

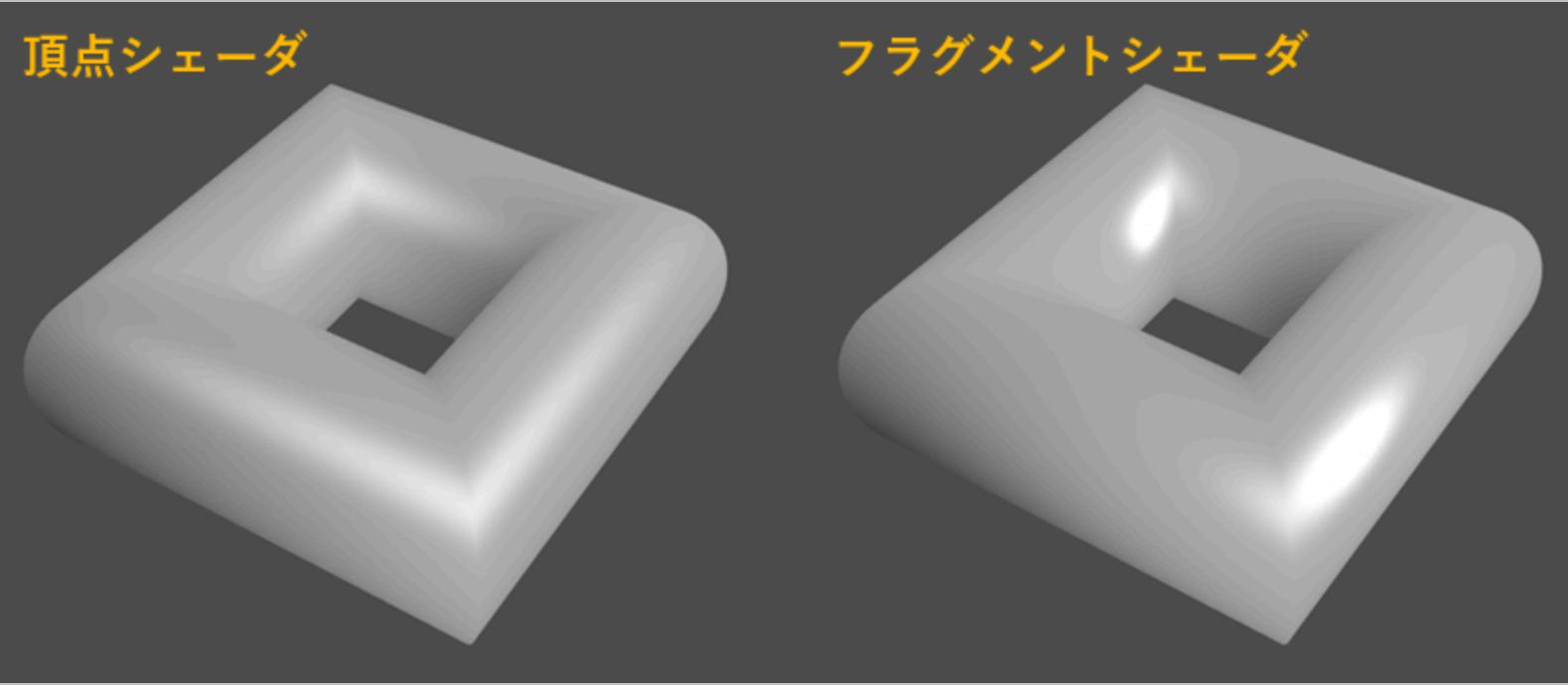
# テクスチャとブレンド

## テクスチャ関連テクニック

# **前回の課題について補足**

前回の課題（フラグメントシェーダで照明効果を実装する）の参考例として、第七回のサンプルと一緒に、実装例を同梱しました。

実際に描画される様子を見ると、頂点シェーダで処理する場合とフラグメントシェーダで処理する場合とで、どのような違いが出るかわかりやすいと思いますので参考にしてみてください。



“頂点シェーダでは、あくまでも頂点単位で計算が行われた上で、その結果が各頂点間で補間される。フラグメントシェーダではピクセル単位で計算が行われるためより厳密な結果となる。

# 今回のお題

前回はネイティブな WebGL のみを利用して、平行光源による陰影付けをシェーダで実装しました。

シェーダは JavaScript などとはまた違った感性を求められる部分があるので、無理せず少しずつ慣れていきましょう。

さて今回ですが、テクスチャ関連の技術とブレンディングについて、詳しく見ていきます。

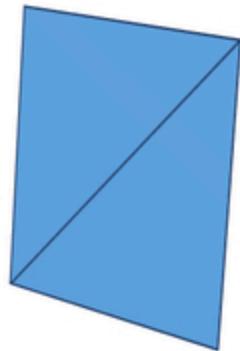
一度 three.js でも扱っている内容ですが、より詳細にネイティブな WebGL API の場合はどのように設定するのかを通じて理解を深めていきましょう。

テクスチャ

今回はまず、CG に欠かせない概念である テクスチャ について考えていきます。

テクスチャとは、画像などのいわゆるビットマップイメージを WebGL で利用するための概念で、three.js でテクスチャを使ったサンプルが以前にもあったかと思います。

もちろんネイティブな WebGL でもテクスチャを使うことができます



画像などのビットマップを.....

頂点（ポリゴンなどを含む）に貼り付ける！

“ビットマップを使って複雑な模様を描画結果に反映させられる”

three.js でテクスチャを使ったときには意識しなかったかもしれません  
が、テクスチャに関する処理は、厳密には「貼り付ける・貼り付け  
ない」といったように二者択一で状態がオン・オフされるようなものでは  
ありません。

ではどうやってテクスチャを使うのかというと「シェーダ内でテクスチ  
ヤからデータを読み出し、貼り付ける（マッピングする）処理」を記述  
します。

手順としてはまず、生成したばかりの 空のテクスチャを VBO と同じようにバインド し、なんらかの画素データを流し込みます。

流し込まれた画素のデータは、GPU 上のメモリ空間へ転送されます。

GPU のメモリ上にあるテクスチャの画素データは、シェーダ内でそれを取り出すようなコードを記述することで参照できます。

テクスチャから取り出した画素の色情報は、主にフラグメントシェーダから出力する色などとして使うことができます。

“テクスチャから読み出したデータを色として使わなければならないという仕様があるわけではないので、読み出したデータをどう使うかは自由”

またこれは地味に重要なポイントなのですが、一度テクスチャにデータを格納することさえできてしまえば、元データがどんなフォーマットのものであれ、そこから先はまったく同じ手順でテクスチャを利用することができきます。

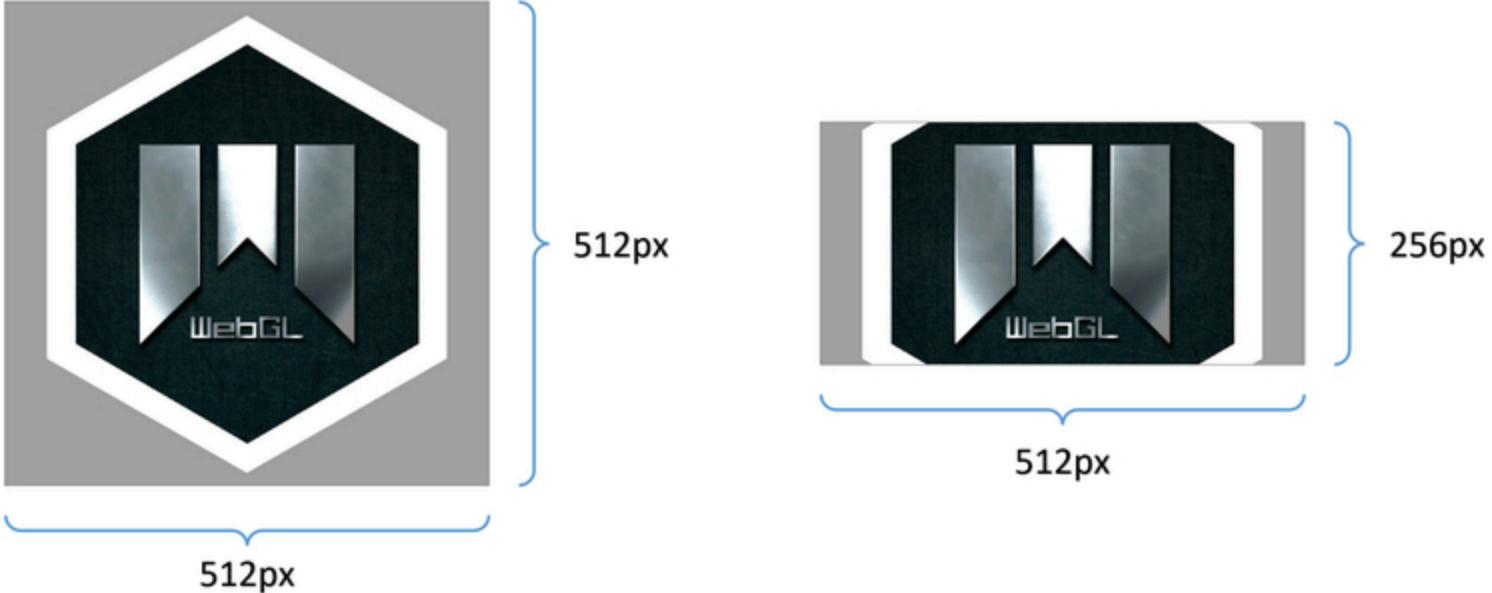
要は元データが JPEG でも PNG でも、あるいはプロシージャルに Canvas2D などを用いて生成したビットマップであっても、テクスチャオブジェクトの使い方さえわかっていれば全く同じように扱うことができるということです。

一度テクスチャというインターフェースへと抽象化することで、データソースがなんであれワークフローは統一されたものになる

ただし注意点として、テクスチャの元データとなる画像は、画像の一辺のピクセル数（サイズ）を 2 の累乗 に合わせることが推奨されています。

それ以外の大きさでも大抵の場合は問題なく表示されますが、これは仕様上で非推奨となっています。確実に使える状況を作るためにも多少面倒でもこのルールについては基本的に守るようにしたほうがいいでしょう。

“ 2 の累乗とは 128px とか 256px、512px などですね ”



元素素材の縦横それぞれの一辺の長さが2の累乗になるように事前に調整しておく

縦横の比率は1：1でなくても大丈夫

ちなみに、テクスチャ用の画像データの大きさ（一辺の長さ）の上限値も実は存在するのですが、これは端末によって上限値が異なります。

大抵の場合、モバイル端末などの場合は PC よりもこの上限値が低くなる傾向があるので、もしターゲットとなる端末が固定されている場合には、事前に調べるようとしたほうがいいかもしれません。もちろん、PC でも GPU の性能によって最大値は変わります。

“もちろん一般的なウェブサイト上のリソースと同様に、リソースが軽いに越したことはない”

このテクスチャ用リソースの一辺の長さの最大値のように、WebGL コンテキストの性能を調べるには以下のように `gl.getParameter` を利用します。引数にどのような指定を行ったのかによって、得られる結果が変わります。

```
const maxTextureSize = gl.getParameter(gl.MAX_TEXTURE_SIZE);
```

参考 : [WebGL MAX parameters test](#)

# ここまでまとめ

- テクスチャとは画素データの入れ物
- JPEG でも PNG でも、一度テクスチャにしてしまえばその後の手順は同じフローで処理できる
- テクスチャの元データは原則 2 の累乗サイズに揃えることが好ましい
- テクスチャの当該環境での最大サイズは事前にチェックできる

`webgl.js` に定義されている `WebGLUtility.createTexture` というメソッドを見てみると、その内部で `gl.bindTexture` というメソッドを呼び出しているのがわかると思います。

この「バインドする」という処理は、VBO などのバッファを扱ったときにも出てきました。テクスチャは、一種の画素データを格納したバッファのようなものなので、VBO などと同様にバインドするという手順が必要です。

テクスチャを初期化する際はまず WebGL コンテキストに対して「テクスチャをバインド」した上で.....

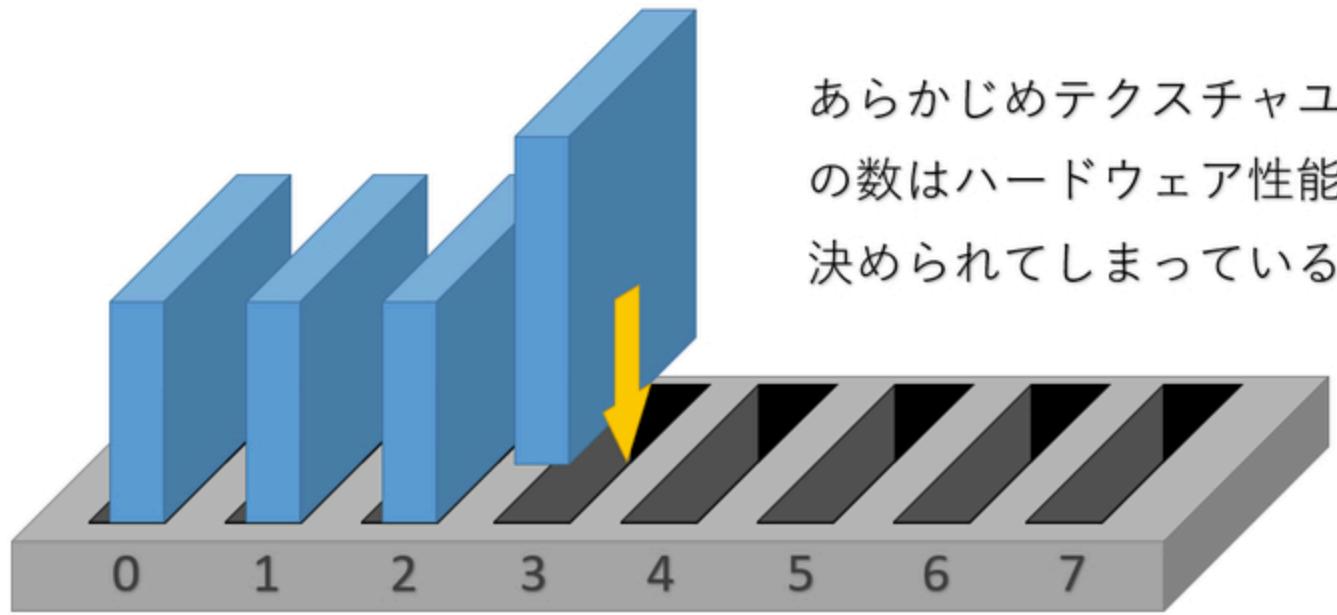
画像などの元となるデータをテクスチャに対して割り当てる事ができるメソッド（ `gl.texImage2D` ）を呼び出し、その時点でバインドされていたテクスチャに対してデータが割り当てます。

ただし、このように「バインドしているものに対してのみ効果が及ぶ」という考え方を前提にしてしまうと「複数のテクスチャを同時に使いたい場合に困ったことになるのでは？」と感じる方もいるかもしれません。

これに対応するために、テクスチャには テクスチャユニット という概念があります。

テクスチャユニットとは、GPU 上に設けられているテクスチャをバインドするためのインターフェースです。

いわゆるカートリッジの差し込み口のようなイメージで考えると想像しやすいかと思いますが、その差し込み口の数はハードウェアの性能によって変動します。



あらかじめテクスチャユニット  
の数はハードウェア性能により  
決められてしまっている

“どのような GPU でも、最低 8 つはユニットが存在しなければならないと仕様で決まっているので、8 つまでは安心して使える

テクスチャユニットは0から始まる連番になっていて、特になにも指定を行っていない場合の既定値は0に設定されています。つまり「アクティブなユニットの既定値は0番」です。

そして、ここからがちょっと紛らわしいのですが バインド操作の対象となるのは常に、アクティブなユニットのみ になります。

たとえば 0 番目ではなく 1 番目のユニットにテクスチャをバインドしたければ「これから 1 番ユニットを使いたいので、 1 番をまずアクティブにする」という処理を先に行っておきます。

その上で `gl.bindTexture` でテクスチャをバインドすると、その時点でアクティブなユニットに対してテクスチャがバインドされます。

ただし、WebGL のテクスチャを使った処理に慣れるまでは あまり深く考えずに規定値である 0 番目のユニットだけを使う ようにしましょう。

経験的に、0 番目のような若い番号のユニットから順番に使っていかないとエラーになることがあったりするためです。

# ここまでまとめ

- テクスチャユニットは GPU 上にあるテクスチャが格納される場所
- テクスチャの最大同時バインド可能数 = テクスチャユニット数
- アクティブなユニットに対してバインド処理が行われる
- ユニットを切り替えてバインドすれば同時に複数のテクスチャをバインドすることもできるが、慣れるまでは 0 番のユニットを前提にロジックを考える癖をつけたほうがよい

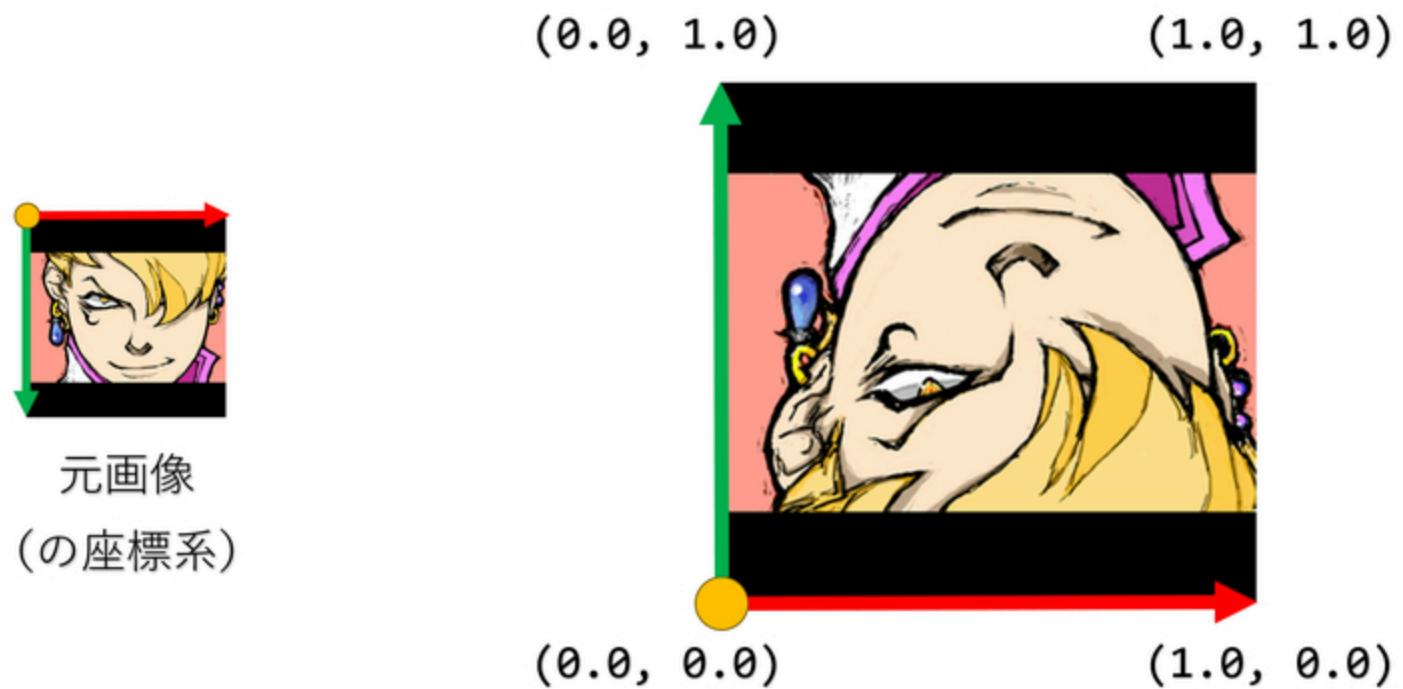
# テクスチャ座標

さて、だいぶ説明することが多いですが、テクスチャを使ったレンダリングを行うためには他にも準備しなければならないことがいくつかあります。

まず最初に変更を加えるべきは 頂点属性の追加 です。ライティングのために法線を追加したのと同じように、テクスチャマッピングのために テクスチャ座標 と呼ばれる新たな情報を頂点に追加します。

テクスチャ座標とは、その頂点に テクスチャのどの部分を貼り付けるか の指標となる座標です。

これは 3D モデリングなどの世界ではよく UV (ゆーぶい) と呼ばれるやつです。OpenGL や GLSL では、S と T を使ってこのテクスチャ座標を表します。



OpenGL や WebGL ではテクスチャの座標系における原点が左下となる。それをそのまま図解してしまうと画像座標系と上下が反転しているように見えるが、原点位置がそろうように読み込まれるので勝手に逆さまになったりはしない。

“ テクスチャ座標の範囲は常に 0.0 から 1.0 で指定

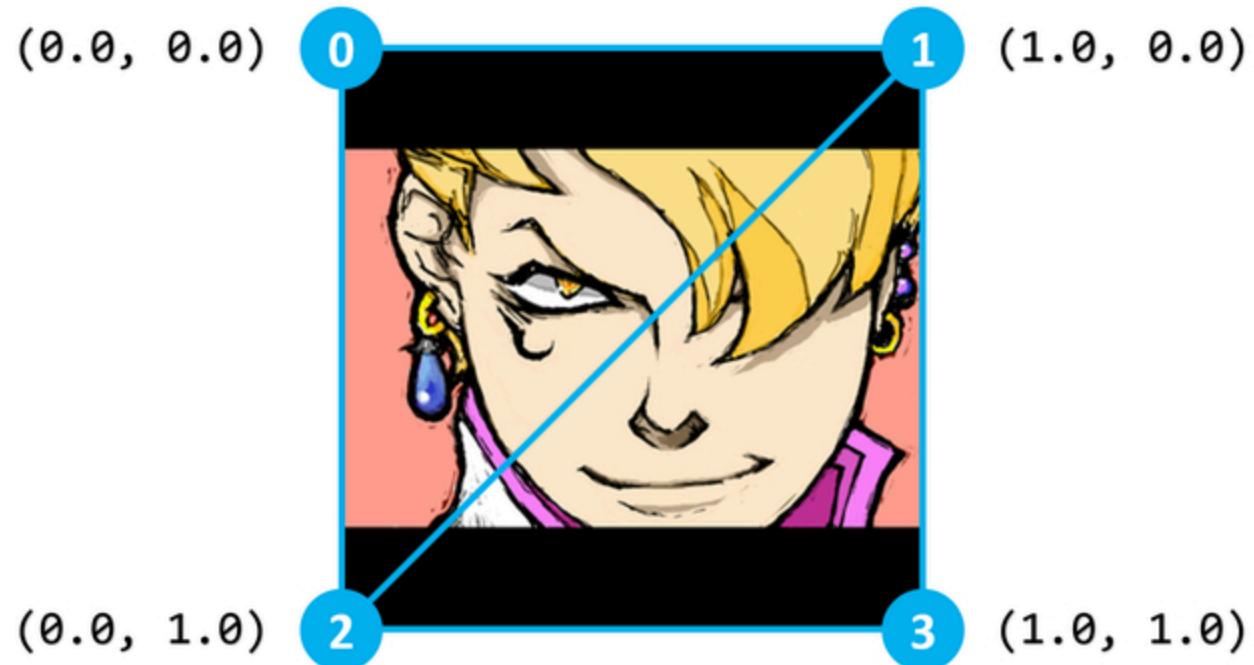
図を見るとわかるとおり、テクスチャ座標は二次元の座標系なので、シェーダ内では普通 `vec2` として定義します。

`attribute` 変数としてシェーダに頂点属性を追加し、JavaScript 側ではこれに割り当てるための新しい VBO を作ります。

以下のようにすることで、すべての頂点がテクスチャ座標という新しい頂点属性を持った状態になります。

```
this.planeGeometry = WebGLGeometry.plane(size, size, color);

this.planeVBO = [
    WebGLUtility.createVBO(this.gl, this.planeGeometry.position),
    WebGLUtility.createVBO(this.gl, this.planeGeometry.normal),
    WebGLUtility.createVBO(this.gl, this.planeGeometry.color),
    WebGLUtility.createVBO(this.gl, this.planeGeometry.texCoord), // テクスチャ座標
];
```



頂点 4 つで板ポリゴンを作っている場合、そこにぴったりと画像を  
貼り付けるにはテクスチャ座標（の頂点属性）を上記のように設定する

テクスチャユニット番号をシェーダに送る

テクスチャのユニット番号は、シェーダに対して「どのテクスチャを使えばよいか」を指示する際にも利用します。

ちょっと違った言い方をすると「uniform 変数としてユニット番号を指定しておくことで、シェーダ内でどのテクスチャが使われるかが決まる」ということです。

レンダリング時に使いたいテクスチャがバインドされているユニット番号を  
シェーダに送る！

JavaScript 側から送ることになるテクスチャユニット番号は、それを受け取る GLSL 側では、テクスチャを扱う特殊なデータ型（`sampler2D` 型）が受け口になります。

```
uniform sampler2D textureUnit; // テクスチャ @@@
```

つまり GLSL 側で `sampler2D` という型になっている `uniform` 変数に対して、JavaScript 側からは `gl.uniform1i`（単体の整数を送るメソッド）でテクスチャのユニット番号を送ります。

`sampler2D` という聞き慣れない語感の型に対して整数を送るのが最初はちょっとわかりにくいのですが、ここはもうそういうものだと割り切って覚えててしまうのがよいと思います。

“テクスチャを `uniform` 変数に直接送るのでなく、バインドしたユニット番号を送るというやり方である点に注意”

## JavaScript 側の記述例

```
// バインドしたユニットの番号を整数で指定  
gl.uniform1i(this.uniformLocation.textureUnit, 0);
```

## 対応する GLSL 側の記述例

```
// sampler2D 型の uniform 変数で受け取る  
uniform sampler2D textureUnit;
```

# sampler2D に関する補足

テクスチャを扱う型なのに「サンプラ」という言葉で表現されていてちょっと紛らわしく感じるかもしれません、この背景を正しく理解するにはちょっとややこしい部分が多いです。

最初はこのあたりは深く考えずにいったんそういうものだと無理やり理解してしまうのもありだと思います。

あとで少しだけ補足する内容が出てきます

シェーダ内でテクスチャから色を読み出す

テクスチャを用いるそもそもその目的は「テクスチャから色を読み出して使う」ことです。

`sampler2D` 型の `uniform` 変数から、GLSL 内部でテクスチャの色情報を抜き出す際には `texture2D` 関数を利用します。

```
// テクスチャから、テクスチャ座標の位置の色を取り出す  
vec4 textureColor = texture2D(textureUnit, vTexCoord);
```

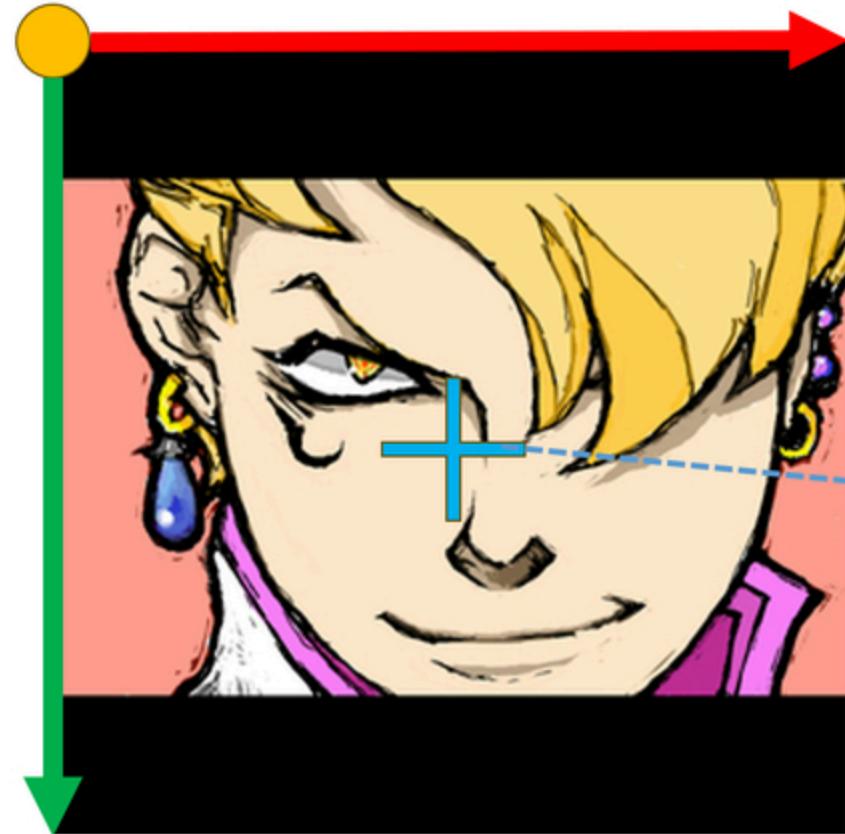
最初はテクスチャから色を読み出す処理を見ても、いまいちピンと来ない場合も多いかもしれません。

まずは「テクスチャ座標で指定された位置の色がスポットでピックアップされてくる」ようなイメージを持つと良いでしょう。テクスチャ座標が `vec2(0.5, 0.5)` であれば、テクスチャのちょうど中央あたりの色が取り出されて `vec4` の RGBA としてシェーダ内で利用できる、ということです。

(0.0, 0.0)



(1.0, 0.0)



(0.5, 0.5)

(0.0, 1.0)

(1.0, 1.0)

仮に (0.5, 0.5) というテクスチャ座標を指定すると  
ちょうどテクスチャの中心を参照することになる

※わかりやすさのため上下を反転しています

当たり前ですがテクスチャから取り出した色をどうするかは自由であり、それこそがシェーダプログラミングの醍醐味の部分でもあります。

ライティングとテクスチャマッピングを同時に行いたければ、ライトの計算を行った上で、テクスチャから読み出した色と乗算してやればいいですし、もちろん加算したり減算したり、あるいはなにか別の用途に使ってももちろんOKです。

# 011

- テクスチャの元データは 2 の累乗サイズにする
- テクスチャ読み込みは非同期処理になる点に注意
- テクスチャはバインドして初めて利用できる
- テクスチャユニット指定と組み合わせて複数同時にバインドできる
- シェーダへはユニット番号を `sampler2D` 型に対して送る
- テクスチャ座標を指標にテクスチャから色がサンプリングされる

# テクスチャパラメータ (フィルタリング方法編)

さて、テクスチャの話はここまで既にかなり長くなっていますが、実はまだ終わりません……

意外と覚えなくてはならないことが多いのがテクスチャの難しいところです。続いては テクスチャパラメータ について見ていきます。

テクスチャパラメータとは、その名のとおりテクスチャに対して設定するパラメータのことを指しています。

これにはいくつか種類があるのですが、暗記する必要はないので「どういう効果があるのか」をまずは把握しておきましょう。

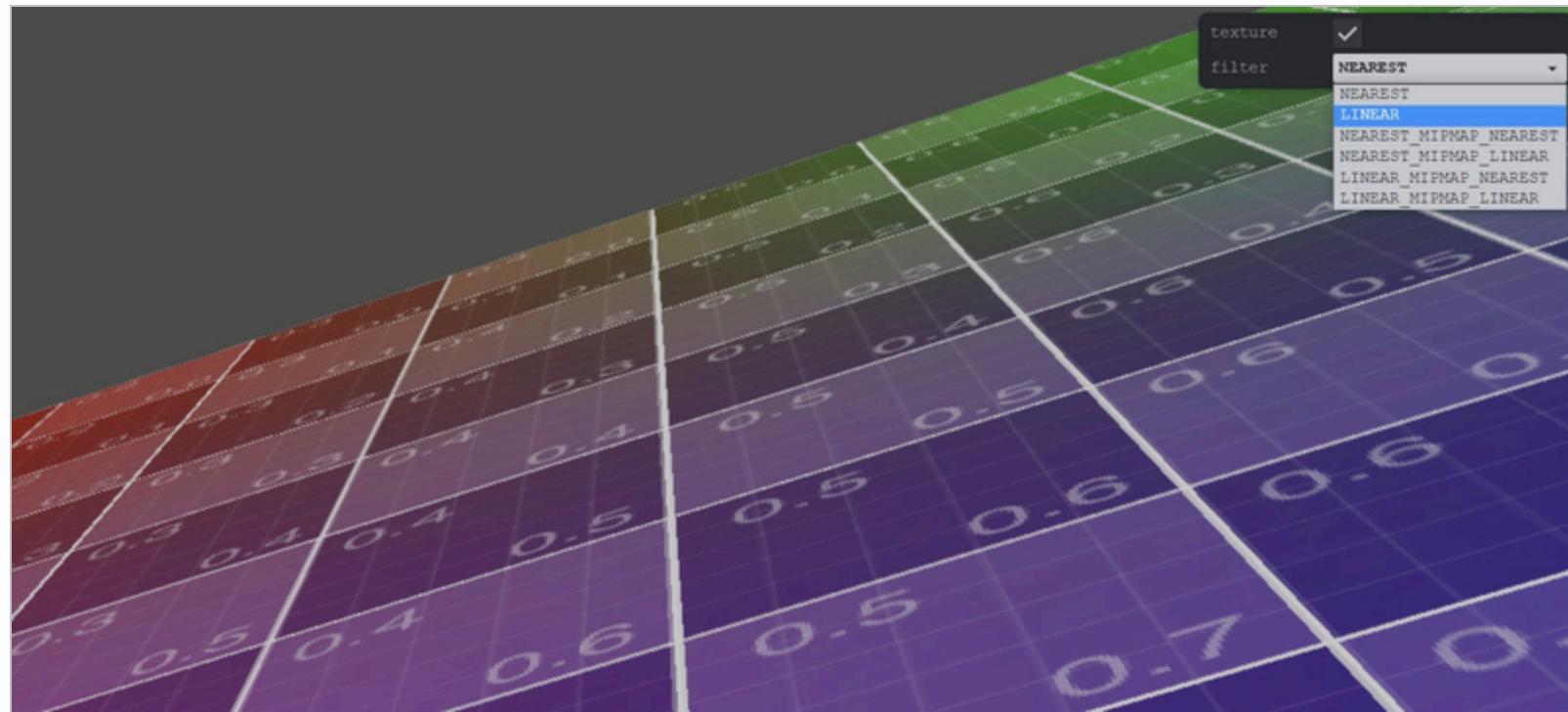
“無理に暗記までしなくても、トラブルがあった際などに記憶の断片から原因が解明されることもあるので、覚えておいて損はないはずです”

テクスチャパラメータには大きく分けて2つの設定項目があり、そのうちの1つが「テクスチャ参照時の補間方法」です。

線形に補間するのか、あるいは最近傍法で補間するのか、などが選択できます。文章で見るとなんかやたら難しいことを言っているように聞こえるかもしれません、要はフィルタリングの設定と捉えるとわかりやすいかと思います。

フィルタリングの設定は、拡大時と縮小時で、それぞれ個別に設定します。

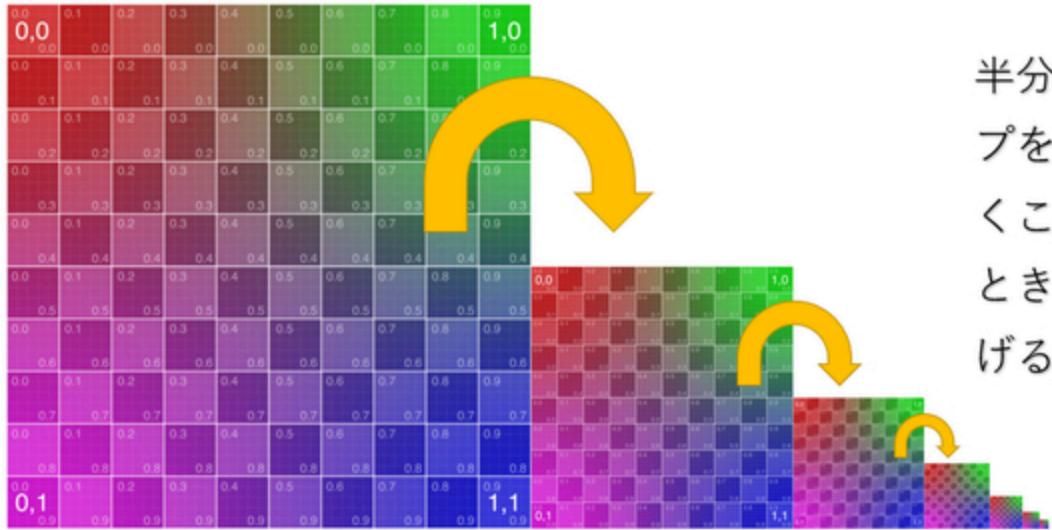
`gl.TEXTURE_MIN_FILTER` が縮小時の挙動、`gl.TEXTURE_MAG_FILTER` が拡大時の挙動の指定になります。これらの定数を引数に、メソッドとしては `gl.texParameteri` を用います。



“ポリゴンを傾けて見てみると、効果がわかりやすい”

# ミップマップについての補足

選択できるフィルタリング設定の中に **MIPMAP** と付いているものがあります。ここで登場したミップマップとは、元のイメージデータの半分の大きさのテクスチャをあらかじめ生成しておき、内部的に持つておく仕組みで、ミップマップがあることによって「縮小時の補間処理を事前に行っておくことができる」という最適化のための技術です。



半分の大きさのビットマップをあらかじめ生成しておくことにより、縮小されたときの補間の計算負荷を下げることができる。

最終的に 1px になるまで縮小したデータを事前に作っておくことで縮小補間の負荷を軽減させる

WebGL にはミップマップを自動生成してくれるメソッドがあるので、テクスチャの初期化処理ではこれを呼ぶようにしています。

webgl.js 内、`WebGLUtility.createTexture` メソッドの中身を見ると、`gl.generateMipmap` というメソッドが呼ばれていますが、この呼び出しによって自動的にミップマップが生成されます。

ミップマップは原理的に「あらかじめ縮小しておく」という機構なので、フィルタの設定としては縮小時用のフィルタ設定にしか利用できない点に注意しましょう。

`MIN_FILTER` 系のフィルタリング処理では、このミップマップが利用できます。ただし `MAG_FILTER` 系のパラメータは拡大時用の設定なので、ミップマップは使えません。

“ 設定してもエラーにならない環境はあるかもしれないが動作は不定になると考えられるので、避けましょう ”

# テクスチャパラメータ (範囲外テクスチャ座標編)

さて、テクスチャパラメータにはもう 1 つ、範囲外のテクスチャ座標が設定された場合の振る舞いという設定項目があります。

本来はテクスチャ座標は 0.0 から 1.0 の間で設定するのですが、その範囲の外の値でテクスチャを参照したときにどういうふうに振る舞うのかが、ここで設定した内容によって変化します。

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, 設定値);  
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, 設定値);  
  
// 設定値  
gl.REPEAT // 繰り返し  
gl.MIRRORED_REPEAT // 反転繰り返し  
gl.CLAMP_TO_EDGE // 切り捨て
```

実際にサンプルを動かして見てみたほうが、範囲外のテクスチャ座標、ということの意味がわかりやすいと思います

# 012, 013

- バインドしているテクスチャに対してパラメータを設定できる
- テクスチャパラメータには大きく2つの種類がある
- 1つ目はフィルタリングの方法を設定するもの
- 2つ目は範囲外のテクスチャ座標が指定された場合の振る舞い
- ミップマップは縮小フィルタにしか使えない点に注意
- なお、フィルタ設定は結構負荷が高いので、その点も注意

# ブレンディング

さて続いてですが、過去に three.js で少し触れたブレンディングを WebGL API で扱ってみます。

ただこれ、テクスチャもそうだったのですが覚えることがかなり多いため全部をもれなく暗記するというよりは、ざっくりと概要を掴んでおく程度で十分だと思います。

とは言え、説明しないのは不誠実だなと思う程度には、最終的な描画結果に大きく寄与する概念でもあるので余裕がある時にじっくり取り組んでみるのをおすすめします

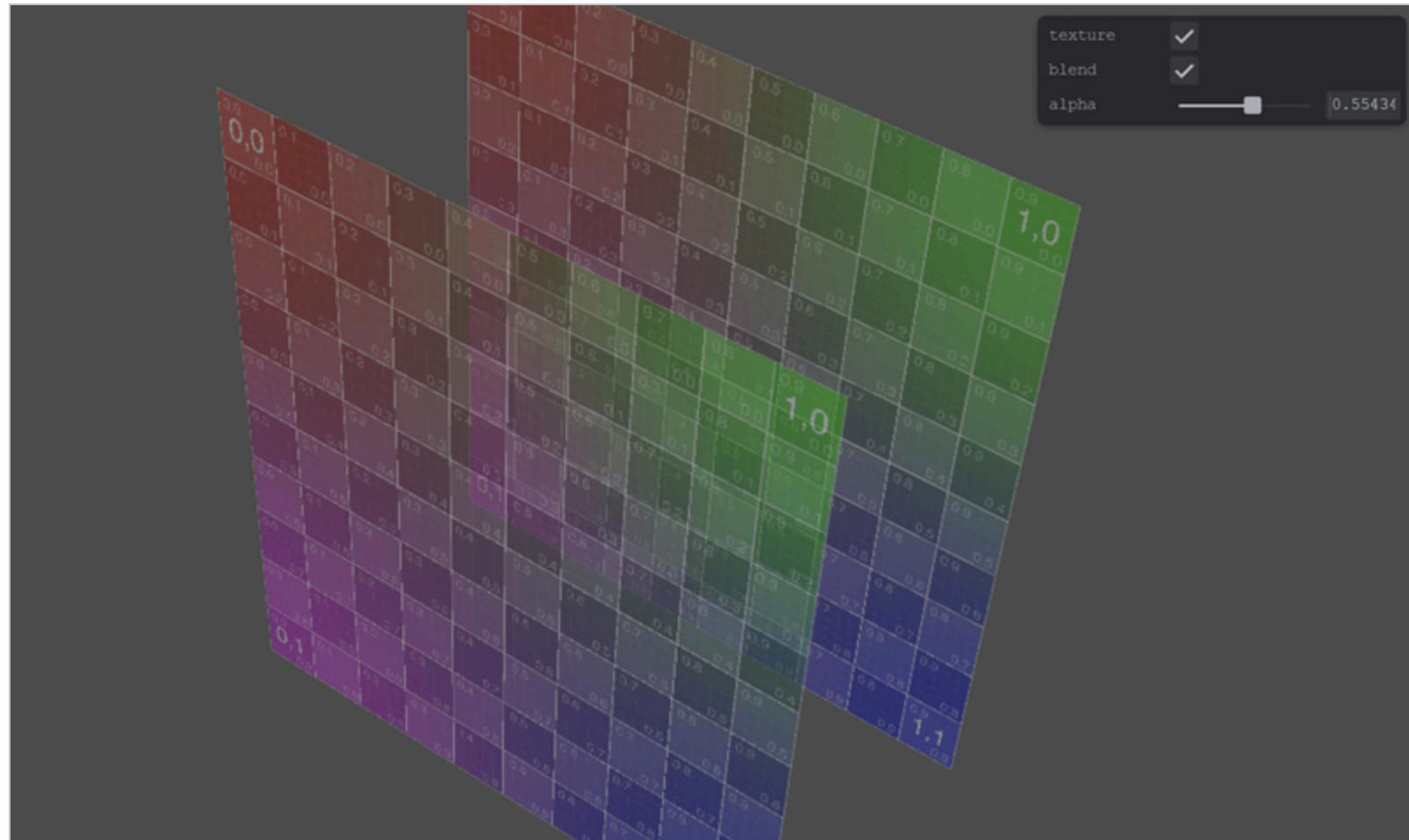
みなさんは「CG におけるブレンディング」と言われたら、いったいどんな処理を想像するでしょうか。

一般に「ブレンド」というと（たとえばコーヒーとか）なにかを混ぜあわせるような現象のことを言いますが、WebGL でブレンディングと言った場合には、これは 色と色とを混ぜあわせる ということを意味しています。

ブレンディングを理解することができると、まるでフォトレタッチソフトを使ったかのような、色を合成する処理が行えるようになります。

身近な例を挙げると「半透明のものを表示する」という処理もブレンディングを活用しています。

つまり透明・半透明なオブジェクトを描画したいときはブレンドが必須



“ブレンドの代表例、アルファブレンディング

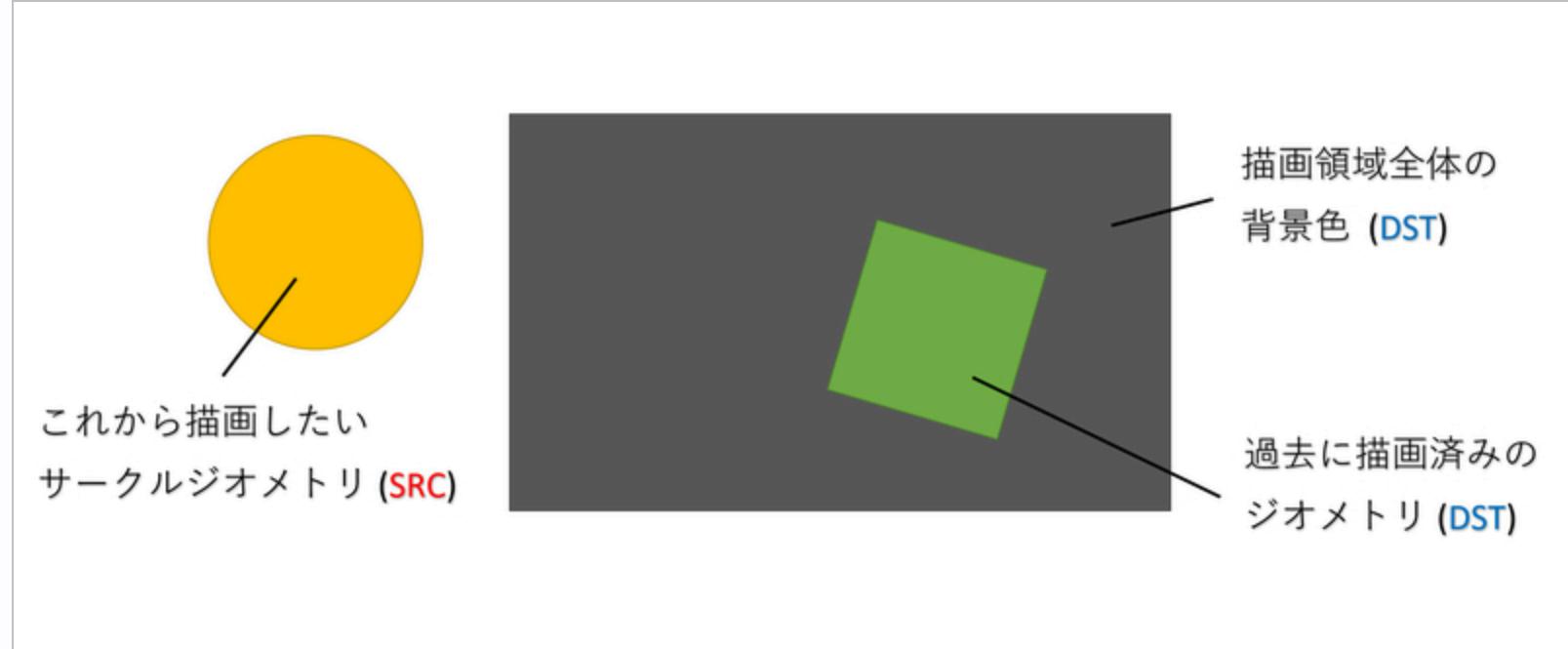
ブレンディングが色を混ぜ合わせる処理だ、と言われても、まだそれが具体的にどういうことなのかはなかなかイメージしにくいと思います。

ここを鮮明にイメージできるようにするために、色と色、この言葉をもう少し掘り下げて考えてみましょう。

# **SRC 与 DST**

ここで言う、色と色、この両者が表すものは厳密には SRC と DST という 2 つのキーワードで表されます。

SRC は、source、つまりソースとなる色で、言い換えると 今から出力しようとしている色 のことです。DST は、destination、つまり 出力済みの色 です。



グレーの部分がキャンバス（描画領域全体）

先程の図で言えば SRC は今まさにレンダリングされようとしている色です。黄色いサークルはこれからまさに描かれようとしているところで、これが SRC に相当します。

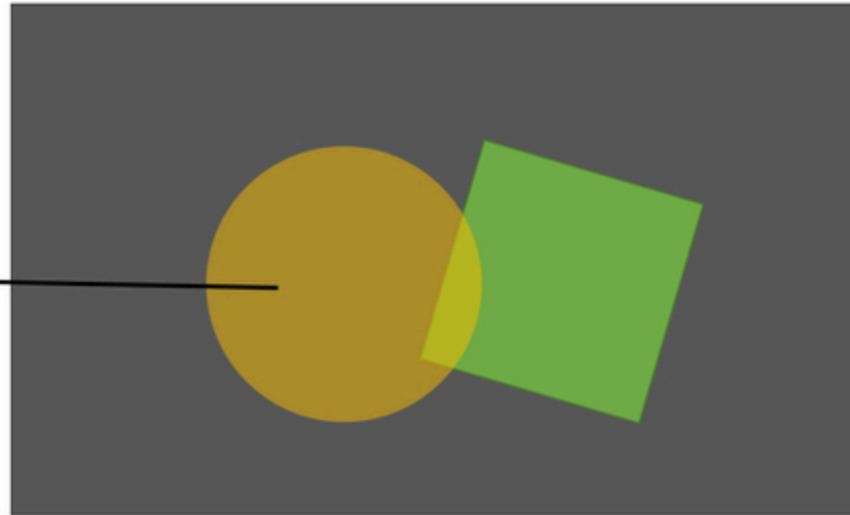
DST は既に出力されている色ということになるので、最初にキャンバス全体をクリアした色はもとより、すでに描画済みのジオメトリなども含まれます。

つまりブレンディングの一種である半透明（アルファブレンディング）を例に取ると、人間の感覚的にはセロファンやビニールなどを光にかざすと後ろにあるものが透けて見えるというのはごく自然です。

しかしこれをよりロジカルに表現しようとすると、既に下地として着色されている色 (DST) と、これから描こうとしている色 (SRC) を なんらかのルールに則って混ぜ合わせる処理 を行ってこれを再現しなければなりません。

ここでいう「なんらかのルール」にはいろいろなバリエーションがあります

自身の色（黄色）と  
下地の色を 50:50 で  
合成した場合の図



色を混ぜ合わせたことで結果的に透明になっているように見える

“透けて見える、ということをロジカルに捉える観点が必要になる”

また、これは地味に重要なのですが、ブレンディングは フラグメントシェーダよりもさらに先のステージで色を操作する 概念です。

つまり、ブレンドをひとたび有効化するとその瞬間から「シェーダから出力した色がそのまま画面に出るとは限らない」という状態になります。

このことからもわかるように、ブレンディングは GLSL とは直接関係のない部分で処理が行われます。

言い換えると、ブレンディングの設定はあくまでも JavaScript 側で WebGL コンテキストに対して行う ものであり、その細かいブレンドのロジックにシェーダで干渉することはできないということになります。

“ブレンディングに関する設定を変えると、もとに戻すまで WebGL コンテキスト全体に影響が及ぶ”

# ここまでまとめ

- 色を混ぜ合わせる処理をブレンディングと呼ぶ
- 身近な例で言うと透明度を直感どおり正しく扱うにはブレンドを駆使しなければならない (CSS の `opacity` 相当のことは勝手には起こらないし、意図して設定して初めて実現できる)
- 混ぜ合わせるのは SRC と DST で表される色と色
- ブレンドが行われるのはフラグメントシェーダのさらに後
- 基本的にブレンドを有効化すると負荷が増えると考えてよい

ブレンディングはまず有効化することで初めて利用できます。つまり既定では 無効化 されています。

思い返してみると、これまで勝手に色が補正されたり合成されたりしたことはありませんでした。これは、ブレンディングが既定では無効化されているからです。ブレンディングの有効化には「深度テストの有効化」などにも利用した `gl.enable` を用います。

引数に `gl.BLEND` を与えることで設定の状態を変化させることができます。深度テストやバックフェイスカリングなどの場合と同様、無効化するには `gl.disable` を利用します。

```
// 記述例  
gl.enable(gl.BLEND);
```

ブレンディングは、有効化した状態からさらに細かく設定を加えていくことによって初めて力を発揮します。

しかし、この設定できる内容がすごく複雑でわかりにくく、完全に理解して使いこなすのはちょっと大変です。

参考例：[wgld.org WebGL sample 018](http://wgld.org/WebGL/sample_018)

かなり大雑把に言うと.....

- ○○equation というのが色同士をどう計算するか
  - 係数を掛けたあとに足すのか引くのか
- ○○blend factor が混ぜ合わせる対象や内容を表す
  - ZERO や ONE のほか SRC\_ALPHA などがある
- RGB 部分と Alpha の部分は別個で考える（それぞれに対して個別に設定ができる）

ここまでの中でもなんとなく想像がつくかもしれません、ブレンディングは画像処理などの知識・理解がないとなかなか難しいジャンルです。

ですからまずは、WebGL にはこういう機能があり、設定にはどのような手順を踏めばいいのかということだけでも構いませんので覚えておきましょう。参考までに、最も利用頻度が高いと思われる、アルファブレンディングの設定例だけここでは掲載しておきます。

アルファブレンディングを実装するには、ブレンドファクターに以下のように指定します。

```
gl.blendFuncSeparate(  
    gl.SRC_ALPHA,           // SRC_RGB  
    gl.ONE_MINUS_SRC_ALPHA, // DST_RGB  
    gl.ONE,                 // SRC_A  
    gl.ONE                  // DST_A  
);
```

前のページの設定例をそのまま言葉で表現すると.....

1. SRC の RGB には、SRC の ALPHA を乗算する
2. DST の RGB には、1 から SRC の ALPHA を引いた結果を乗算する
3. SRC も DST もアルファは 1.0 を乗算して（要はそのまま）出力する

上記の 1 の結果と 2 の結果とを、足すのか引くのかは ○○equation の設定内容によって決まる。

また、three.js でブレンディングを扱ったときにも同じ話をしましたが、ブレンドと深度テストは性質上どうしても相反する部分があるので、ブレンドを有効化する場合は深度テストの設定状態にも常に気を配りましょう。

基本的な考え方は「透明より不透明を先に、手前より奥を先に」それぞれ描画する、です。

サンプル 014 でブレンドを有効にした状態で、様々な視点からポリゴンの重なりを眺めてみると理解が深まるでしょう

# 014

- ブレンドは基本的にかなり難しい（選択肢が無数にある）
- まずはよく使う設定例を覚えておく程度でよい
- `gl.blendFuncSeparate` に様々な引数を指定できる
- ブレンドを有効化するときは深度テストについて常に気にする
- カメラから見てどういう順番で描画されているのかが大事

three.js はメッシュを自動で Z ソートしてくれるので、可能な限り奥から順番に描画してくれます

# 環境マッピング

さて続いては、ガラリと内容が変わりまして、キューブ環境マッピングをやってみたいと思います。

キューブ環境マッピングなんてまったく聞き慣れない言葉なのではないかなと思うのですが、WebGL などの CG では結構頻繁に使われているテクニックです。ここではまず最初に「環境マッピング」がどのようなことを表しているのかから考えてみましょう。

環境マッピングとひとことで言った場合、これは「周囲の風景を写すような外観」を実現するためのテクニックの総称のことです。

たとえば、鏡とか、ガラスの表面とか、金属の表面といったような、周辺の 環境の様子がマッピングされる 質感を実現するために用いられます。



周囲の環境が頂点（プリミティブ・ジオメトリ）にマッピングされる

環境マッピングにはいくつかの手法があり、それぞれに、計算の仕方や原理が違います。

WebGL では、基本機能のひとつとして キューブ環境マッピング を API的にサポートしているため、一般的にはこれを用いて環境マッピングを行なっていく場合が多いかなと思います。

ただし、きちんと理論どおりに実装してやれば、キューブ環境マッピング以外の環境マッピングが使えないということではありません。

たとえば「スフィア環境マッピング」という球体をベースにしたマッピング手法も世の中にはありますし、これを WebGL で実現することも、シェーダを駆使すれば可能です。

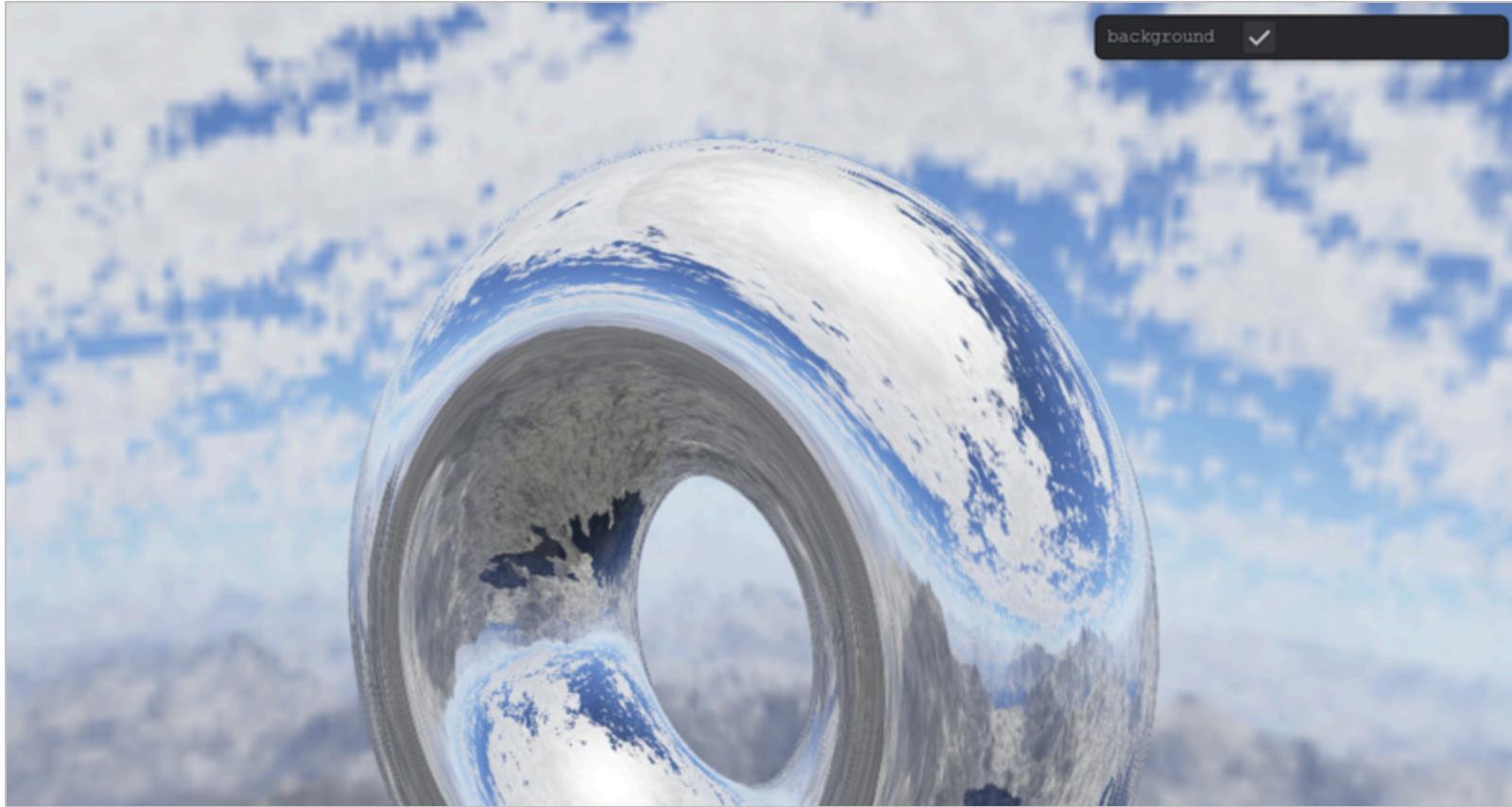
参考例：[wgld.org | WebGL: スフィア環境マッピング \(Matcap Shader\)](https://wgld.org/webgl/spherical_environment_mapping_matcap_shader.html)

さて、それでは肝心のキューブ環境マッピングとはどのような技術なんでしょうか。

キューブ環境マッピングの名前からもわかるとおりで、この技術はキューブ、つまり立方体のような構造を利用して周辺環境の写り込みを実現します。

ちょうど、正立方体のカタチをした部屋の中に入ってしまい、そこから景色を眺めるような、そんな状態を再現します。

この方法を使うと、描画される 3D オブジェクトが、まるで鏡や磨き抜かれた金属であるかのように、周囲の景色を写し込んで描画されるようになります。



“background のチェックを入れて背景をよく観察すると、実は箱のなかにいるのがわかるかも”

キューブ環境マッピングでは、箱の中に入ったような状態を再現するために、正六面体の各面に、風景に相当するテクスチャを設定します。

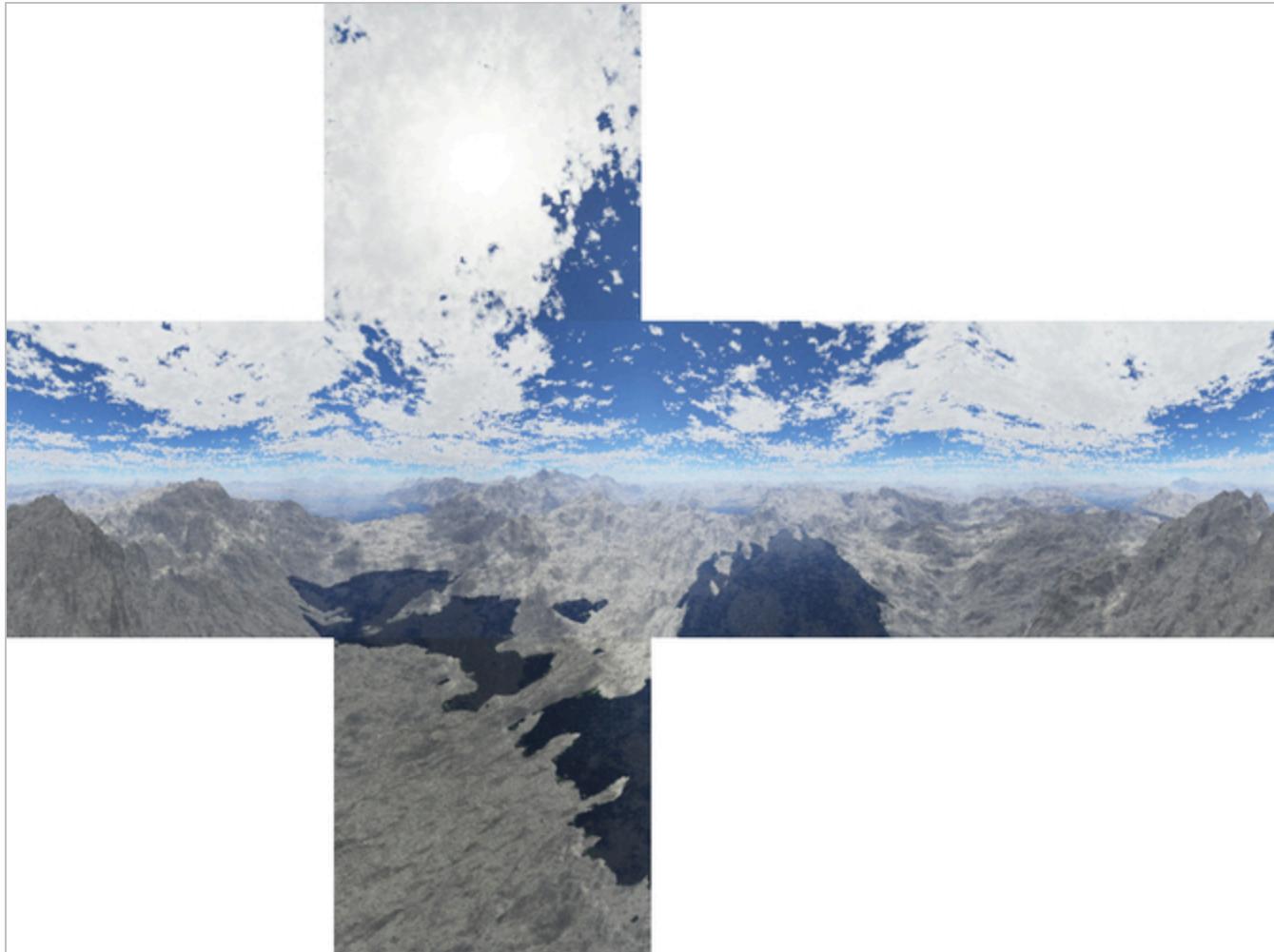
このとき利用するテクスチャは、これまでの二次元的なテクスチャとは種類が異なっており、特に キューブマップテクスチャ（または単に キューブテクスチャ）というように呼ばれます。

キューブマップテクスチャを利用する場合は、テクスチャを初期化したりデータを流し込んだりする際に正しく引数の指定をする必要があります。

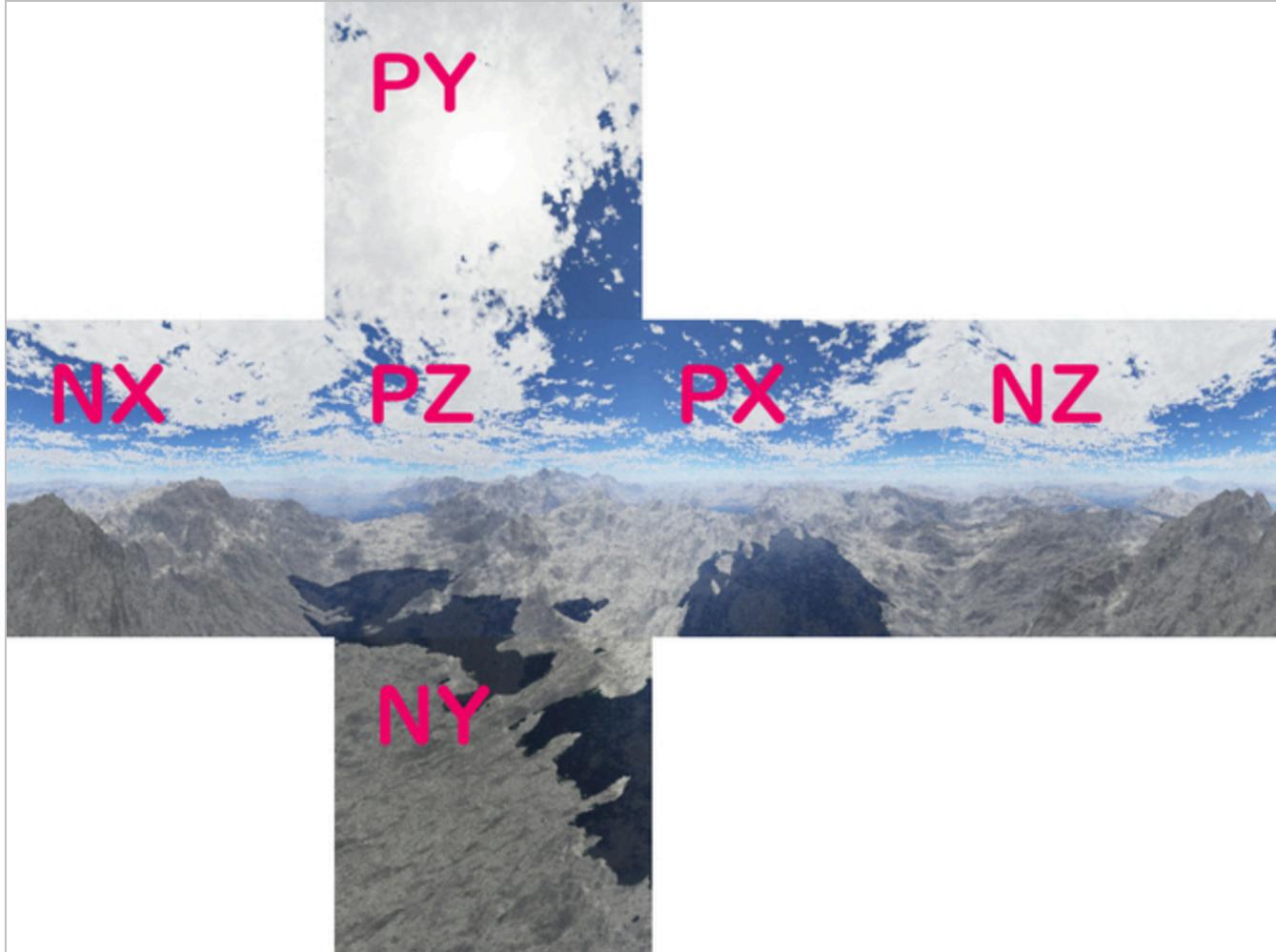
これまで `gl.TEXTURE_2D` などを指定していた箇所はすべて `gl.TEXTURE_CUBE` に置き換える必要があり、これにより内部的なデータ構造が変わります。

また `gl.TEXTURE_CUBE` を指定されたテクスチャには、正六面体すべての面を構成する画素を割当てしてやる必要があります。

1つのテクスチャオブジェクトに対して、一度に複数の画像がアタッチされた状態になります。`gl.TEXTURE_2D` の場合は二次元的な処理に用いられるので画像が1枚でよいのですが、キューブマップテクスチャの場合は、合計で6枚のイメージが必要になります。



すべての辺がシームレスに繋がるようになっている 6 つの画像



“P はポジティブ、N はネガティブの略です

このように、元となるイメージは単なる二次元的なビットマップで構わないのですが、テクスチャに割当てを行う際に、内部では各面に対して立体的にデータをアタッチします。

ポジティブやネガティブ、あるいは XYZ といった細かい指定をしながら、ひとつのテクスチャオブジェクトに対して 6 枚の画像を連続で割当ていきます。

せっかくなので、簡単に構いませんので `webgl.js` でどのようにキューブマップテクスチャを初期化しているのか見てみましょう。

`webgl.js` の `createCubeTexture` というメソッドの中で初期化を行なっています。

```
// キューブマップ用のファイル名配列 @@@
const sourceArray = [
  'cube_PX.png', ...
];
// キューブマップ用のターゲット定数配列 @@@
const targetArray = [
  gl.TEXTURE_CUBE_MAP_POSITIVE_X, ...
];
// キューブマップ用画像の読み込み @@@
return WebGLUtility.createCubeTextureFromFile(gl, sourceArray, targetArray);
```

“ 定数の配列と、ファイルへのパスの配列を引数から渡す ”

```
static createCubeTexture(gl, resource, target){  
    (中略)  
  
    // ターゲットを指定してテクスチャに割り当てる  
    target.forEach((t, index) => {  
        gl.texImage2D(t, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, resource[index]);  
    });  
  
    (以下続く)
```

すべての画像を読み込み終わってから、ループを回して一気に初期化する

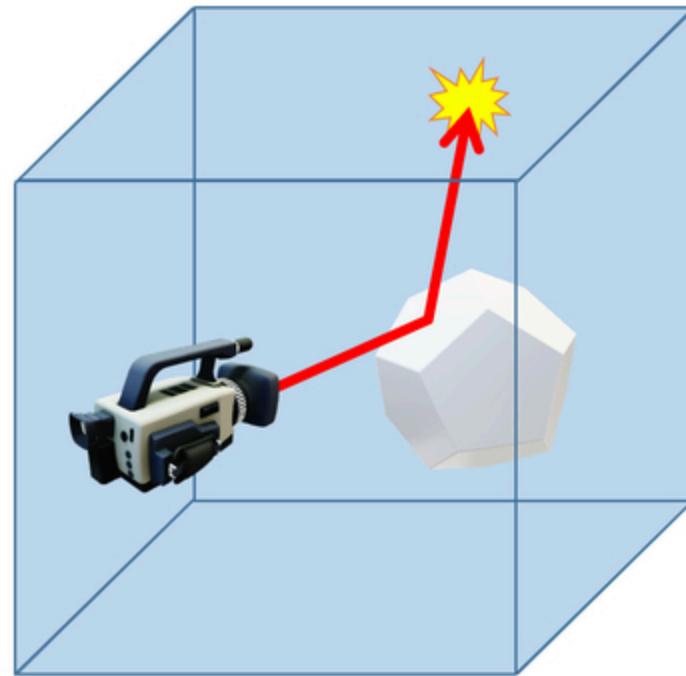
こうして、無事にキューブマップテクスチャが初期化できたら、あとは、これを今までの 2D 用のテクスチャと同じようにバインドし、シェーダ側で参照しながら利用します。

読み込んだ画像は 6 枚分でしたが、テクスチャとしては 1 つのオブジェクトなので、バインドする場合はテクスチャユニット 1 つだけを消費します。

しかし、二次元ならともかく、立方体のような構造をしているテクスチャって、頭のなかでイメージしにくいですよね。

最初のうちは、単にキューブのような構造をしたテクスチャが三次元空間上に置かれているような状態をイメージできさえすればそれでいいと思います。

視線ベクトルを法線で反射させて、キューブマップテクスチャから色を採集する



“キューブマップテクスチャの画素をサンプリングするイメージ図

# キューブマップテクスチャとシェーダ

さて、それでは具体的にシェーダ内でどのようなことが行われるのか、今度はそちらを考えてみます。

シェーダ内部では、キューブマップテクスチャを扱う際には専用の型を用います。

2D の場合は、GLSL ではテクスチャを `sampler2D` という型で扱いましたが.....

キューブマップテクスチャの場合は `samplerCube` を用います。元になった画像は 6 枚分でしたが、1 つのテクスチャに全ての画像をアタッチしているのでシェーダ内では 単体のテクスチャとして参照 します。

そして、キューブマップテクスチャを参照して画素を読み出しする際は、2D のときとは異なり `textureCube` という関数を用います。

記述例としては以下のような感じで、第一引数が `samplerCube` 型のデータ、第二引数は `vec3` のベクトルになります。

```
vec4 envColor = textureCube(cubeTexture, reflectVector);
```

テクスチャ座標が `vec3` 型で三次元ベクトルになるのちちょっとおかしな感じがするかもしれませんか.....

先程図解して見せたように、キューブの中から視線ベクトルを伸ばしていくようなイメージで考えるとわかりやすいと思います。

具体的な実装方法としては「視線ベクトルを、法線をもとに反射させたベクトル」を用います。

反射させたベクトルなんてどうやって定義したらいいのかと思ってしまうかもしれません、GLSL には `reflect` という反射ベクトルを計算するための関数がビルトインで実装されているので、それを用います。

実際のコードはもう少しあとで説明します

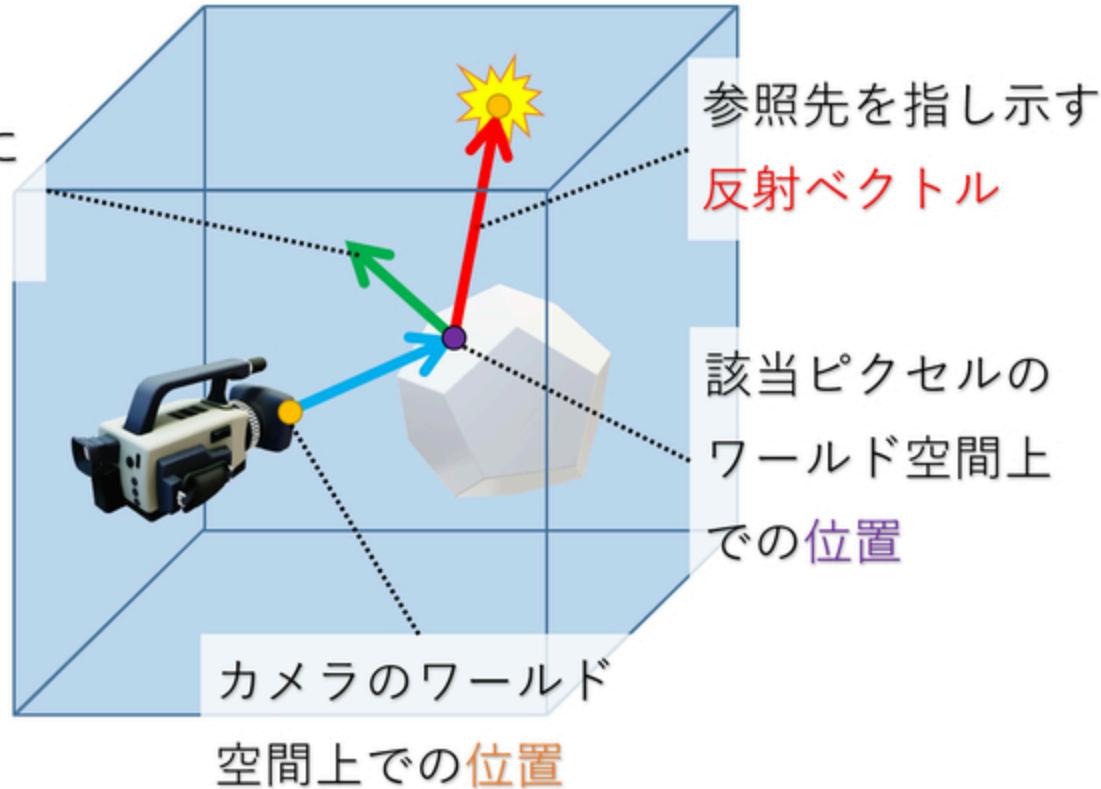
# ここまでまとめ

- 環境マッピングとは周囲の環境を写し込むような外見を得られる手法
- 周囲の環境が写し込まれたかのように見えるテクスチャマッピング
- キューブ環境マッピングは WebGL で API でサポートされている
- キューブ型の専用のテクスチャタイプを用いる
- 法線などを用いて三次元ベクトルでテクスチャを参照する

具体的なコードを見ていく前に、まず方法論をしつかり押さえておきましょう。

キューブ環境マッピングでは先程図解したとおり、カメラの位置と、カメラから伸びる視線、さらには視線がぶつかった面の法線、などがシェーダ内で使える状態である必要があります。

該当ピクセルに  
おける法線

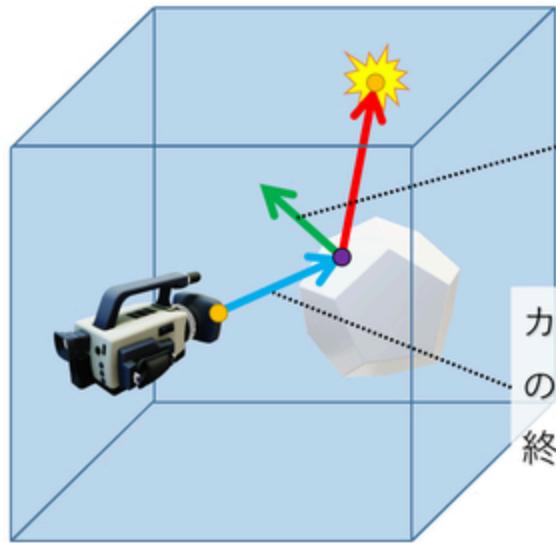


シェーダに送る uniform 変数としては「ワールド空間でのカメラの位置」がまず必要そうだな、というのは図を見るとわかると思います。

「該当ピクセルのワールド空間の位置」というのは言葉で書くとイメージしにくいかもしれません、これはモデル座標変換行列を使って頂点シェーダ内で求めることができます。ですから MVP 行列だけでなく、モデル座標変換行列も別途シェーダに送ります。

「該当ピクセルの法線」は、陰影計算を行った場合と同じように `normalMatrix` を使って法線を変換してやれば得られます。これは以前と同じです。

カメラの位置、ピクセルのワールド空間の位置、という 2 点が明確ならそれを結ぶことで視線ベクトルが定義できますので、これですべての必要な情報が揃います。



該当ピクセルにおける法線は陰影計算の  
ときと同様に、行列で変換して求める

カメラのワールド空間上での位置が始点、該当ピクセル  
のワールド空間上での位置が終点なので.....  
終点 - 始点で視線ベクトルを求めることができる

# 頂点シェーダ

```
// 頂点のモデル座標変換  
vPosition = (mMatrix * vec4(position, 1.0)).xyz;  
// 法線の変換  
vNormal = normalize((normalMatrix * vec4(normal, 0.0)).xyz);  
  
gl_Position = mvpMatrix * vec4(position, 1.0);
```

“varying 変数 vPosition に頂点のモデル座標空間での位置を格納してフラグメントシェーダへ”

フラグメントシェーダ内では、各種情報から視線ベクトルを求め、前述のビルトイン関数 `reflect` を用いて視線ベクトルと法線から反射ベクトルを求めます。

コード量が少し多くなるので、実際のサンプルのシェーダを見ながら、コメントをじっくり読みつつ落ち着いて考えてみてください。

## 015

- 頂点シェーダでモデル座標変換後の頂点の位置を求める
- フラグメントシェーダ内で視線ベクトルを求める
- 視線ベクトルを法線で反射させる
- 反射ベクトルをテクスチャ座標とみなしてサンプリング
- 取得した色を使うと写り込みのような効果が得られる！

# 屈折マッピング

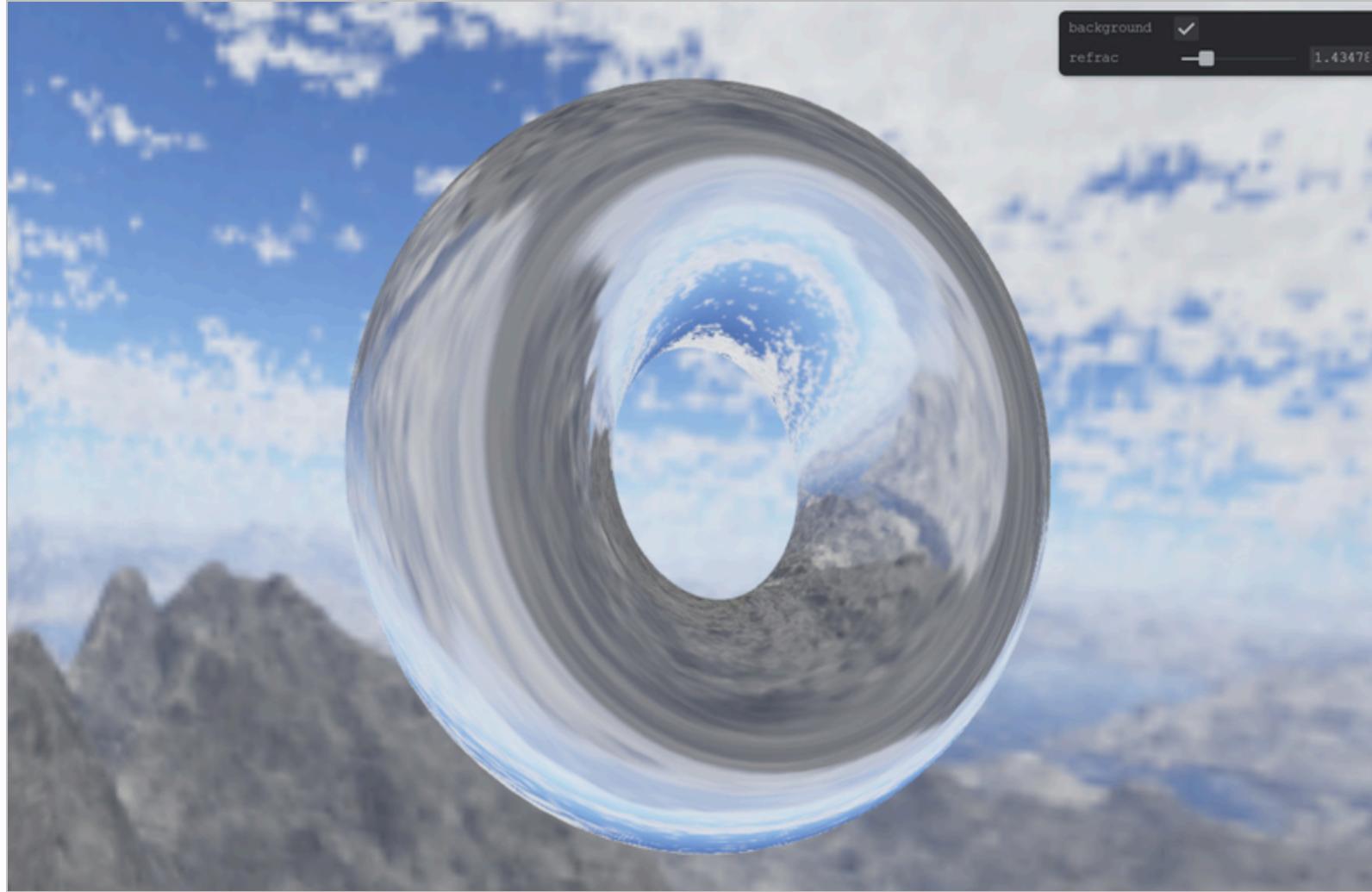
キューブ環境マッピングの実装は、キューブマップ用の画像を用意したり、JavaScript での初期化処理が冗長になったり、という感じで実装というより準備のほうに手間が掛かる印象があります。

一方で、一度動く状態にまで持っていってしまえば応用することも比較的簡単にできます。

そんなキューブ環境マッピングを利用した応用例のひとつが 屈折マッピング です。

先程のキューブ環境マッピングでは反射ベクトルを使いましたが、屈折マッピングではその名のとおり「屈折ベクトル」を求めて、それでキューブテクスチャを参照します。

“ 雰囲気がガラッと変わり、鏡面からガラスや水晶のような質感に ”



屈折する様子を再現することで違った質感に

屈折マッピングの下準備は、基本的にキューブ環境マッピングのものとまったく同じです。

変化があるのはシェーダ側になります。

反射ベクトルは `reflect` という関数と法線を使うだけで簡単に求めることができましたが、屈折ベクトルって言われても、それをどうやって用意したらいいのか、簡単には想像できないかもしれません。

光、あるいは光線が、水やガラスなどにぶつかって屈折する、ということはみなさんもなんとなく想像できるかと思いますが、それを数学的に計算する方法をご存じの方はかなり限定されるのではないかでしょうか。

ただこれに関してはあまり深く心配する必要はなく、GLSL には屈折ベクトルを計算するためのビルトイン関数が最初から用意されています。

屈折マッピングを行う際には、先程使った `reflect` の代わりに屈折ベクトル用のビルトイン関数を使ってやるだけで基本的には OK です。

屈折ベクトルを求めるプロセスは以下のような感じ。

```
// 屈折率の比（大気を 1.0 として、入射するオブジェクトの屈折率で割る） @@@
float eta = 1.0 / refractiveIndex;
// refract の第三引数に eta を指定する @@@
refractVector = refract(eyeDirection, normal, eta);
```

通常のキューブ環境マッピングと比較して、変化した部分としてわかりやすいのは屈折率を計算している部分が増えていることでしょう。

正確には、屈折率の比を求めるのですが、これには「物理的に正しい屈折率で 1.0 を割る」という計算を行ってやります。

屈折率は、以下のページなどを参照すると知ることができます。

これは単純に物理の話なので、これを元に計算を行えば、現実の世界と同じような外見に近づけることができます。

参考： [屈折率 - Wikipedia](#)

さらに、屈折率の比から、今度は屈折ベクトルを求めます。

GLSL では、反射ベクトルを求めるための `reflect` という関数の他に、屈折ベクトルを求める `refract` という関数があります。2 文字違いなうえに意味も似ているので、間違えないように気をつけましょう。

“`reflect` は反射、`refract` は屈折！”

# 016

- JavaScript 側の実装は通常のキューブ環境マッピングと同じ
- シェーダで反射ベクトルではなく屈折ベクトルを求めて利用する
- 屈折ベクトルを求めるために「屈折率の比」を使う
- `reflect` が反射ベクトル
- `refract` が屈折ベクトル

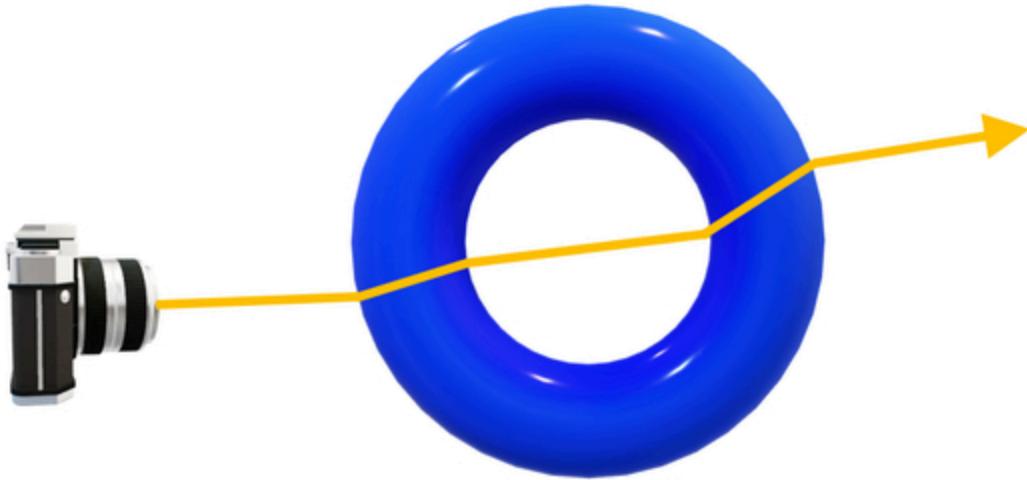
# 屈折マッピングの補足

最後にちょっと補足すると、キューブ環境マッピングにしても、あるいは屈折マッピングにしても、これらは単なる「テクスチャへの参照とサンプリングでしかない」ということには留意しておきましょう。

たとえば、周囲の風景が写り込んでいるように見えるとは言っても、これはあくまでもテクスチャマッピングの一種でしかないので、隣にあるオブジェクトの姿が勝手に写り込んだりすることはありません。（要は現実世界の鏡と同じ挙動になるわけじゃないということ）

また、屈折マッピングも、あくまでも屈折ベクトルを使ってそれっぽく近似しながらテクスチャを参照しているだけです。

ですから、現実世界の屈折とまったく同じ状態を完全に再現できるわけではありません。たとえばトーラスとかを屈折マッピングで描画しているときを考えてみると……



実際の屈折は、大気中に抜ける際にも起こる上、再度同じオブジェクトの中に入ることもあるが、屈折マッピングがこれを完璧にシミュレートしているわけではない。

現実世界の屈折はもっともつとはるかに複雑！

環境マッピングは結局のところ、テクスチャを参照してそれっぽく見せているだけなんですね。

もし本当に周囲の風景が動的に変化する様子まで写り込ませたければ、キューブマップテクスチャを動的に描き続け、参照するテクスチャそのものをアップデートし続けてやる必要があります。

さいごに

今回はテクスチャに関連した話と、ブレンディングについて扱いました。

いずれも CG の文脈から言えば基本中の基本ではあるのですが、ウェブではほとんど意識しないような内容なので、ちょっと難しく感じたかもしれません。

CSS で気軽にできる半透明も、CG 的な観点で見てみるとなかなか複雑なことをやっていることがわかる

こういった基本的なことは知らなければ知らないで（three.js などを使う上では）なんとかなってしまうことも正直多いです。

しかし、なにかしらのトラブルに見舞われた場合や、より表現の幅を広げていく過程では、こういった基本知識が役に立つ場面がたくさんあります。無理せず、少しずつ身につけていくのがよいでしょう。

さて今回の課題ですが、テクスチャを複数同時に利用する実装に挑戦してみましょう。

たとえば、複数の画像からテクスチャを生成してシェーダに送り、画像をフェードさせ切り替える、などはテーマとしては手軽ですし面白いと思います。

また若干難易度が高いですが、CG の用語で言う「ノーマルマップ」などの技術もテクスチャを複数同時に利用するタイプのテクニックです。

ここで提示したのはあくまでも一例ですが、応用できる範囲も広いのでぜひマルチテクスチャ実装に挑戦してみてください。

ノーマルマップなどはかなり発展系のテクニックなので、課題はもっと単純なものでもちろん構いません

# ディストーションエフェクトの実装例

[Creative WebGL Image Transitions](#)

# ノーマルマップを利用した実装の参考例

[Exploring bump mapping with WebGL](#)

あくまでもアイデアの参考程度に