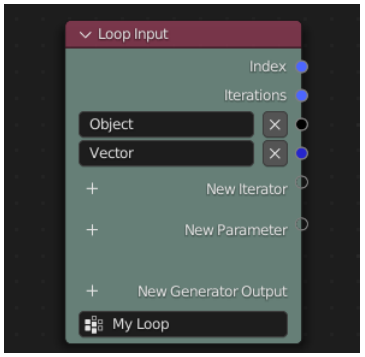
Subprograms - Loop のメニューからノードを作成します。
ループの名前はデフォルトの "My Loop" になっています。



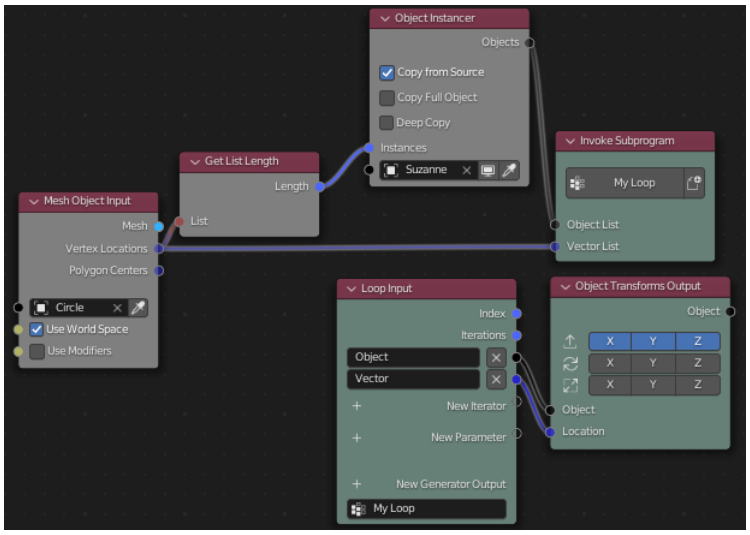
メニューから Loop ノードを作成すると、入力用の Loop Input ノードが作成されます。
「＋」ボタンが複数ある、少し変わったノードです。

ここで、New Iterator の「+」ボタンで、Iterator （反復子）を追加します。
Iterate もプログラミング以外ではあまり聞かない単語ですが、繰り返しを意味する単語です。
ここで「何に関するループ処理なのか」種類を決めます。





イテレーターを２つ追加します。
種類を選択して、Object List と Vector List に関してのループにします。



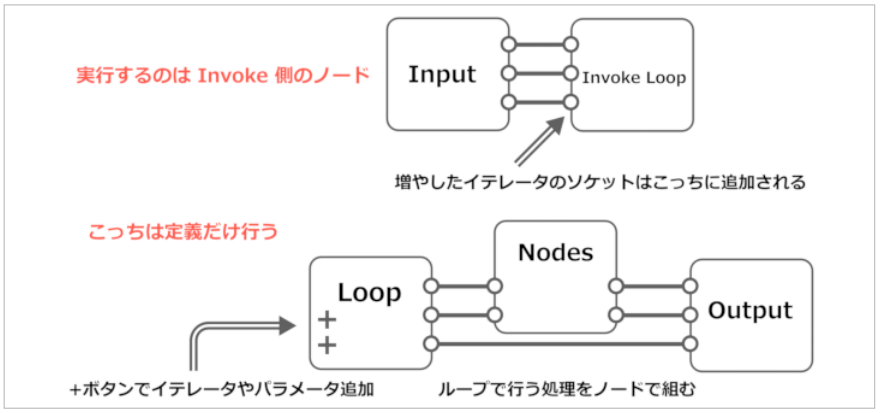
元々 Object Transforms Output があった場所で、
接続を切り離して、
Subprograms - Invoke Subprogram - My Loop をつなぎます。

これで、オブジェクトの移動の代わりにループ処理が実行されることになります。
では、ループ処理で何をするか…という事を設定しないといけません。

Loop Input のソケットから Object と Vector を Object Transform Output につなぎます。

この時点では機能としてはまったく同じです。
8つのスザンヌに関して、順番に位置を設定する処理のループが行われることになります。
1番目のオブジェクトを、1番目のVector Listの位置に配置
2番目のオブジェクトを、2番目のVector Listの位置に配置
…
というループ処理を明示的に行っていることになります。

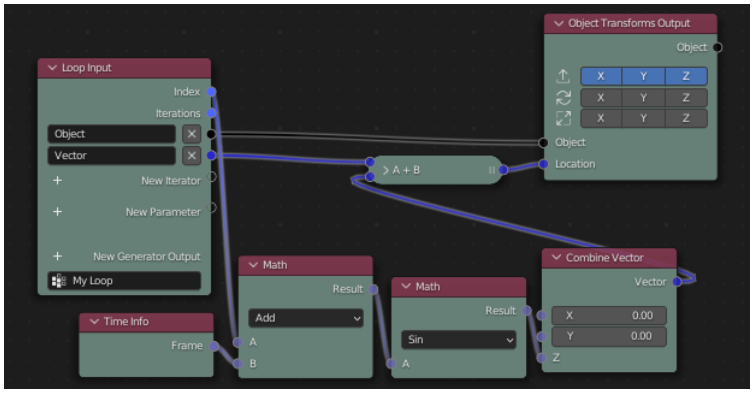
下半分の、Loop Input のノードから繋がっている部分は、ループ処理の定義をしている部分です。
実際にループを実行するのは Invoke Subprogram のノードです。



定義部分では、オブジェクトの位置を変更する、という機能の設定だけが行われています。
この場合は、Object Transform Output によって「オブジェクトの位置を移動する」という機能です。

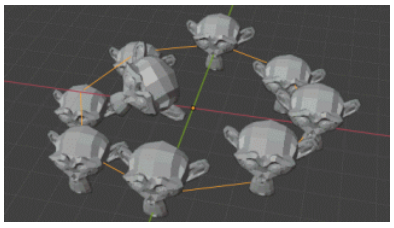
Invoke 側で、さっき定義したループ処理（My Loop）が実際に実行されているわけです。

このままだと、Object List を Vector List に従って位置を移動する…というだけなので、
そもそもループを利用しない最初の例とまったく同じことをしているだけです。
スザンヌ位置を変化させるようように、ループの処理の定義部分を変更してみます。



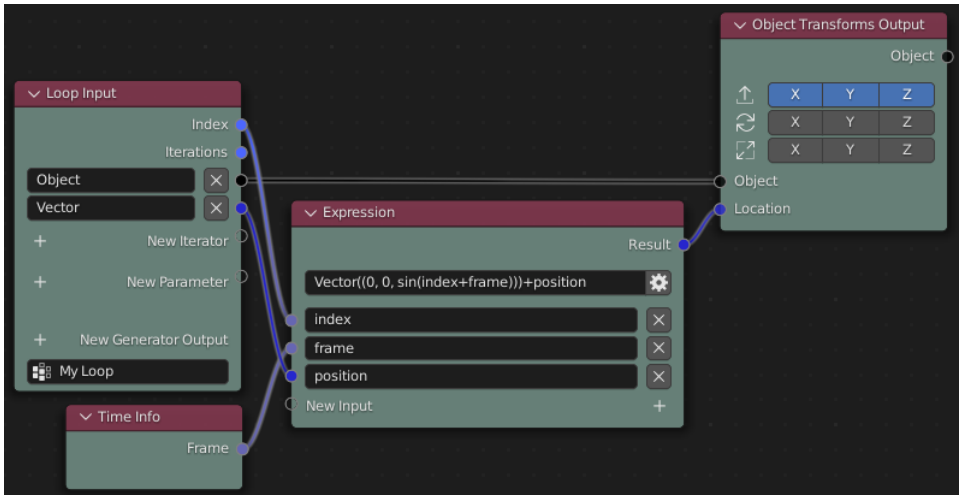
Loop ノードの Index ソケットは、その名の通りループのインデックスになります。
（Iterations はループのトータル数、この場合は 8 になります）

Index と 時間情報(Time Info)を足して、オブジェクトごとにずれた位相で振動(Sin波)するベクトルを作成します。
それを元の位置ベクトルと足し合わせて、Object Transforms Output につなぎます。



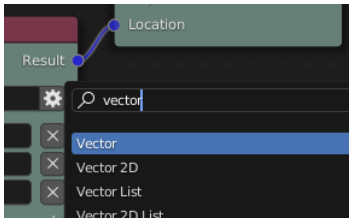
バラバラに上下動するスザンヌの動画を作ることができました。
動画)Loop01.gif

このように「バラバラに動くオブジェクト」自体はジオメトリノードでも
「インスタンスを配置する位置を Index と時間情報に応じて動かす」ことで実現できます。
そこで、アニメーションノードの利点を生かす為に、この位置を設定するノードをエクスプレッションで作成をしてみます。



エクスプレッションを利用することで、だいぶすっきりします。
スクリプトの入力などに慣れていれば、多くのノードをつなぐよりもこの方が扱いやすいかもしれません。

この場合の出力は位置情報なので、
（デフォルトの Float 型ではなく）Vector 型である必要があります。

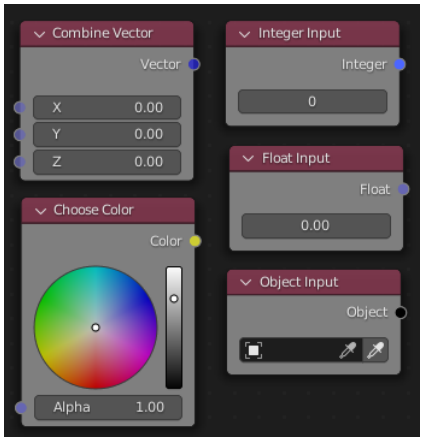


歯車マークで、出力のタイプを変更します。

このスザンヌを動かすのにループ機能を使用した例を、01_SCRIPT/02_Loop.blend として同封しました。

リストについて

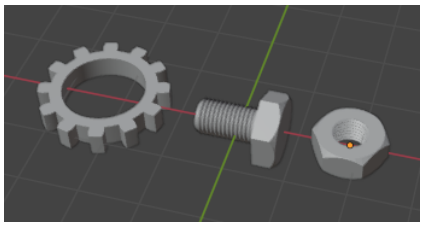
今までまずはサンプルを見ながら、「そこそこ面白いことができるところまで進もう」と、やや応用めいた内容に踏み込んでいました。
ここで、少し基本に戻ります。
今までに、特に説明なしに出てきた「リスト」機能に関して説明します。



アニメーションノードでやり取りするデータには幾つも種類があります。

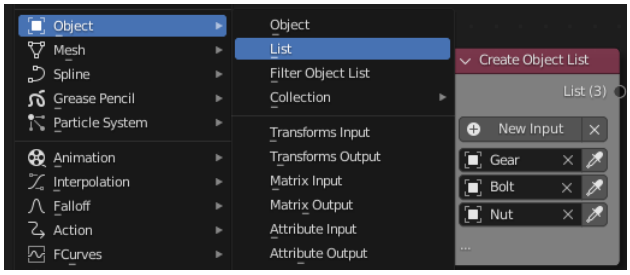
数値であれば例えば整数(Integer)や浮動小数(Float)
その他、色やベクトル、オブジェクトそのものといったものがあり、
ソケットの色で区別がつくようになっています。
これらのデータのタイプは、対応したただ1つのデータをやり取りします。

リストは、複数のデータをまとめて扱えるようにしたものです。
それぞれのデータタイプには、対応するリストのタイプが用意されていて、
一度に複数のデータをノード間でやり取りができるようになっています。
（ソケットの色が少し薄くなって、区別できるようになっています）

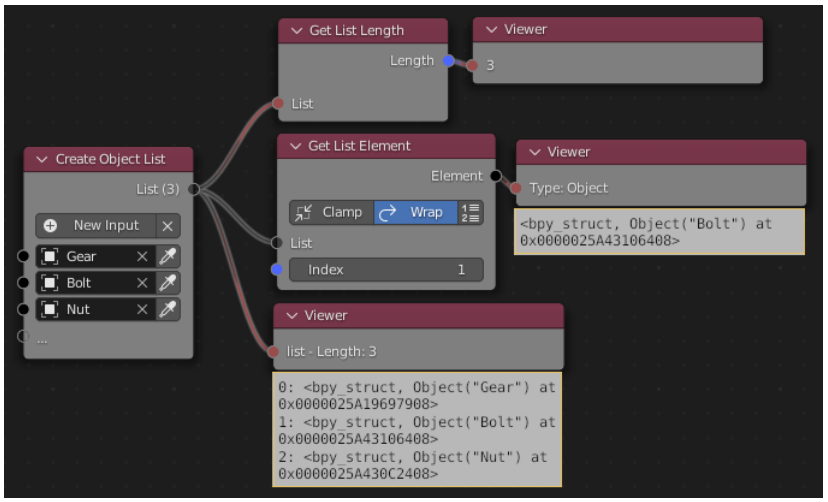


例として、
標準アドオンを使って、ギア、ボルト、ナットを作成しました。

リストを作るメニューは２通りあります。
List - Create - Object
Object - List
どちらかのメニューから Create Object List のノードを作成して、オブジェクトのリストを作る事ができます。



New Input によってリストの数を増やし、
3つのオブジェクトのリストを作成します。
それぞれ、オブジェクトを選択します。

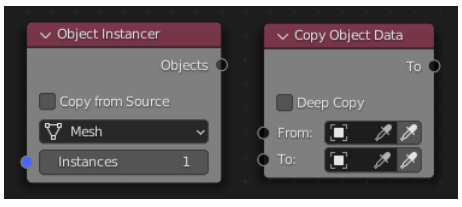


ビューワーノードをつないでみると、中身をいろいろとみることができます。
直接つなげると、オブジェクトのデータが3つ並んでいることがわかります。

Get List Length (List - Get Length) のノードにつなぐと、
リストの長さが3であることが分かります。

Get Lsit Element (List - Get Element) のノードをつなぐと、
リストの中から1つのオブジェクトを取り出せることが分かります。

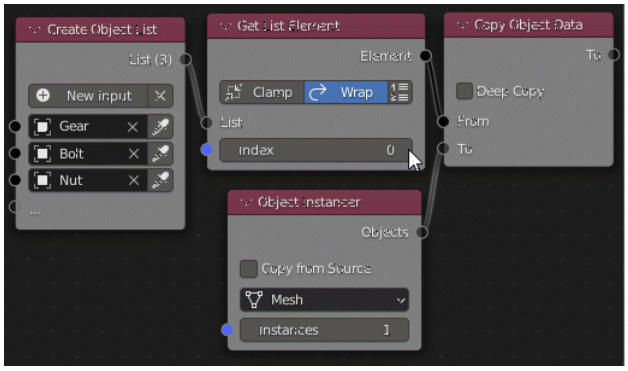
オブジェクトリストを利用して、形状の差し替えをしてみます。

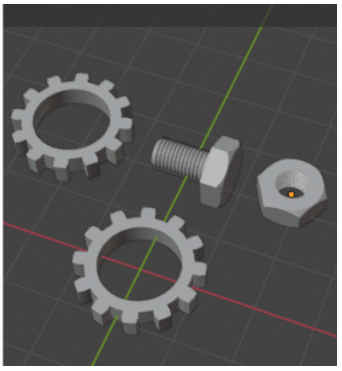


インスタンサーでオブジェクトを作成するときに、Copy from Source のチェックを外すと、
完全に何もない空のオブジェクトを作成します。

Object - Copy Data のノードを使うと、To のオブジェクトのメッシュを
From オブジェクトのメッシュデータに差し替えることができます。

そこで、以下のようにノードを組むことで、数値の指定で形状の差し替えをすることができます。

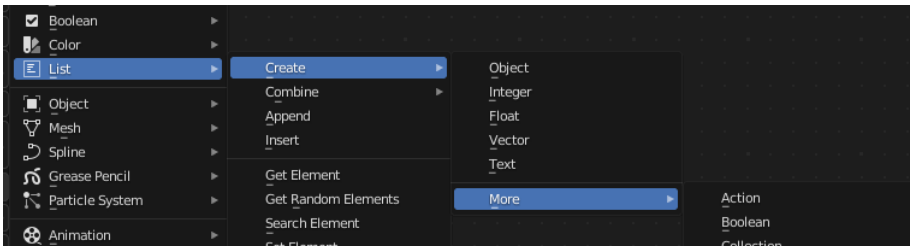




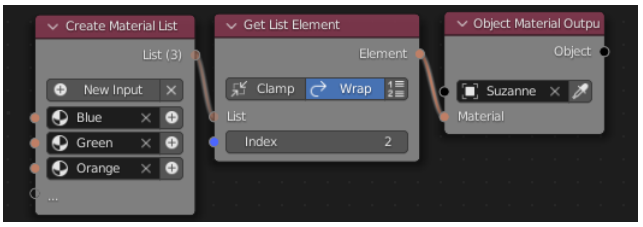
もちろん、Indexを手で編集する代わりに、
Time Info ノードなどをつないで時間進化させることもできます。
動画)List01.gif

マテリアルのリスト

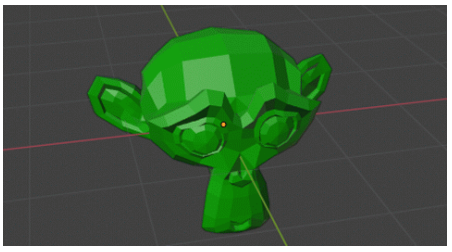
アニメーションノードで扱えるデータには、それぞれリストを作ることができます。
別の例としてマテリアルのリストを利用してみましょう。



マテリアルのリストは、（頻出するリストではないと分類されていて）メニューの深いところにあります。
追加メニューから List - Create - More とメニューをたどっていくと、作れるリストの一覧がメニューから選べるようになっています。



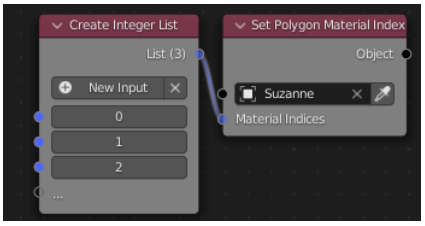
マテリアルのリストから要素を選んで、Object Material Output ノードにつなぎます。
Get List Element を使ってリストの中から１つマテリアルを設定できます。

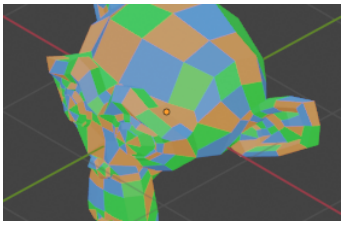


Index の値を変えることで、マテリアルの差し替えをすることができます。
使用しているマテリアルの種類は、キーフレームを打ったアニメーションのできないパラメーターなのでアニメーションノードの利点の一つです。
動画)List03.gif

ただし、この場合はスザンヌが単色（先頭のマテリアルスロットを１つのマテリアルで差し替えれば良い)なので制御が簡単なのですが、
マテリアルを複数使ってリストをアニメーションさせたいような場合は、多少面倒かもしれません。
マテリアルが複数ある場合の例は後のメッシュの説明の章で見てみます。

その他、Mesh - Mesh Data - Set Polygon Material Index のように、ポリゴンごとの使用するマテリアルインデックスを変えるノードもあります。





値１つではなく、整数のリストを渡すことで、面ごとに違うマテリアルを指定することができます。

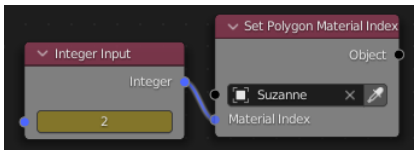
例えばスロットに３つのマテリアルが設定されている時に、0,1,2 のリストを渡すとこのように面ごとにマテリアルが設定されます。
（はみ出した分は繰り返し処理がされるようです）

ただし、ポリゴンのインデックスを知ってリストを作る必要があるため、モデリングしたメッシュを制御するのはちょっと厳しそうです。
「一定のルールに従ってマテリアルをセットすればよい」という規則的な形状や、エフェクト用のオブジェクトの時が主な使い道でしょうか。

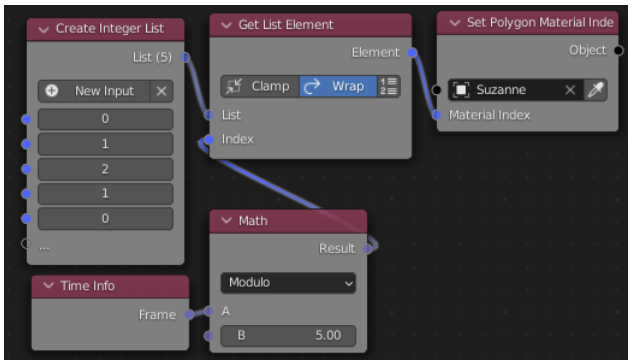
ノードのキーフレームアニメーション

Time Info などを利用すれば、動くオブジェクトなど時間によって変化するアニメーション表現ができます。

それとは別に、「アニメーションノード内の変数に対してキーフレームを打って」アニメーションを表現するという考えもあります。
ところが、ノードの変数にキーフレームを打とうとしても、上手くいきません。
これは、どうやら現時点(3.3)で Addon として追加できるカスタムノードに対しては、キーフレームを打てない（打ってもうまく働かない）という Blender の問題のようです。
（このトピック自体が、TODO のカテゴリーに分けられていました）



入力する数字部分に ショートカットの I などでキーフレームを打って、アニメーション制御しようとしても、（キー自体は打てるのですが）編集用のグラフなどには表示されず、数値も変化しないという状態になります。



そのため、ノードの変数を変化させる必要があるアニメーションは

「制御用に別のオブジェクトを用意してその位置情報などを使う」
「リストと Time Info 、Get List Elemet を組み合わせて変化する値を表現する」
「エクスプレッション(とTime Info)を使って直接式で表現する」

などの手段を使って、実現させることになります。

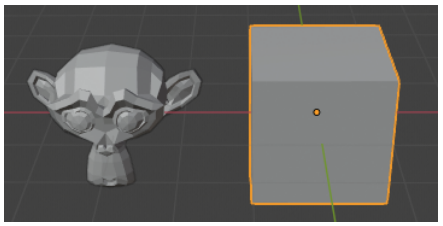
この例はマテリアルを 0,1,2,1,0,… のように時間変化させるために、リストを利用した例です。

メッシュの操作

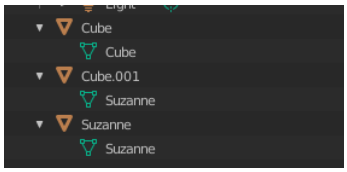
オブジェクト単位での移動や、要素の差し替えといった操作を行ってきました。
今度は、一つのオブジェクト内のメッシュを変形させる操作を行ってみます。
そうしたメッシュの変形のような操作は、ジオメトリノードを使う方が簡単という局面が増えてきましたが、アニメーションノードでそれを行ってみます。

メッシュデータの基礎

単純にスザンヌのメッシュデータを、キューブにコピーしてみます。

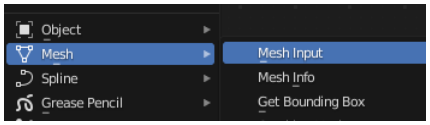


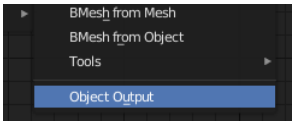
まずはデフォルトキューブの脇にスザンヌを用意します。



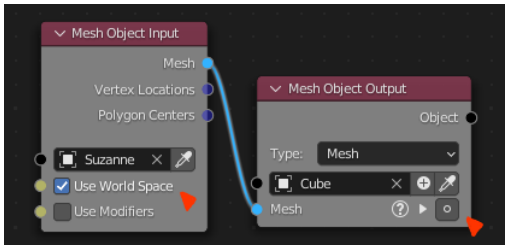
キューブのオブジェクトが使うメッシュデータを Suzanne に差し替える…
という操作はCopy Object Dataノードで行うことができました。
左の図はそうやってメッシュを差し替えた状態を、アウトライナーで確認した例です。

今度はそうではなく、キューブの持つメッシュ(Cube) のデータ**自体**を書き換えて、
スザンヌのデータのコピーに変更してみます。



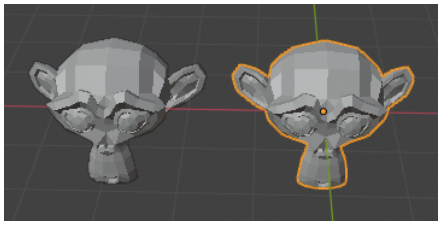


Mesh - Mesh Input と Mesh - Object Output でメッシュの入出力用のノードを用意します。
メッシュのサブメニュー先頭と最後にあります。



Mesh のソケット同士をつなげばメッシュデータがコピーされます。

ここで注意点が２つあります。
Mesh Object Output は安全のために最初は無効になっています。
無効なソケットにつないだ時は、？マークが出ているので有効にしておきます。
また、Use World Space が有効だと、コピーしたスザンヌが完全に元の位置に重なってしまうので、
このオプションは外しておきます。



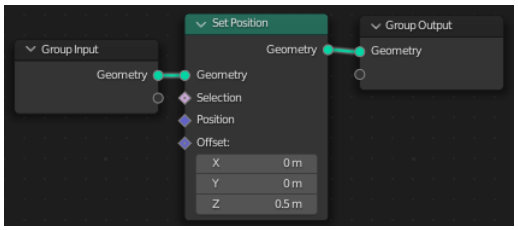
これで使うメッシュを差し替えるのではなく、
メッシュデータ自体が、別のメッシュデータのコピーに書き換えられたことになります。

ジオメトリノードは、モディファイアなので非破壊編集です。
元のメッシュ形状 -> ジオメトリノードによる変形 -> 変形した結果を表示。
という流れがありました。

一方アニメーションノードの場合は、メッシュの情報を書き換える…という考え方なので、直接元の形状を編集すると元の形が残らない破壊的な編集になります。
そのため、非破壊な編集をしたい場合は、
元形状用のオブジェクトを別に用意しておいて、その形状をコピーして使う という運用になります。
（表示する必要がなければ、最後に非表示で隠しておけばいいわけです）

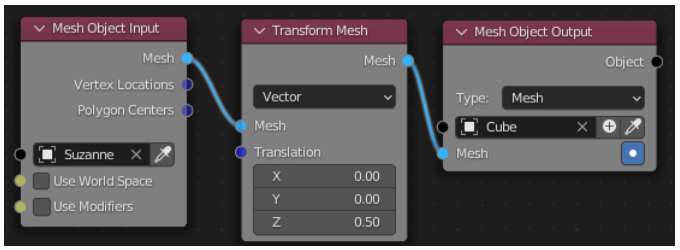
メッシュデータの変更、配置

メッシュのデータを Mesh Object Output ノードにつなぐ前に、メッシュを操作するためのノードをつなぐことで様々なことができます。
頂点１つ１つを細かく制御するようなことを考えると、「頂点一つ一つに何らかの処理をするためにループ処理をする」ということになり、随分大げさになってしまいます。
しかし、ループ処理をしないでも、それなりの処理が行うためのノードが用意されているので、まずは「ループを使わないで行える処理」を見てみます。

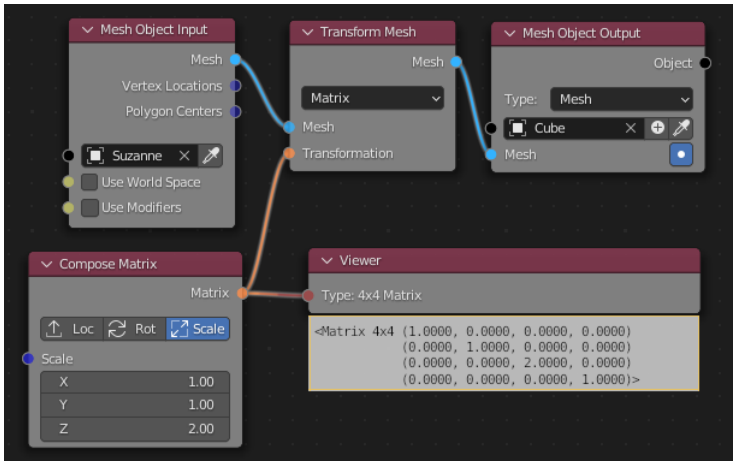


ジオメトリノードで**各頂点を(平行)移動する**ような操作は、
簡単に行うことができます。

アニメーションノードで相当する操作を見てみましょう。



メッシュの変形を行うのは、Transform Mesh ノードです。
モードを Vector にして、ベクトルの定数を入力すれば、平行移動できることになります。

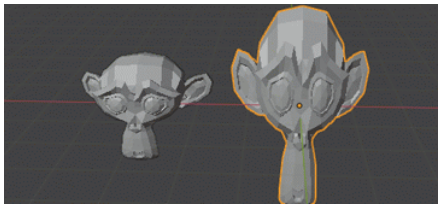


平行移動ではない変形を（拡大縮小や回転）を表現するときには、モードを Matrix にします。
つなぐソケットは Transformation となっていますが、これの正体は 4x4 の行列(Matrix)です。

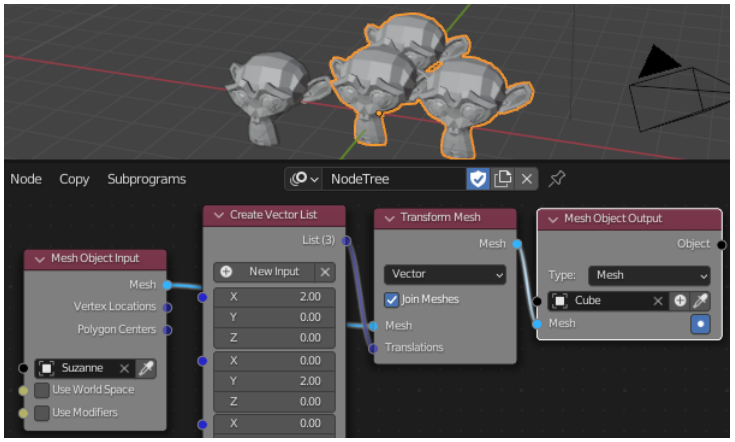
行列は、3DCGの表示の仕組みなどに踏み込むと良く出てくるものですが、要は「平行移動」「回転」「拡大縮小」（その他にもひずみ変形など）をまとめて 4x4 の数字で表現できるものです。

線形代数などを選択していないとなじみが薄いものですが、とりあえず使う分には中身を知っている必要はないです。
（知っていればより理解しやすいものではありますが…）

「平行移動」「回転」「拡大縮小」から Matrix を作成するノードは Compose Matrix ノードです。
(Matrix - Compose)



例えばスケールのz成分を変化させると、メッシュがz方向に拡大縮小するのが分かります。
動画)Transform01.gif



また、Transform Mesh のノードにはリストを渡すことができます。
その場合はリストの数だけ複製が配置されることになります。
Join Meshes オプションを有効にすると、複製された形状を１つのメッシュにまとめて出力します。

例えば、ベクトルで位置情報を並べたリストを使うと、
数値で指定した任意の場所に複製を置く、小数多体用の複製ノード
のように使うことができます。