マニアックなシェーダーの話

（Eevee用のマニアックなシェーダーあれこれ）

Version 2.2

Q@スタジオぽぷり
@popqjp

作者 Twitter アカウント
Q@スタジオぽぷり

目次

初めに … 3

カメラ座標でのテクスチャとハッチング … 4
カメラ座標のテクスチャ … 4
ハッチング … 6

カメラ座標のボロノイ分割(クロスハッチング) … 9
ボロノイ分割によるタイリングのなじませ … 11
ノイズとブラー … 13

Mat Cap あれこれ … 15
カメラ座標を使った Mat Cap … 15
ワールド座標を使った Mat Cap … 17
ブラシストロークのような効果の Mat Cap … 20

バンプマップの小技 … 22
その高さ、あってる？ … 22
バンプマップの重ねがけ … 23

角の摩耗表現 … 25
反射とシェーダーミックス(Mix Shader) … 26

ジオメトリノードの利用 … 28
色やパラメータ情報の受け渡し … 28
インスタンス単位、シーン単位での情報の受け渡し… 30
UDIMを利用したテクスチャの切り替え…32

Parallax Mapping(視差マッピング)とParallax Occlusion Mapping(視差掩蔽マッピング) … 34
Parallax Mapping(視差マッピング) … 34
Parallax Mapping の例 … 38
Parallax Occlusion Mapping (視差掩蔽マッピング) … 41

Interior Mapping(インテリアマッピング) … 45
左右の面 … 45
上下の面、奥の面 … 47
陰影の設定 … 48
Equirectangular(正距円筒図法)を使ってみる … 49

Shell 法による 凹凸の表現 … 51
シェルの組み立て … 51
シェル用のシェーダー … 53
Shell 法の利点、欠点 … 53
シェル法とジオメトリノード … 54

Shell 法によるファーや雲の表現 … 56
アルファハッシュ法による不透明 … 56
毛皮（ファー）の表現 … 57
毛の変形 … 59
Shell 法による雲 … 60

AOVとCryptmatte(クリプトマット) … 63
AOV Output … 63
Cryptomatte(クリプトマット) … 64

半透明オプションとレンダリングパス … 66
加算乗算処理、輪郭線処理への影響 … 67
スクリーンスペース屈折 … 68

Light Probe(ライトプローブ) … 70
Reflection Plane(反射平面) … 70
Reflection Cube(反射キューブマップ) … 71
Irradiance Volume(イラディアンスボリューム) … 74

サンプル.blendファイル … 76
終わりに … 76

マニアックなシェーダーの話

(Eevee 用のマニアックなシェーダーあれこれ)

作者 Twitter アカウント
Q@スタジオぽぷり

2020.12.30 ver 1.0
2021.01.07 ver 1.2
2021.03.12 ver 1.4
2022.01.29 ver 1.6
2022.11.18 ver 1.8
2022.12.30 ver 2.0
2023.06.27 ver 2.2

初めに

この本は、Blender の、特に Eevee でレンダリングする時のマテリアル設定、シェーダーについて書いた本です。

シェーダーの使い方にはいろいろありますが、まず一番に思いつくのが普通のマテリアル設定での使い方です。
背景の物体のマテリアルに、リアルな質感を設定したり、キャラクターに見栄えのする肌や目の表現を入れたり、服装の細部の模様を作りこんだり…などです。
ところが、ツィッターなどで作者の作風を知っている方はお分かりでしょうが、作者はそうした絵作りのノウハウに関しては、大した知識はありません。

代わりに、いろいろな効果を入れたり、エフェクトを入れたり、立体表現を入れこんだりと、ややマニアックな使い方をあれこれとしています。

これまでにそうして作ったマテリアルや、そのためのノウハウの中で公開したら役に立つ人もいるのではないか、というものをまとめました。
そのため、テーマなどはなく話の流れもとりとめのないものですが、お付き合いください。

この本は、シェーダーとしてノードを組み立てることには既に慣れている方を読者に想定しています。
操作法や、主だった機能については大体分かっているものとして、説明を飛ばしていることが多いです。
Blender やその他の CG ソフトにはほとんど触ったことが無く、初めて弄ってみる、という場合や、
マテリアルの設定にはほとんど手を出していなかったけど、ちょっと興味があるからやってみよう、
という場合は厳しいかと思いますので、そういう場合はまず基本的な入門書などから手を出してみてください。

また、どうしても座標に関する話は避けて通れないので、高校幾何の範囲の知識、内積や外積といったものがどのようなものかという知識は必要になると思います。
スナップショット画面も、一部を除いて英語設定でスナップショットを取っています。
（スナップショットの中の注意書きは日本語で書いています）
元々が英語のソフトですので、日本語設定だとノード幅にうまく合わなかったり、訳が微妙だったりすることが多いので、ご了承くださいませ…

本書のver 1.0の内容は、主に Blender 2.91 から 2.92 の時点で書かれています。
ver 1.2 では、視差マッピング(Parallax Mapping)の亜種として、窓から覗く部屋の内部などを描画するインテリアマッピング(Interior Mapping)の内容を追加しました。
ver 1.4 では、半透明オプションとレンダリングパスについての内容を追加しました。
ver 1.6 では、AOノードを使ってのエッジ検出と、Blender3.0で強化されたジオメトリノードを使ったシェーダーに関するノウハウを追加しました。
ver 1.8 では、Blender の 3.3 ごろの内容に合わせた記述の更新と、幾つかのサンプルの追加を行いました。
ver 2.0 で、Blender 3.4 で追加された機能を使う例と、コンポジットとの連携についての記述の追加を行いました。
ver 2.2 で、Blender 3.5 に対応した記述の変更と、ブラーのサンプル、Light Probe(ライトプローブ)のセクションを追加しました。

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。

カメラ座標でのテクスチャとハッチング

カメラ座標のテクスチャ

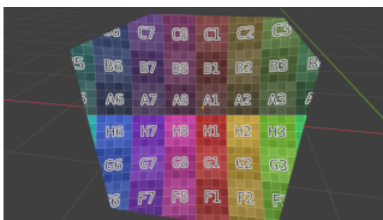
テクスチャというものは、物体の表面に張り付けるので、通常はポリゴンの形状の凹凸をきっちり感じとることができます。
というか、バンプマップやノーマルマップなど、立体感を強調するために利用することが多いです。

ところで、一部のエフェクトや、ハッチングなどの2次元に近い表現をしたい場合いにはむしろ立体感が邪魔な場合があります。
そのような場合にカメラ座標を使ってテクスチャを貼り付ける時のやり方を幾つかまとめます。

ビューベクトル(View Vector)

Input - Camera Data からカメラから見た座標を得られます。

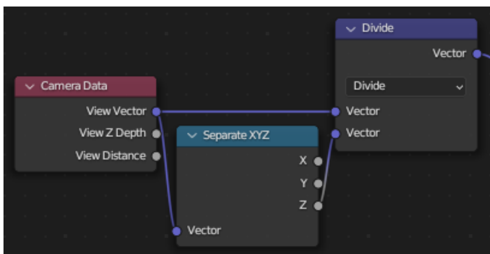
このベクトルは長さ１の単位ベクトルで、カメラを中心にした球殻への投影と考えることができます。
z 方向が画面の奥になっていて、x,y 方向がカメラの左右上下です。



これを使ってテクスチャを表示すると、カメラを中心にテクスチャを貼っているように見えますが…

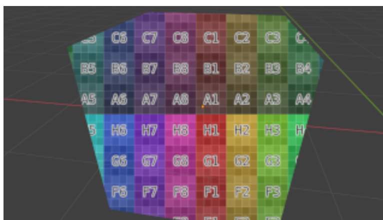
球殻なので、透視図法のピクチャープレーン（平面）とは違うので、
視野が狭い時はいいのですが、視野が大きいと差が歪みとして見えてきます。

左の図のグリッドが少し曲がっているのが分かるでしょうか。



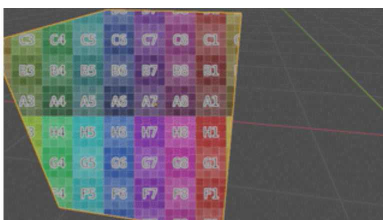
最も簡単に歪みを補正するのは、z方向成分で割る方法です。

zが1よりも大きい場合は、座標が縮小され（表示は大きくなります）
zが1よりも小さい場合は、座標が拡大され（表示は小さくなります）
結果として z = 1 のピクチャープレーン（平面）に投影された状態になるわけです。



表示の歪みが無くなりました。

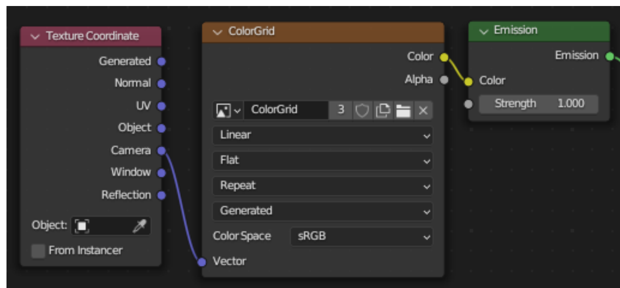
四則演算などの計算の時には、数式ノードでの値としての計算と、
ベクトル演算ノードでのベクトルとしての計算は、間違えることがあるので気を付けてください。



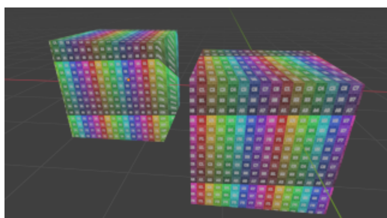
このやり方は、当然ながら完全にビューに固定されています。
オブジェクトが動いたりカメラが動いたりしても完全に追随します。
(動画)ViewVectorA.gif

カメラテクスチャ座標 - カメラ (Texture Coordinate - Camera)

もう一つ似た機能を持つのが、
Input - Texture Coordinates - Camera です。

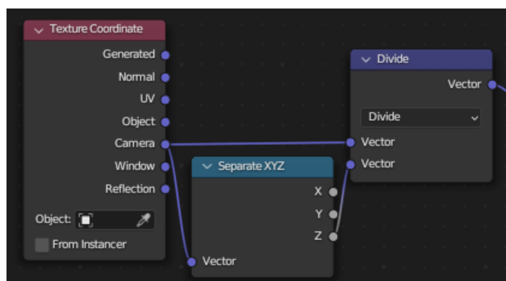


これはカメラ位置を原点として奥をz方向とした、
カメラを中心とした座標系そのものを表します。

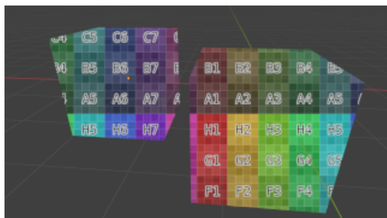


View Vector の時には、長さ１のベクトルに変換していましたが、そのような変換無しに位置を表しています。
透視図法によるパースの影響はそのまま受けるので、遠くでは小さく近くでは大きく映ります。

この場合の立方体の側面のように、視線方向に対して垂直に近い面では、
パースの影響を大きく受けます。



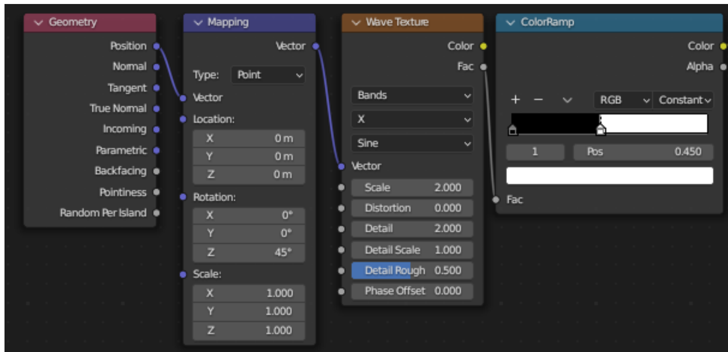
Camera View で得られたベクトルとは、長さが違うだけで方向は一緒です。
そのため、面白いことに z で割る、という操作をすると同じベクトルになり、同じ結果が得られます。



View Vector で同じ操作をした時と同じ結果になりました。
違いは 対応する z の値だけで、方向は同じものなのでこうなるわけですね。

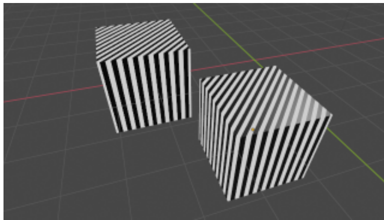
ハッチング

単純なハッチング（縞模様）をこれらの座標系で使ってみます。



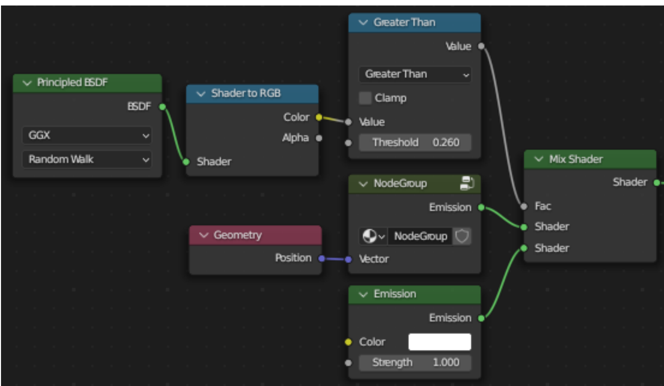
Wave Texture と Color Ramp を使って
境界のはっきりした縞のノードを組みました。

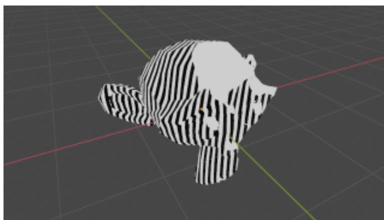
座標は、まずはグローバルな座標系で試します。
斜め縞にするために、マッピングで45度回転をさせています。



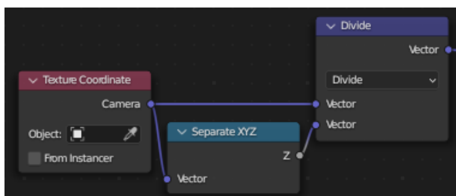
そのままだと z 軸方向から見て斜めになる線が入ることになります。
ノードをコンパクトにするために、このノードの組をノードグループ[Hatching]としてまとめておいて利用します。

影になる部分にだけ縞が入るように、Shader to RGB ノードを利用します。



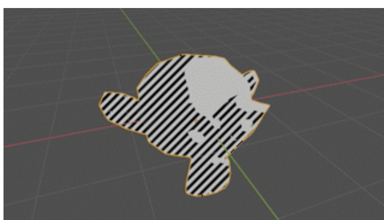


スザンヌに適用しました。影の部分にだけ縞が走っていますが、
模様が立体に沿って走っているので、ハッチングとは言い難い状態です。
これはこれで味はありますが…

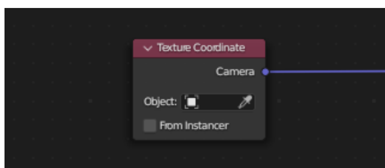


[Hatching]につなぐ座標系に、カメラ座標系(ビューベクトル)を利用します。
zで割って、歪まずカメラに完全に固定された座標です。

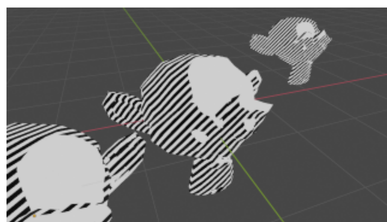
注）図を小さくするためにノードの余計なソケットは隠しています



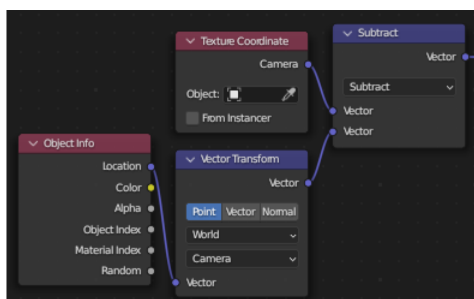
カメラから見た縞模様になりました。
ただし、完全にカメラの座標にあわせた模様なので、
模様がオブジェクトの動きとは全く無関係になるウィークポイントがあります。
(動画)HatchingB.gif



回避方法を一つ見てみます。
z で割らずに、そのままのカメラ座標を使います。

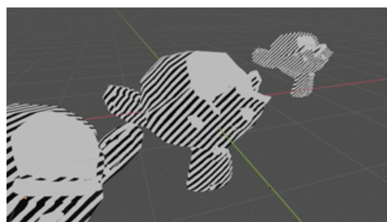


これは、距離に応じてパースがかかるので、当然ですが縞の密度が距離で変化します。
（この時点ではオブジェクトの動きと無関係なのもそのままです）



オブジェクトの動きを反映するように、
Object Info - Location でオブジェクトの位置情報を得ます。

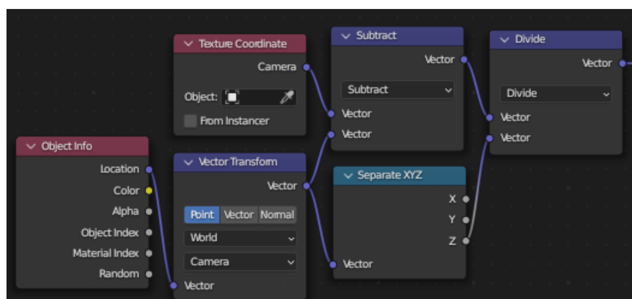
これを Vector - Vector Transform
でカメラ座標系に変換して、元の座標から引く、ということをします。
これにより、オブジェクトの移動の影響がキャンセルされることになります。



オブジェクトの動きに模様が追随するようになりました。
回転などには対応できませんが、縞の動きを気にせずに多少のカメラワークが可能になるので、
より広い状況に対応できます。
(動画)HatchingC.gif

この時点のサンプルを、01_HATCHING/01_SimpleHatching.blend として同封しました。

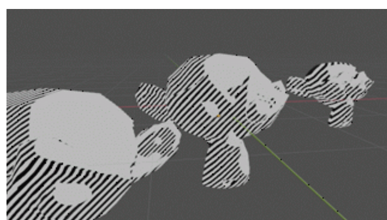
これらの考えを組み合わせて、さらにカメラからの距離 z で割るという考え方もできます。



あまり大きくないオブジェクトの場合には、
距離で縞のサイズを変えずに動きに追随するハッチングができます。
ただし、「オブジェクトとしての位置」を利用しているので、大きなオブジェクトでは「実際のポリゴンの位置」と「オブジェクトの位置」の差が大きくなって破綻をします。

※)この Vector Transform ノードはモードを Point にしておかないといけない点は注意です。

例えば地面などで、カメラの近くからずっと遠くまで占めているような物体で使うには無理があるので注意します。



え？今度は遠くのものの縞が大きく見える…？
確かに、距離によらずに縞の大きさが固定されるので、そういう印象になります。
例えば z の代わりに z のルートを使い、遠くの縞が多少細かくなるような工夫などが考えられます。
(動画)HatchingD.gif

向きの調整

完全にカメラの方を向いた一様な縞の場合は良いのですが、
距離に依存するようなハッチングは、形状と角度によっては木目のような模様が出てしまう場合があります。
カメラ座標に模様を変換しても、画角が大きいと画面右側と左側で見る角度が違うということが影響しています。
縞の向きとの関係から、右側のウサギの縞の具合があまりよくありません。

オブジェクトのサイズが大きくなければ、オブジェクトの位置に応じてカメラ側を向くようにテクスチャの向きを回転させることで、軽減ができます。
オブジェクトのカメラ座標での位置から、Arctan2を使って角度を求めています。
（逆向きに適用したいので-1をかけています）

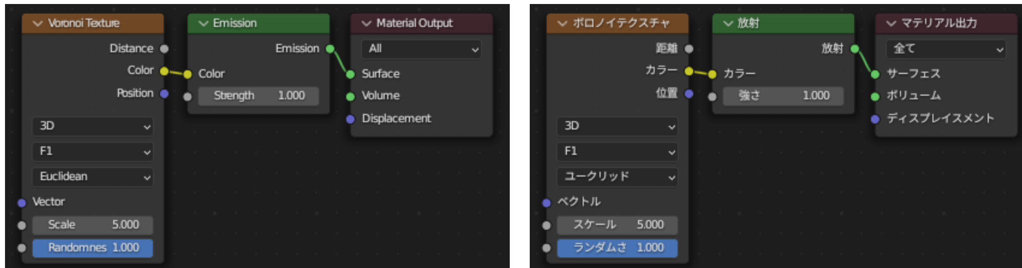
補正されて、だいぶ良くなりました。

この状態のノードの全体図です。

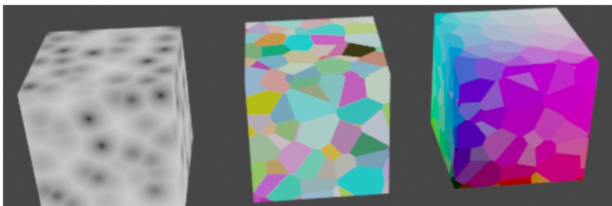
ハッチングのサンプルとして、01_HATCHING/02_Hatching.blend をサンプルに入れました。

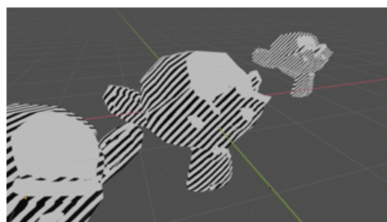
カメラ座標のボロノイ分割 (クロスハッチング)

ここで、カメラ方向を向いた座標系と、ボロノイを組み合わせた面白い使い方があるので、それについて解説します。
ボロノイは、割れ目のようなエフェクトによく使われるテクスチャノードですが、出力のソケットが 3 つあります。



Color と Position のソケットからは、各セルごとにランダムな値と、セルの中央に相当する位置の情報が得られます。

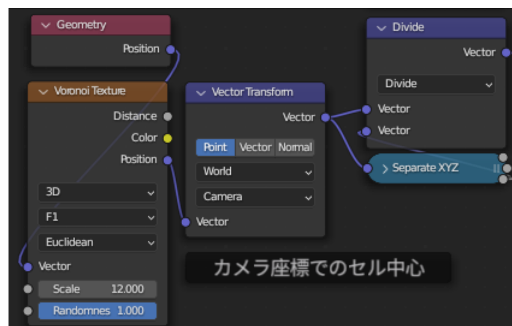




ところで、先ほどのハッチングの項目で、
「オブジェクトの位置情報を引くことで、オブジェクトの動きにも追随する」効果をみてみました。
この動画が出てきたところですね。
(動画)HatchingC.gif

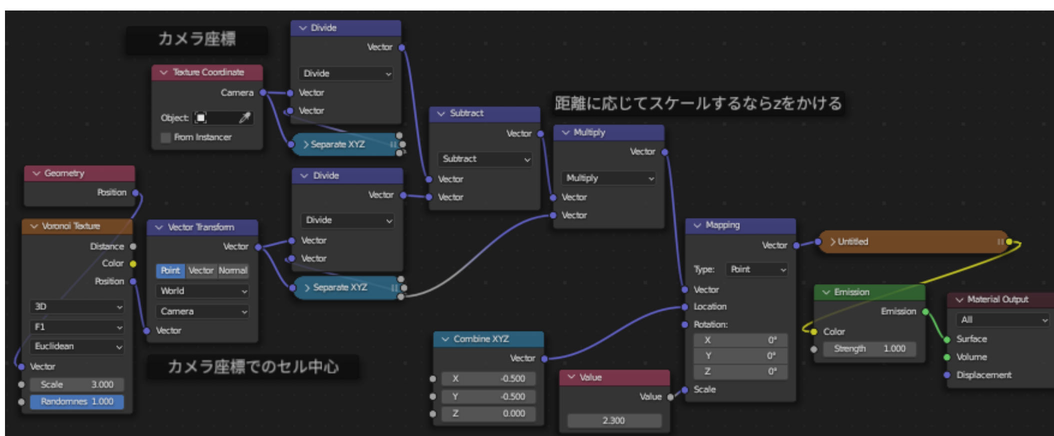
ここで同じように「セルの中央の位置情報」を引くことで、各セルの位置ごとに座標系を作成することができます。

ノードを組んでみます。



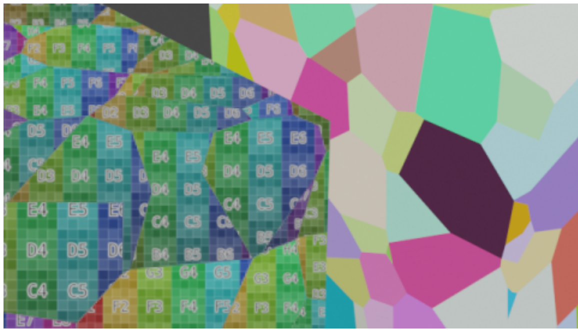
ボロノイノードの Position 情報を、Vector - Transform でカメラ座標へと変換して使います。

カメラ座標からこれを引くことで、セル中心を座標の基準にすることができます。
その際、距離に依存するかどうかで z 成分で割る処理を する/しない を切り替えます。

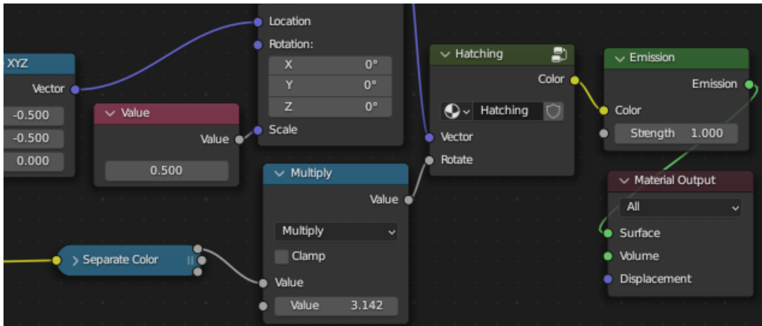


ここでは、z で割って、距離に依存しないようにして計算した後、
最後に z をかけることで、セル中心までの距離に依存してスケールするようにしました。

この時、セル中心が座標の中心になってしまうので、テクスチャで利用するUVのように(0.5, 0.5)が中心にするならば、Vector Mapping で 0.5 だけ平行移動することになります。



この座標系を利用してグリッドを貼ると、
それぞれのセルの中に独立してカメラの方向を向いた座標が貼られました。



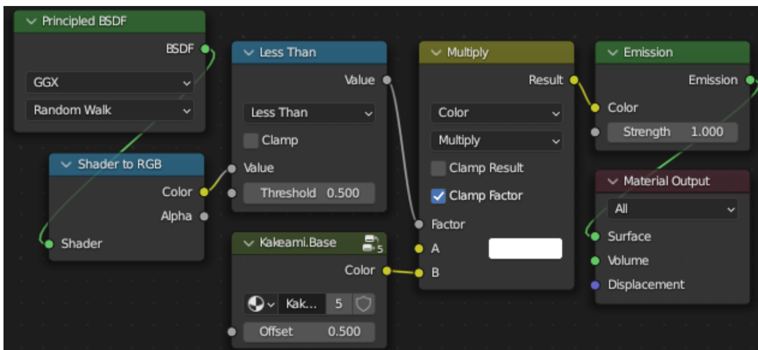
先ほど作ったハッチングのノードをここにつないでみます。

ボロノイの色ソケットを利用すると、セルごとのランダムな値が取れるので、
それを使って回転ができるように少し拡張をします。
ランダムの値は[0-1]の範囲なので、3.14倍して半周回転できるようにします。
（ラジアンが単位ですね）

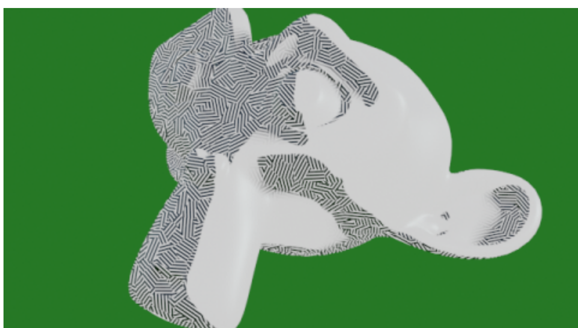
セルごとに向きが違うハッチングをすることができました。

これを影のところにかかるようにすれば、漫画のカキアミに似た効果が作れそうです。
カメラから近い位置と遠い位置にハッチングしたいオブジェクトがある場合は、
ハッチングのスケールを距離に依存するようにするか、一定にするかは一長一短でなかなか難しいところです。
作風に応じて調整をします。

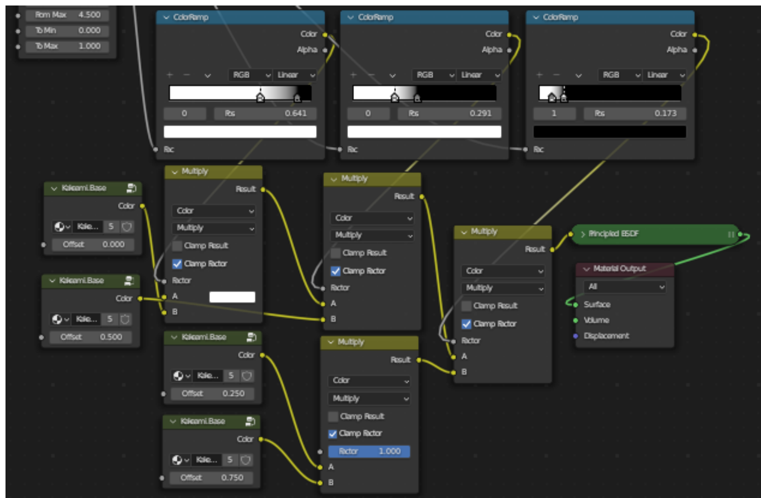
Shader to RGB ノードを利用して、影のところにだけ効果を付けてみましょう。



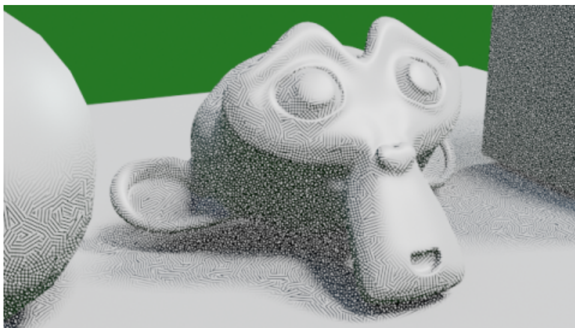
先ほどのカケアミの効果全体をグループ化して、
影の部分にだけ掛け算で効果がかかるようにしています。



カケアミ状の効果が得られました。
漫画では、こうしたカケアミの斜線を重ねていくことを２カケ３カケと呼ぶそうです。
再現できるように、斜線の角度をパラメーターで変えられるようにします。
傾きにオフセット値を入れられるようにノードを拡張しました。



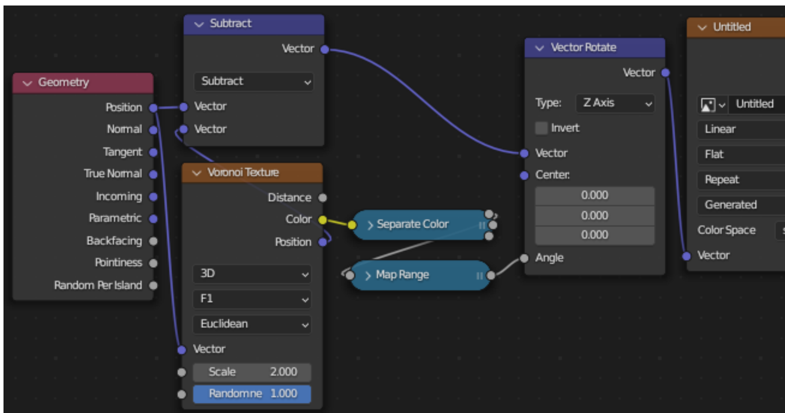
陰影の濃さに応じて、角度の違う複数のカケアミを重ねるようにしてみます。
Color Ramp と色の掛け合わせを複数組み合わせています。



カケアミのサンプルファイルとして、01_HATCHING/03_Kakeami.blend を同封してあります。

ボロノイ分割によるタイリングのなじませ

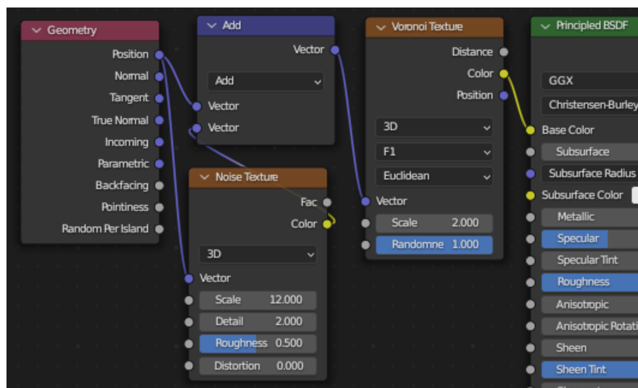
先ほどは、カケアミを表現するために、カメラから見た座標に変換してボロノイ分割を使いました。
普通に空間座標で考えれば、地面の模様などにテクスチャをタイルのように敷き詰めるのと同じことができるわけです。



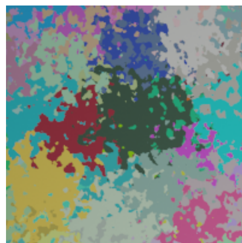
先ほどと同じ理屈で、ボロノイのセルごとに座標を貼ってランダムに回転させた座標を作ります。
これを使ってテクスチャを貼れば、

ランダムな敷き詰めが行えます。

しかし、セルの境界は直線状です。
テクスチャによっては境界があまり目立たないのですが、
色見や風合いの違う部分があるテクスチャだと、直線状の切れ目が目立つときがあります。

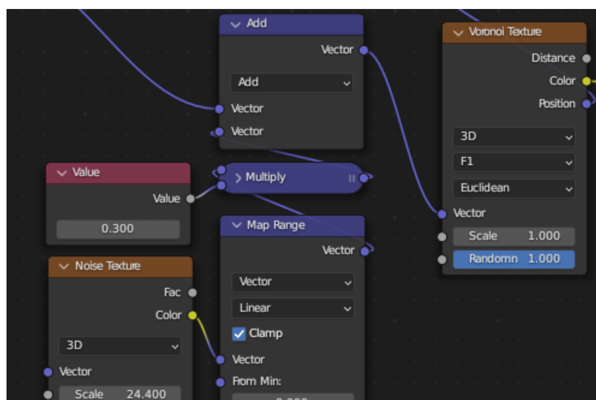


こういう時は、ボロノイテクスチャに送る座標を、ノイズなどを加えて乱すということが有効なことがあります。
例えば、ノイズのスケールを適度に調整して、ボロノイの色情報を表示をしてみると、



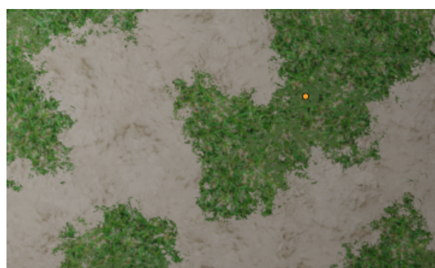
このように境界が乱れます。

これは「ある座標がどのセルに所属しているか？」という分け方に相当していますので、
先ほどの計算で単純なボロノイの代わりに、乱れた座標系でのボロノイを使うようにすれば
境界の乱れたボロノイ分割をすることができます。



ただし、実際のノードの組み方は、乱れの大きさを調整できるように係数をかけ合わせたり、
ノイズの色[0, 1]の範囲だと足し合わせの時に不便なので、[-1, +1]の範囲に焼き直したり、
再度テクスチャとして利用するために逆に[-1, +1] の範囲を[0,+1]の範囲に戻りたりなど、
やや複雑なノード構成になってしまいます。

直線だった境界が乱れて、境目が目立たなくなります。
実際には直線ではなくなった分目立たなくなっただけで、境界そのものは存在するのですが、
自然物のテクスチャを敷き詰めるような場合にはだいぶなじむようになります。



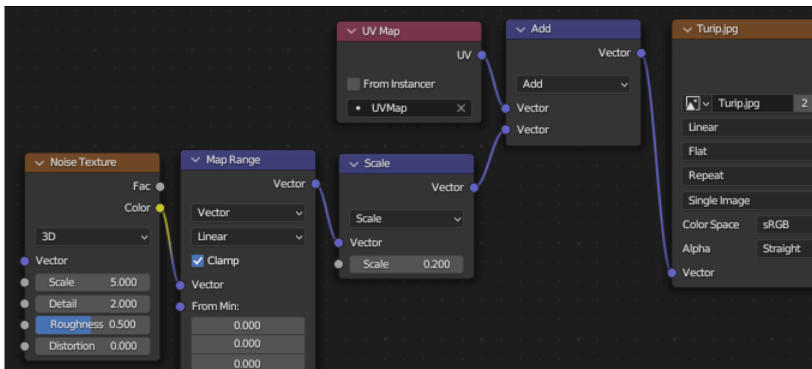
極端な場合は、セルごとに違ったテクスチャを使っても、
境界が乱れているので（離れて見る分には）ある程度自然に見えるようにもできます。

このサンプルでは、後ほど少し解説する、UDIMという仕組みを使って、砂地と芝生の２種類のテクスチャを混ぜ合わせてみました。

この例は、02_TILING/01_TilingPattern.blend というサンプルとして同封しました。

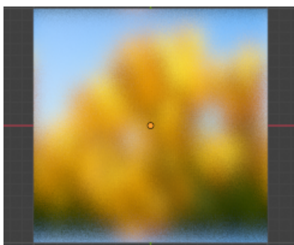
ノイズとブラー

テクスチャを利用するときに「テクスチャに加工をして使いたいな」という場合があります。
色味の加工などは、シェーダーの中で色を変更するノードを加えれば良いのですが、例えば「ブラーをかけてぼけた画像を使いたい」というような場合にはノードでそれを（素直に）実現する方法はありません。
ただ、先ほどの例で使った「境界を乱すために座標にノイズを加える」という手法を極端に拡張すると、ブラーのような効果を得ることが出来ます。

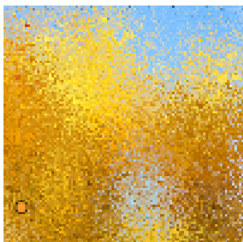


表示される座標をランダムに変化させます。
本当は、ガウス分布のランダム…などのノードがあると良いのですが、特にないのでノイズテクスチャで代用をします。
Map Range は縦長なノードなので画面から切れていますが、テクスチャの色は0-1の範囲なので、-1から+1の範囲になるように変換をしています。
Scale でランダムの度合いを調整して、元のUVマップに足し合わせて利用をします。

当然、ノイズのスケールと歪みの度合いに応じてぐにゃぐにゃと変形したテクスチャが表示されます。
この時に、ノイズのスケールをやみくもに大きな値、例えば1万とか２万といった数にすると…

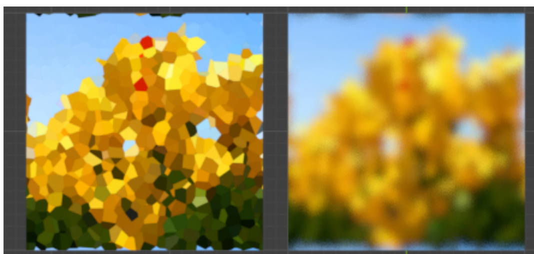


ブラーがかかった表示にすることが出来ます。



表示のサンプル数を１にすると仕組みが見えてきます。
余りにノイズが細かいので、一見ランダムにドットごとの表示位置がズレたように見えます。
表示サンプル数を１よりも上げると、１ドット以下座標をずらしたレンダリングを、それらを重ね合わせて平均化することで画像の品質が上がります。

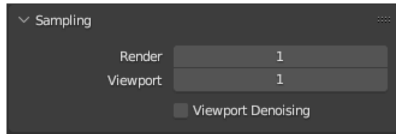
この時、ノイズのスケールがあまりに細かいので、全然違う位置のテクスチャの色を表示して平均化をしていくることになり、サンプル数を充分に上げれば、周りのいろいろな位置の色情報の混ぜ合わせ…すなわちブラーをかけたような結果になるのです。



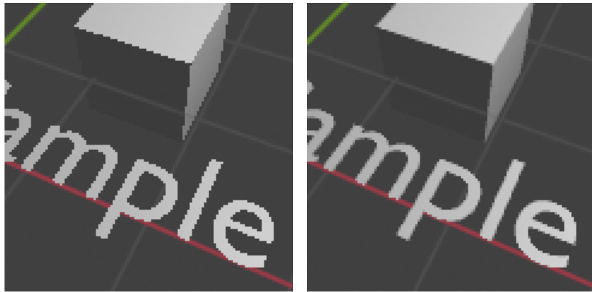
先ほどの、ボロノイを使って分割する方法の亜種として、セルごとにセルの中心の色を使ったステンドグラス調の処理をしてみました。
これも、ボロノイ分割に使う座標に、さらに非常に細かいノイズを載せることで、
ぼやけたステンドグラス風味になります。
これは、ボロノイやノイズのスケールを調整することで、簡単な点描風味にもすることが出来ます。

この例は、02_TILING/02_BlurAndNoise.blend というサンプルとして同封しました。

Eevee のサンプル数について



Eevee の設定で、普段は目立たないけれども重要な設定に、サンプリングがあります。
サンプル数、と呼んだ方が意味が伝わりやすいでしょうか。
一度のレンダリングに、何回サンプルを取るかの指定をします。



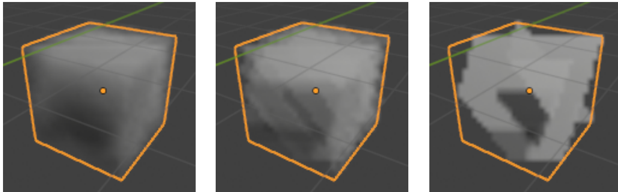
実は、Eeveeの基本のレンダリング方法はピクセルごとに描画する色を決めて描画…というプロセスなので、
拡大して見ると本当は左の図のようにドットが見えるジャギジャギの結果になります。

こうしたポリゴンの境界付近だと、ほんの少しカメラをずらせばポリゴンからはみ出るピクセルや、そうではないピクセルがあります。
そこで、1ピクセル以下のサブピクセルずらした結果を沢山レンダリングして、結果を重ね合わせて平均化すると、
もっと解像度が高い画像を元に作成した画像の様に、滑らかになるのです。

上の例では、サンプル数を8にするとまぁまぁ滑らかになってきました。
デフォルトの Eevee のサンプル数は、プレビューで 16 レンダリング時に64 なので、普通は充分なサンプル数です。

この少しずれた画像を平均化するという仕組みを逆手にとったのが、先ほどの非常に細かいノイズを利用したブラーのテクニックになります。

この「ほんの少し違う画像を何度もレンダリングして平均化」によって画質を上げるという仕組み、なんだか回りくどいようにも感じますが、
これを利用することで Eevee の半透明の表現(Alpha Hash)やボリューム表現などが成り立っているので、頭の片隅にでも入れておくと良いかと思います。

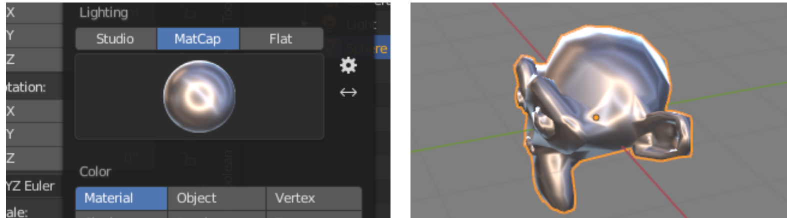


Eevee でボリュームをレンダリングして、サンプル数を下げると、実はボリューム表示の正体はカメラの方を向いた断面の板であることが分かります。
サンプル数が大きい場合には、板の位置をずらしたレンダリングを複数回行って平均化をしています。
充分にサンプル数が大きければ、ボリュームが滑らかに見えてくるという仕組みです。
この表示の仕組みから、雲や霧などのボリューム表示の品質にも、サンプル数が影響してくることが分かります。

Mat Cap あれこれ

Mat Cap は、陰影の表現を予め用意されたテクスチャ利用して行うものです。
予め凝ったマテリアルや、ライティングでレンダリングしたものを用意して、その結果だけを利用するような使い方をします。
もしくは、粗い筆のタッチのような、絵として用意したものを使うこともあります。

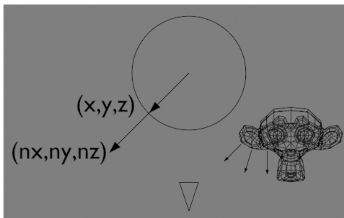
たとえば、blenderでデフォルトで用意されたものを例にすると、



スザンヌを表示するとこのようになります。
通常は、球をレンダリングしたようなテクスチャとして用意して、別の形にも適用します。

カメラ座標を使った Mat Cap

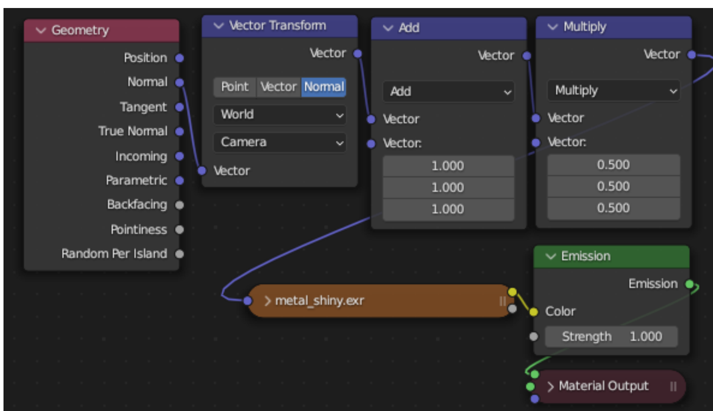
カメラから見たカメラ座標を使って、Mat Cap を適用してみます。



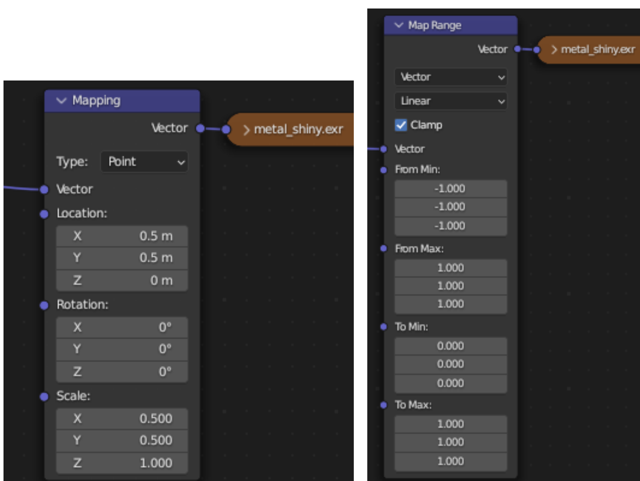
法線の成分から、テクスチャのUVに変換します。
原点にある半径１の球の場合は、法線のベクトル(nx,ny,nz)と、表面上の位置(x,y,z)が同じになります。
そのため、 z 方向を奥行きと考えた場合、
(nx, ny)の成分がそのまま球の(x,y)になります。

つまりテクスチャとして球の絵があれば、それをそのままUVとして使えることになります。
ただし、範囲が[-1, +1] なので、UVとして使うために[0, +1]の範囲に変換することだけ必要になります。

bldender 本体のフォルダの datafiles/studiolights/matcap/ の中に matcap 用画像があるので、例として metal_shiny.exr というファイルを使ってみます。



まず、非常に単純な実装の Mat Cap 用のノードです。
法線をカメラ座標にベクトル変換(Vector Transform)をした上で、
そのままでは[-1, +1]の範囲なので、1を足して0.5倍することで [0,+1]の範囲にします。

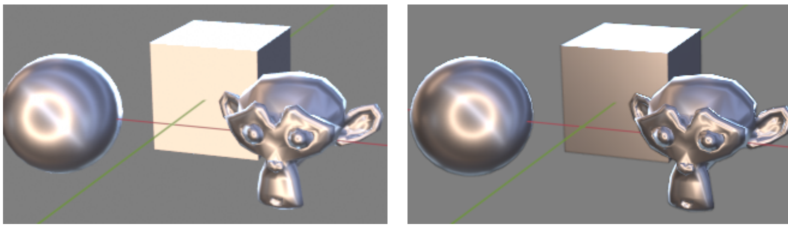


後半の座標変換は部分は、
Vector - Mapping (マッピング)のノードで１つのノードで処理することもできます。
この場合は、0.5倍してから0.5を足す、という処理の順番です。

もしくは、Map Raneg(範囲マッピング)を Vector で行って、
[-1, +1]の範囲を [0, +1] の範囲にすると直接指定もできます。

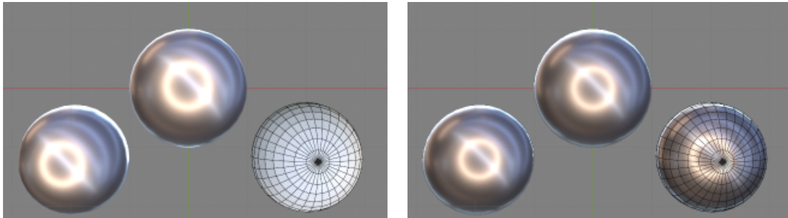
(このノードは、ベクトルを多数入力するので縦長になるのが弱点ですね)

この状態のサンプルを 03_MATCAP/01_MatCapSimple.blend として同封します。
これで、多くの場合は充分なのですが、実際に画面に出してみると、blender の MatCap表示とは違いがあることに気が付きます。



左が今回の実装、右が blender の MatCap 表示です。
立方体の面のように平らな面は完全に単色になってしまいます。
（この実装では、面の向き（法線）が同じところは全て同じ色になってしまいますので当然ですが…）

また、画面の端にオブジェクトを置くとやや歪んで見えることに気が付きます。



ワイヤーフレームで見た球を見ればわかりやすいのですが、パースが効いている分画面端では極部分が球の中心からずれて見えます。
実装した Mat Cap でもそれを忠実に反映していることになります。
これを補正するには、「カメラの向いている方向」を基準にした座標ではなくて、「カメラと表示している点との位置関係」を基準にした座標で考える必要があります。
（この２つは画面中央では同一ですが、視野の端に行くと違いがでてきます）

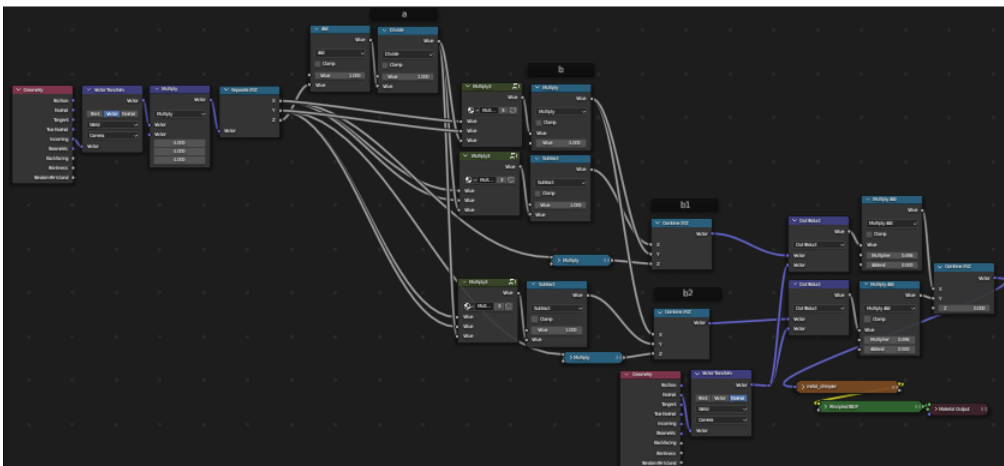
blender の MatCap 実装の再現

blender の Mat Cap がどのように補正をしているか、見た目の違いから調べるのは大変そうです。
そこで、直接ソースコードの中の Mat Cap の部分を見てみます。 ソースコード中の Mat Cap 表示部分は次のようになっていました。

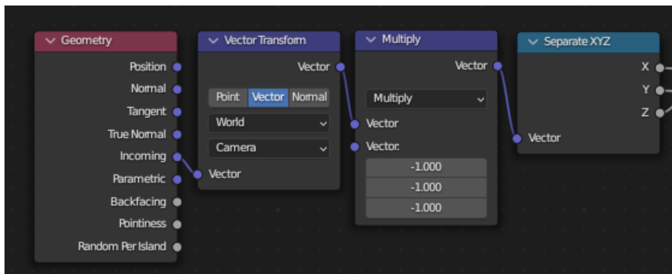
source/draw/engines/workbench/shaders/workbench_matcap_lib.glsl
(2.92頃の情報です。3.4時点で既に改修がはいり、Mat Cap の表示も当時とやや変わっているようです)

```
vec2 matcap_uv_compute(vec3 I, vec3 N, bool flipped)
{
  /* Quick creation of an orthonormal basis */
  float a = 1.0 / (1.0 + I.z);
  float b = -I.x * I.y * a;
  vec3 b1 = vec3(1.0 - I.x * I.x * a, b, -I.x);
  vec3 b2 = vec3(b, 1.0 - I.y * I.y * a, -I.y);
  vec2 matcap_uv = vec2(dot(b1, N), dot(b2, N));
  if (flipped) {
    matcap_uv.x = -matcap_uv.x;
  }
  return matcap_uv * 0.496 + 0.5;
}
```

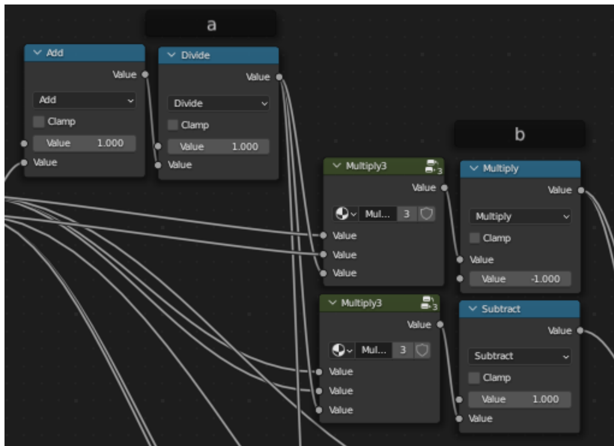
これを再現するノードは次のようになりました。
成分分けをしてかけたり引いたりの演算が多いので、コードの割には長いですね。



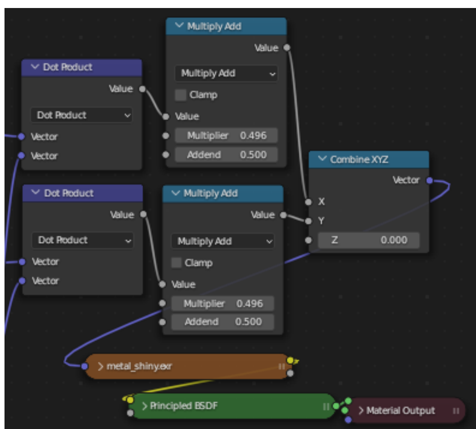
ここでは、一部だけ説明をして、
ノード全体は 03_MATCAP/02_MatCapBasicSample.blend にサンプルとして入れておきます。



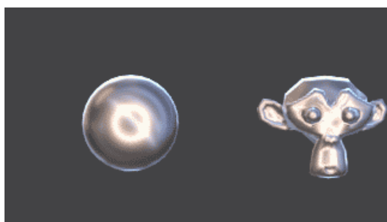
ソースコードの引数にある I ベクトル(vec3 I)を最初に用意します。
これは、Incoming ベクトルと言われ、処理している点から見た相対的なカメラの方向を示すベクトルのことです。
Geometry - Incoming を Vector Transform を使いカメラ座標系に変換しました。
向きの定義が逆のようなので、-1倍をしています。



glsl の式に従ってノードをつないでいきます。
掛け算の部分は 3 つの数値をかけ合わせる Multiply3 というグループを使ってノード数が少なくなるようにしています。
分かりやすくなるように、a,b といった変数名のラベルを配置しました。



同様にして、b1, b2 ベクトルを作成し、内積を取って、0.496倍して0.5を足して…
とシェーダーの式を再現するようにしています。
導出したUVを使ってテクスチャをオブジェクトの表面に表示します。



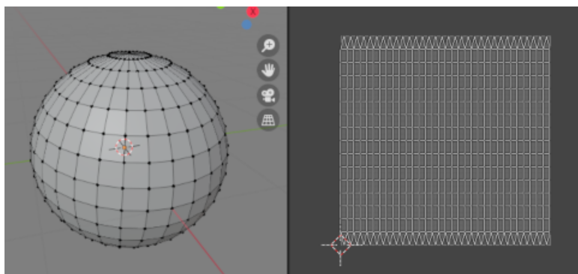
blender の Mat Cap と同等の見え方をするノードができました。
(動画)MatCapSample01.gif

このタイプの Mat Cap は、カメラ座標に乗っているので、カメラが動いても見え方は変わりません。
カメラと光源が一体化しているような状態ですね。

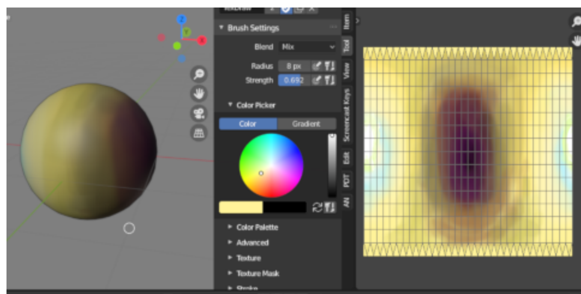
次に、少し別のタイプの Mat Cap を見てみます。

ワールド座標を使った Mat Cap

カメラの向きに、ワールド座標系を使って考えてみます。
まず、質感の元になる球の表現ですが、カメラが裏に回り込めるようになりますから、裏から見た情報も必要になります。
そこで、球のUVを利用してみることにしましょう。

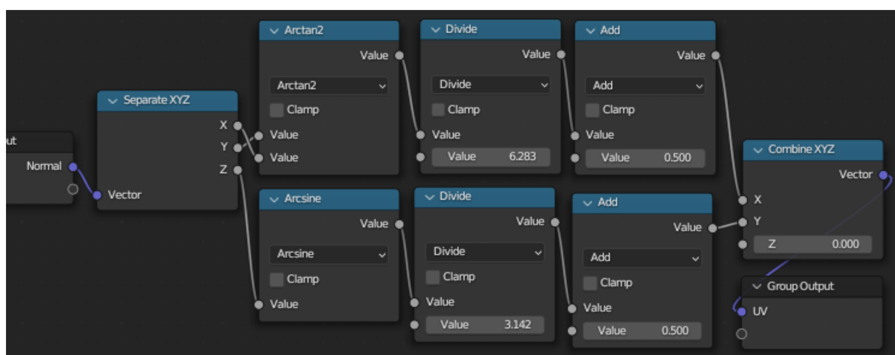


UV球を用意します。この上に質感をテクスチャペイント機能で描きこんでゆきます。

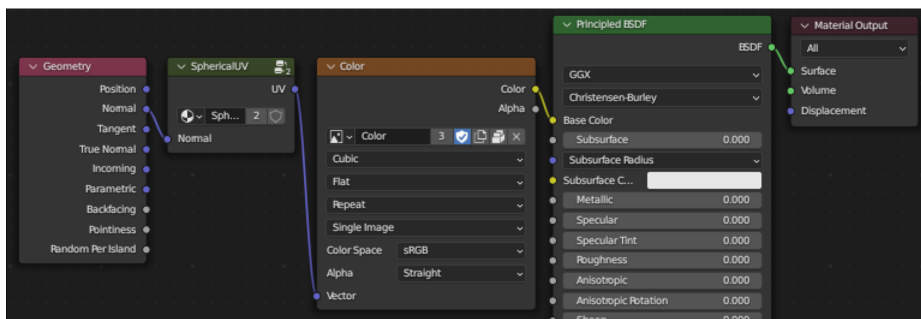


光源の向き、影の向きは、完全にワールド座標に固定なら好きにすればよいのですが、
光源の向きを動かすことを考えると、分かりやすい軸に合わせておきます。
ここでは -X 方向から光が当たって、+X 方向に影ができるようにしてみました。
これは、逆にしても少しノードの組み方が変化するだけなので、好きな方にすればO.K.です。

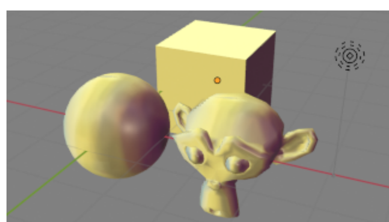
次に、ノード構成を考えます。
全体のノードの規模が大きくなることが予想できますから、グループ機能をうまく使ってまとめます。
まずは法線方向から UV に変換をするノードを考えます。



この変換は、単純な球座標への変換ですね。
Arctan2, Arcsine を使って、θφ成分にします…が、それだと範囲が[-pi, +pi]などになってしまいますから、pi や 2piで割ったりして[0,1]の範囲に収まるように調整します。
このノードグループを、SphericalUV と名付けました。



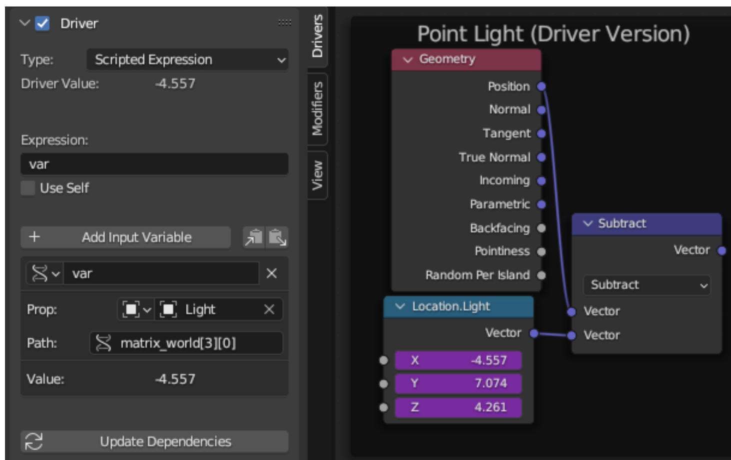
法線ベクトルをつなぎ、UVに変換してテクスチャノードにつなげば、球に描きこんだ質感のような風合いをコピーできます。



スザンヌが球の質感と同じような質感になりました。
ただ、光源の方向ガン無視ですし、立方体のような面は単色になってしまいます。
（平面では法線の向きが一定ですから、この仕組みでは必ず単色です）

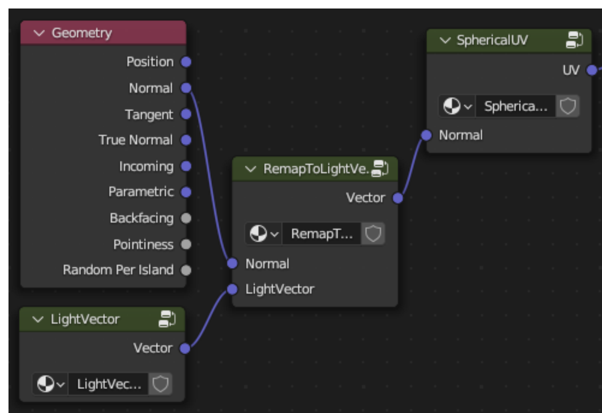
順に改善していきます。

まず、光源の方向の情報を得るノードを作ります。
ドライバーを使って指定した光源の位置を反映するようにしてみます。
（ドライバーを使わない場合の方法は、このセクションの最後に解説します）



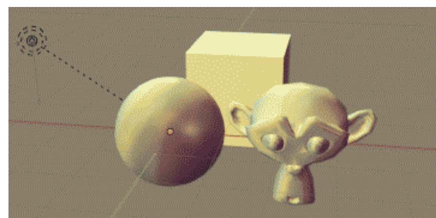
Light の位置(X座標)を得る Path は、location[0]です。
同様に、Y座標はlocation[1], Z座標はlocation[2] です。
位置ベクトルとの差を取れば、光源の方向の情報が得られるノードになります。
これもグループ化をして、LightVectorという名前にしました。

> ※ もし光源が親子関係を持っていたりすると、location ではそれを反映した位置を得られません。
> （親子関係の構築前の段階の位置情報になります）
> そういう場合は、matrix_world[3][0], matrix_world[3][1], matrix_world[3][2] を使います。



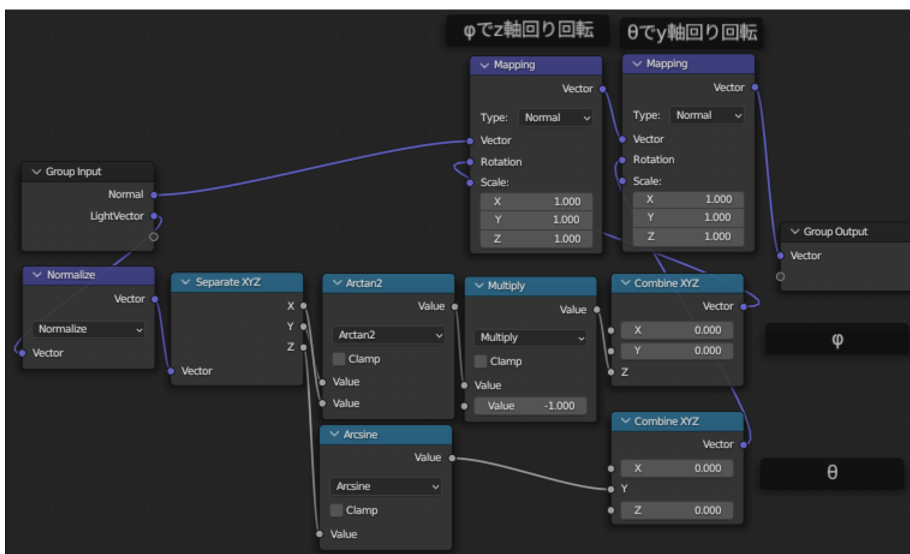
法線ベクトルを、ワールド座標上で評価するのではなく、
Light Vector の方向に合わせた座標で評価すれば、光源の向きを反映させることができます。

中身は後で見てみることにして、ノード間にそのような座標変換のノードグループ(RemapToLightVector)を挟んでみます。



光源の向きを反映させることができました。
(動画)UVSphere01.gif

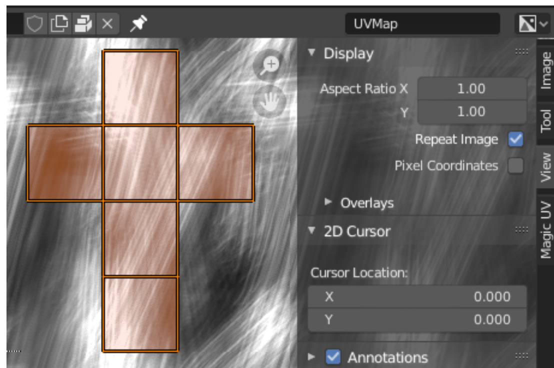
では、RemapToLightVector の中身を見てみます。



基本的には LightVector の成分から Arctan2, Arcsine を使って、極座標形式にしていることになります。
光源方向から見た場合の θとφを求めて、法線の方向を回転させているわけです。

ブラシストロークのような効果の Mat Cap

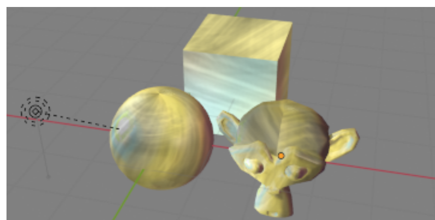
元から手塗りのテクスチャを使っているので、筆の跡のような効果は乗っているのですが…
さらにブラシストロークのような効果を重ねて乗せることをしてみます。
まず、ストローク情報の元になるテクスチャを用意します。



プロシージャルに作ることも理論的には可能でしょうが…ここは、テクスチャを用意しておくことにしましょう。

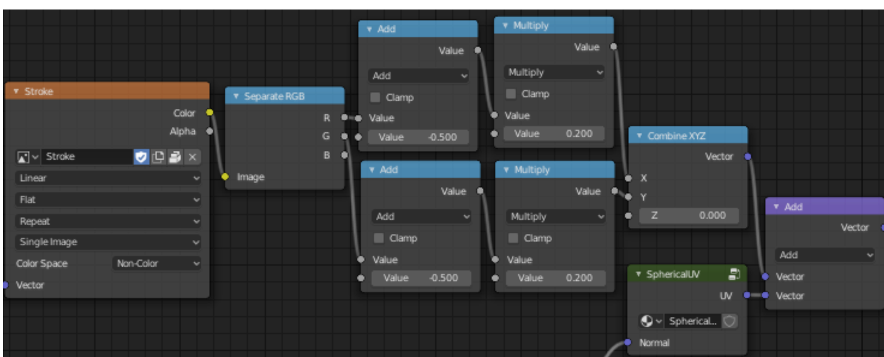
用意するテクスチャは、繰り返しのタイル状になっている方が切れ目が発生しないので良いです。
blender上のイメージエディタで描く場合には、View と Tool にそれぞれ繰り返しのオプションが（表示と描きこみ別個に）があります。
これらを、必要に応じて有効にして描きこめば、繰り返し状のタイルにすることができます。

先ほどの Mat Cap 用に計算した色に、単純に色を重ね合わせてみると、



まぁ重ね合わせだねという表面になりました。

ここで、本当のブラシストロークであれば、色が筆の後に沿って伸びるということなるはずです。
そこで、重ね合わせではなくて、UVの座標を歪めてみることにしてみます。



ストローク用のテクスチャの成分を分解して、元の Mat Cap 用のUVに足し合わせています。
（今回はグレーで作ったので、R,Gの違いには意味は無いですが）
これで、ストローク用のテクスチャの成分に従って、Mat Cap の色が引っ張られる効果がうまれます。

ブラシストロークが乗ったように、色が筆のタッチの方向に伸びた表面にすることができました。
ややリッチな絵にするために、地面を置いて影や光源の設定を付けています。
(動画)UVSphere02.gif

今回は色の設定を Principled BSDF の Base Color につないでいます。
そのため、陰影や影の処理も、テクスチャによる色の変化に重ねる形で行われています。

逆に言えば手描きで陰影を完全に処理するのであれば、Emission 成分につなぎ、Base Color を低めにした方がダイレクトに色が反映されて良いかもしれません。
ただ、Base Color を黒にしてしまうと今度は影の表示がされないので、欲しい絵との兼ね合いになります。

このサンプルは、03_MATCAP/MatCapUVSample.blend にサンプルとして入れておきます。