

# **three.js の基本機能を使いこなそう**

**最低限押さえておきたい各種機能群**

はじめに

前回は three.js の基本中の基本を押さえるような内容でした。

初めて 3D プログラミングを行う、という場合は、知らない概念や用語がたくさん同時に出てきますので最初は面食らうかもしれませんが、少しずつ慣れていくしかありません。前回も何度か言いましたが、楽しく継続していく、ということがとても大切です。

今回も、前回に引き続き three.js を利用したサンプルを使って進めていきます。

three.js には本当に多彩な機能がありますので、今回はそのあたりを少し掘り下げていってみましょう。

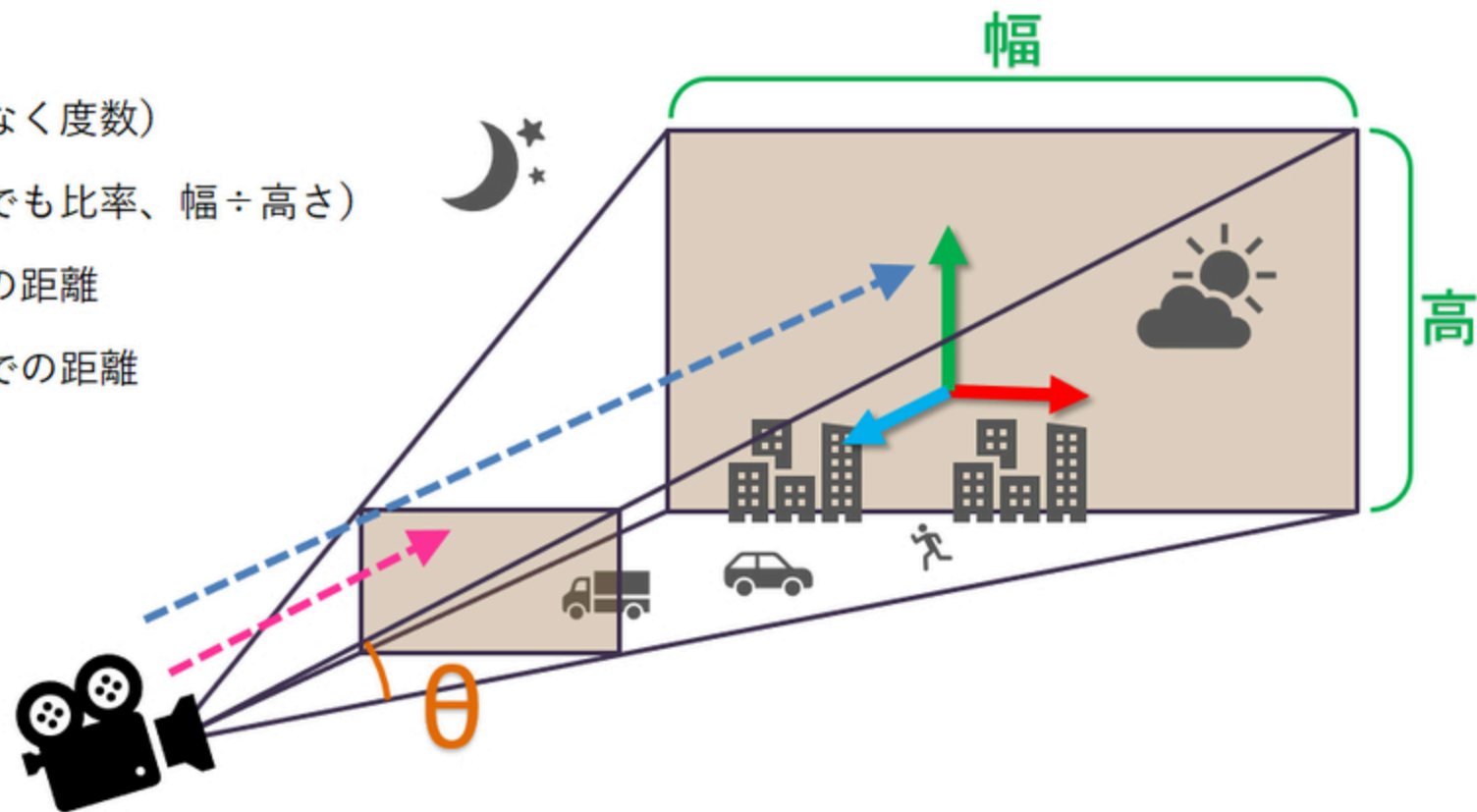
**カメラが変わると世界が変わる**

さて、今回はまず three.js のカメラについて見ていきます。

これまでのサンプルでは、一貫して `THREE.PerspectiveCamera` を使ってきました。このカメラは、前回の講義の最後の方に出てきた図式にあるように空間を四角錐のような形として切り取ります。

- fovy**  $\theta$  の部分の角度（ラジアンではなく度数）
- aspect** クリップ空間の縦横比（あくまでも比率、幅÷高さ）
- near** カメラからニアクリップ面までの距離
- far** カメラからファークリップ面までの距離

ニアクリップ面とファークリップ面の間の、  
視錐台に囲まれた空間内だけがレンダリング  
の対象になり、範囲外のは画面には出ない  
※つまり月は範囲外なので画面には描かれない

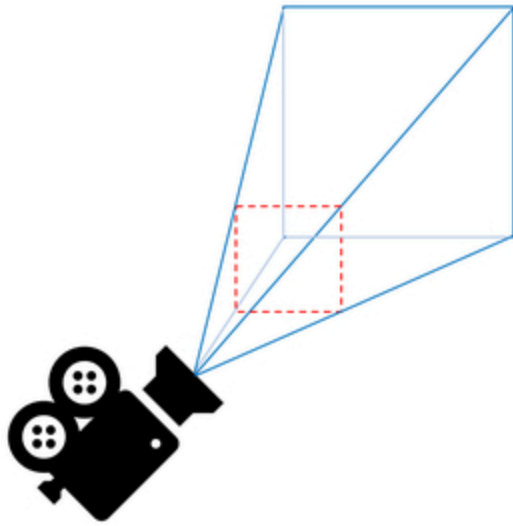


カメラを中心に四角錐の形に空間を切り取る

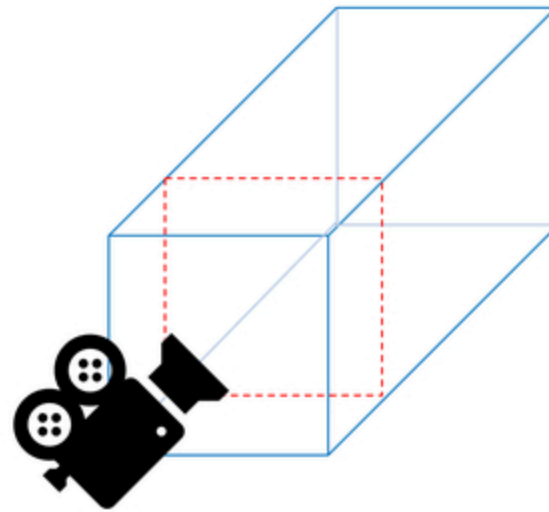
一方で、three.js には他にも `THREE.OrthographicCamera` という種類のカメラがあります。このカメラを使うと、空間を切り取る方法が変化します。

どのように変化するかというと.....





Perspective  
透視投影



Orthographic  
平行投影

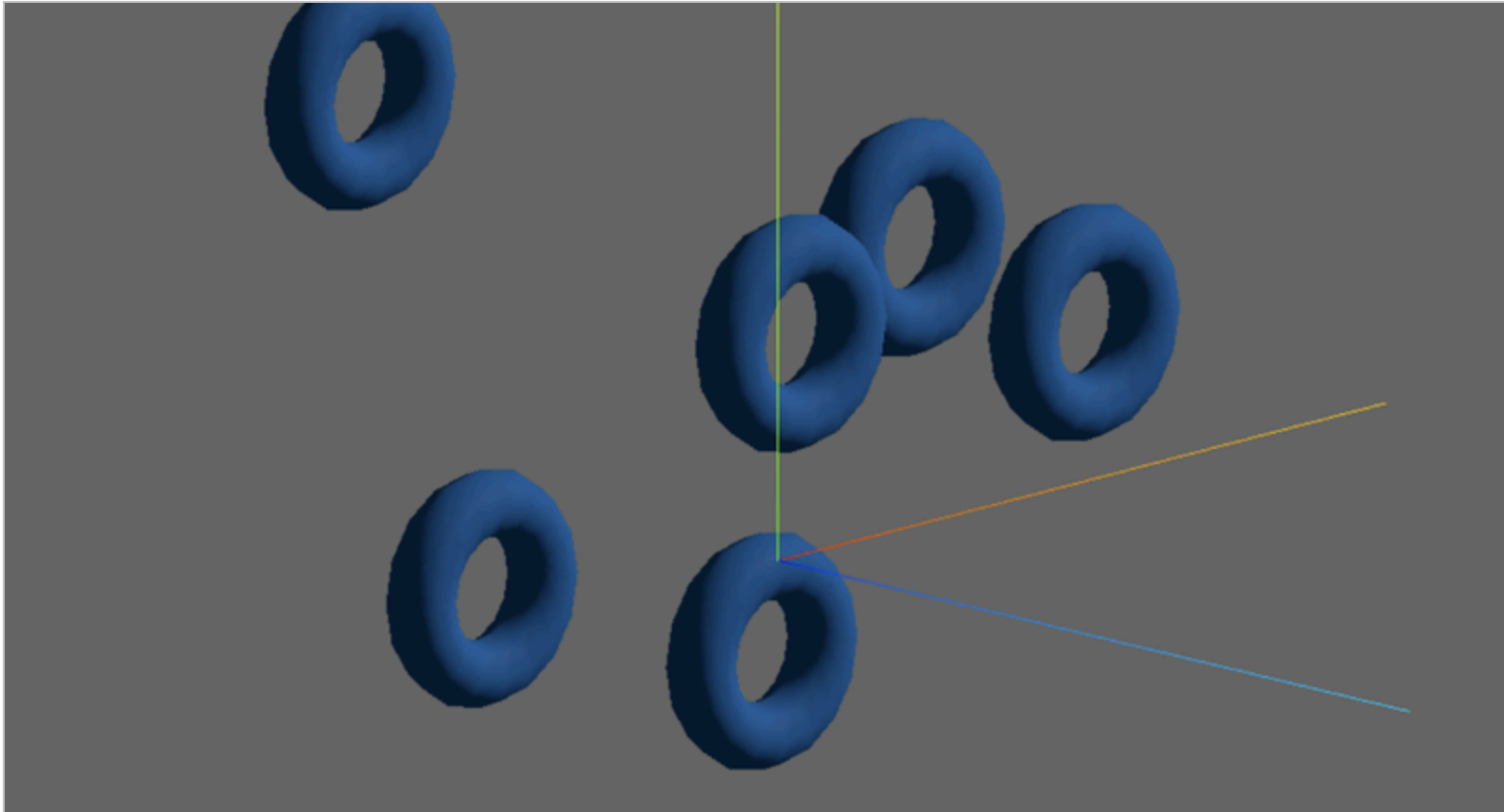
“ 切り取られる空間が直方体の形に！ ”

実際の現実世界には Orthographic なカメラは存在しませんが、CG 的には（あくまでも計算でシミュレートしているに過ぎないので）こういうことも可能なんですね。

では、このように「空間の切り取り方」が変わると、見た目にはどういった変化が起こるのでしょうか。

言葉で書くと、簡単に言えば「遠近感のあるパースが掛からなくなる」というのがわかりやすいでしょう。

また、カメラを定義するときの初期化処理でも、透視投影時のカメラの設定とはまったく異なるパラメータが必要になります。



“手前でも奥でも、同じ大きさに画面に出てくる（パースが掛からない）”

# 010

- これまでとは違い `THREE.OrthographicCamera` を用いる
- 空間を直方体で切り取るイメージなので.....
- 視野角（fovy）などのパラメータは存在しない
- 上下左右の位置を直接指定して、カメラの正面の空間を切り取る
- 画面上に現れる描画結果はパースが掛からないものになる

# 平行投影の使いどころ

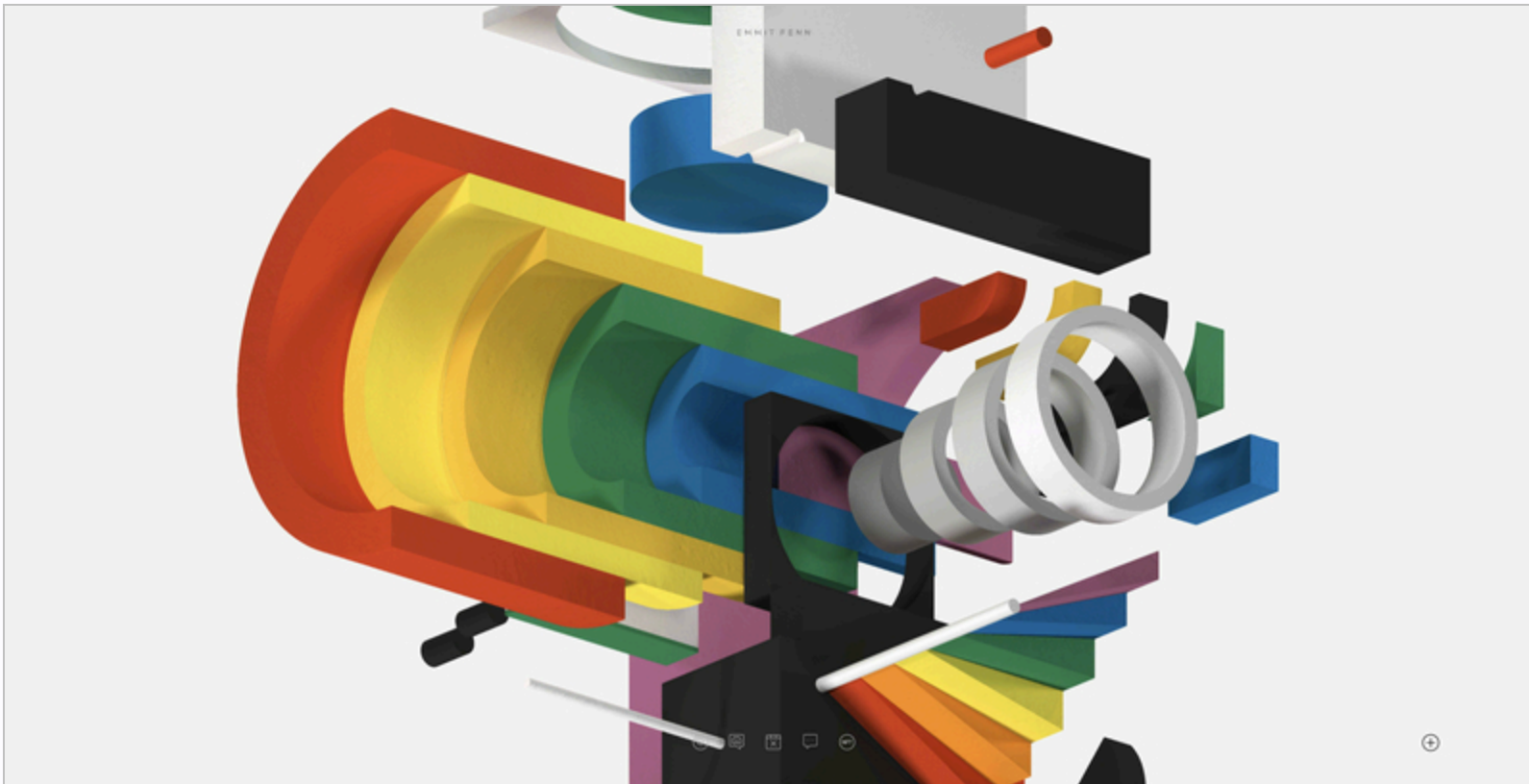
平行投影の CG では、現実世界ではありえない「パースの掛からない 3D 空間」が描かれます。

こんなパースの掛かってない CG なんて使いみちがあるかな？ みたいを感じる方ももしかしたらいるかもしれませんが..... これはこれで、実は結構有用なんです。

たとえば、以下のようなケースでは平行投影のほうが優れていることが多いです。

- あえて非現実的な質感でトイっぽさやゲームらしさを演出
- インターフェースなどのパースが不要なオブジェクトを描く
- 2D 的な描画結果を WebGL や OpenGL などでも描きたいとき
- 建築物を描く場合など、直線的にピッタリ揃えて描いたほうが絵になるとき





## Emmit Fenn – Far From Here

“ パースが掛からないからこそその表現手法 ”

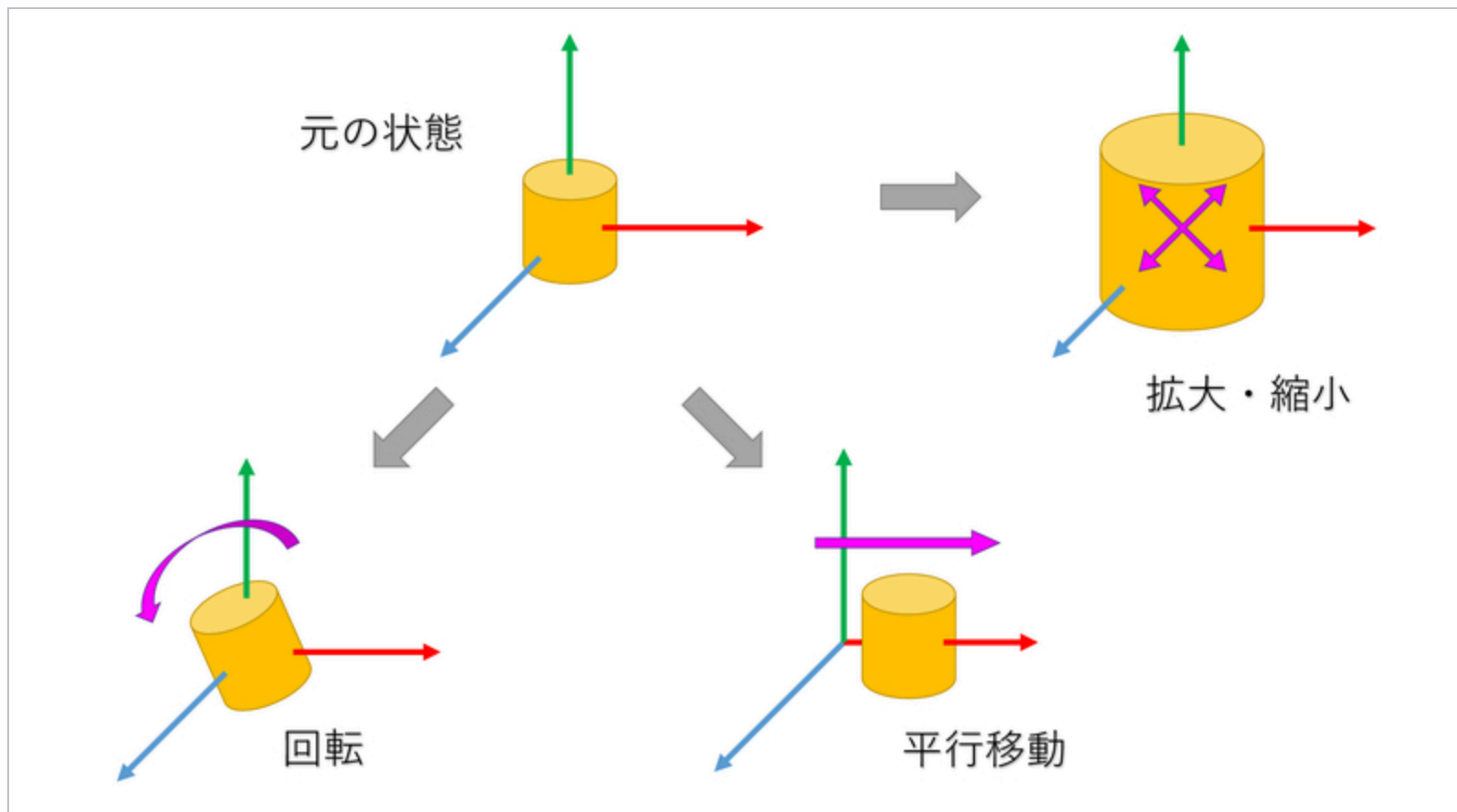


“ ゲームの残り HP 表記など、3D シーンに連動して位置が変化してほしいが奥行きに応じた拡大縮小はしてほしくない、みたいなことは実は意外と多い ”

# 回転の概念

さて、続いては 3D に不慣れなうちにハマりやすい 回転の概念 について考えてみます。

前回の講義では、three.js の Object3D というクラスには `position` や `rotation` といった、オブジェクトを操作することができるプロパティが備わっているという話をしました。



rotation で回転、position で平行移動、scale で拡大縮小を操作できる

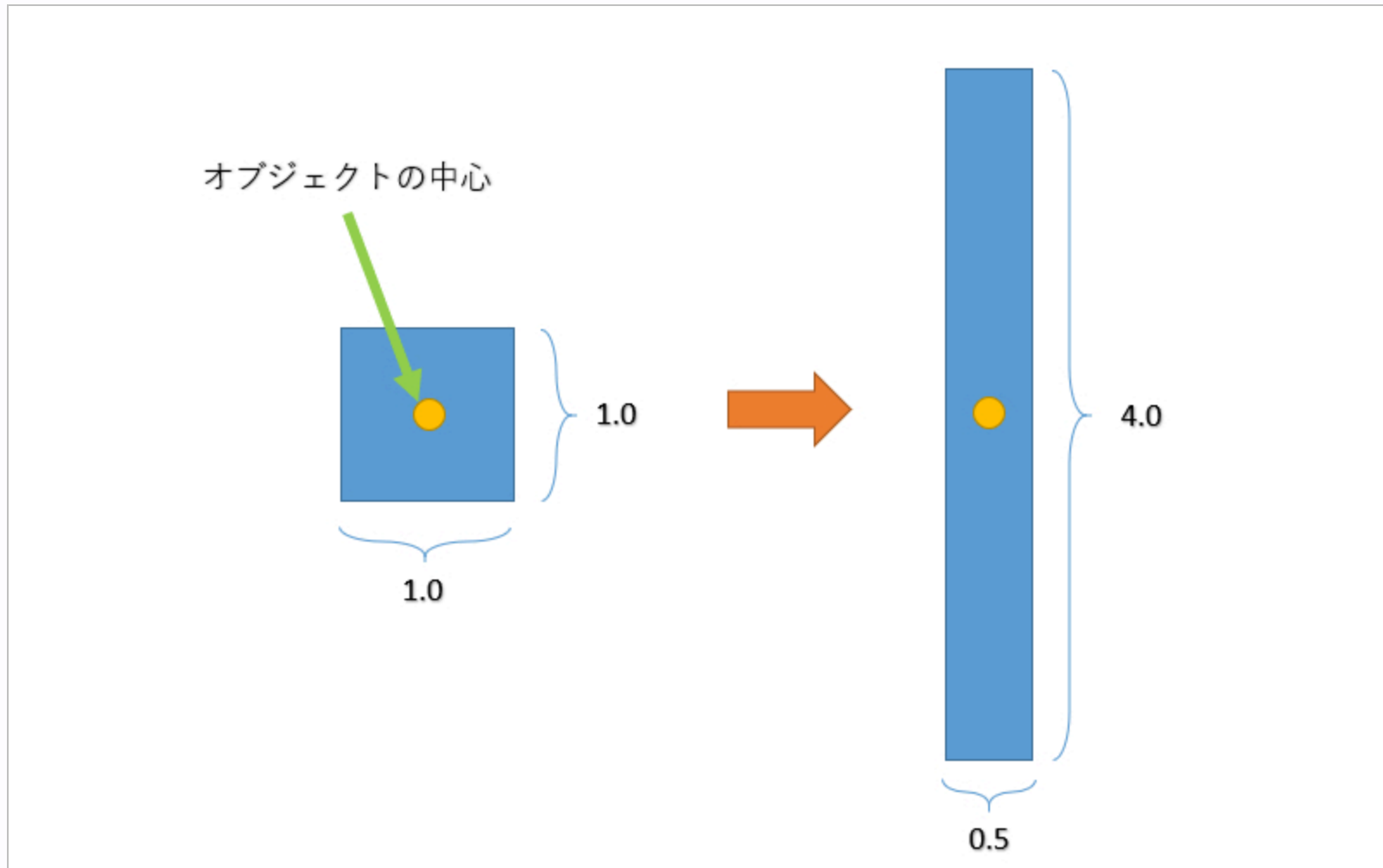
このようなプロパティを操作してオブジェクトを動かしたり、あるいは回転させたりといった処理は、恐らくみなさんにとってもそれほど難しくなく直感的な操作だったと思います。

でも、3D プログラミングにおける回転の概念というのは、実は結構複雑で、ちょっとだけ難しい部分があります。

たとえば、アナログ時計の針をイメージしてみましょう。

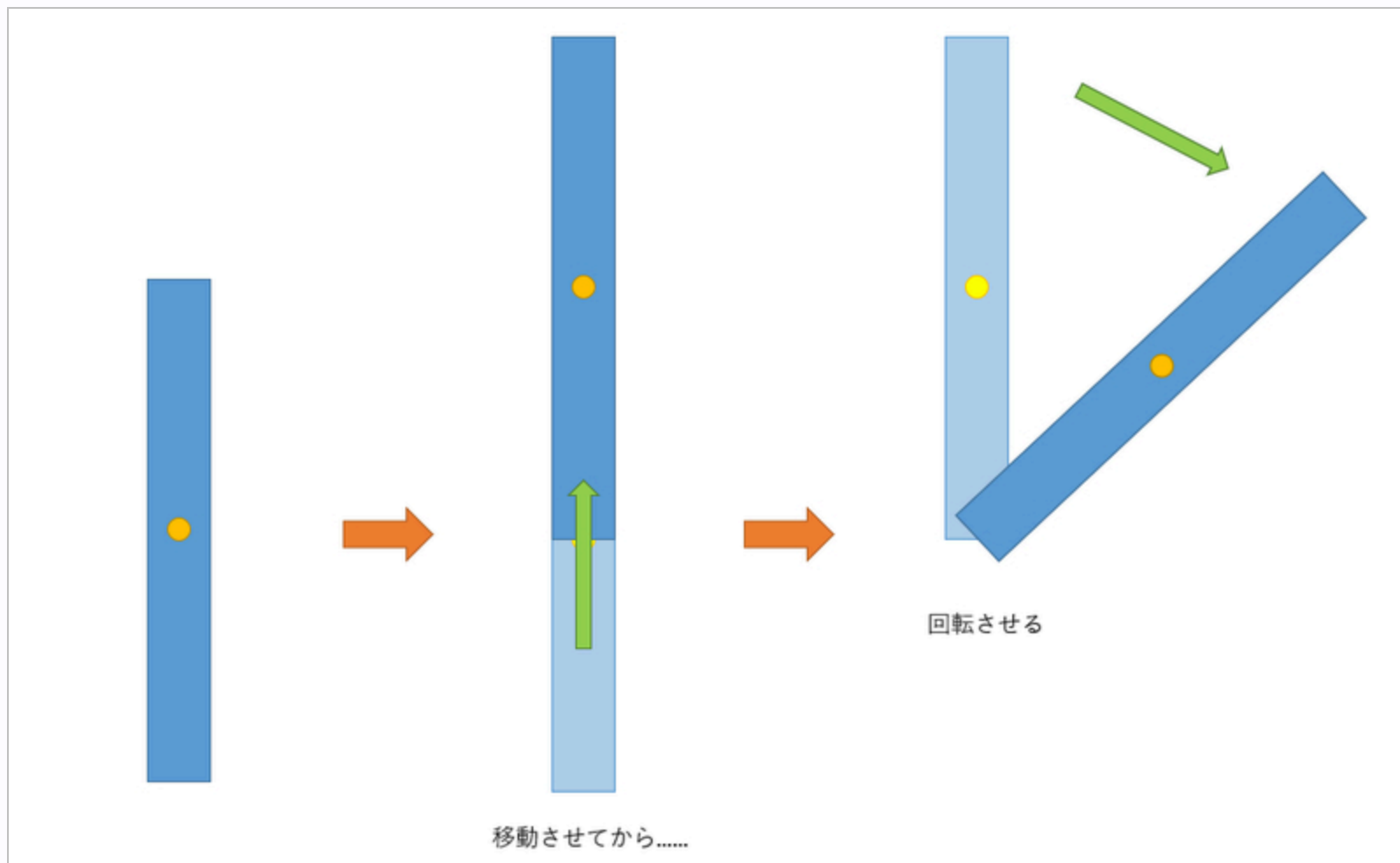
時計の針って、文字盤の中心を軸にして回転しますよね？あのような回転って、前回のサンプルにあったような `rotation` を使った回転では実は 表現できない のです。

そもそも時計の針のようなオブジェクトを描画したければ、ボックスを細長くしたようなオブジェクトを作り.....





順番的には「移動」が先で、「回転」が後から行われれば良さそうなように思えます。



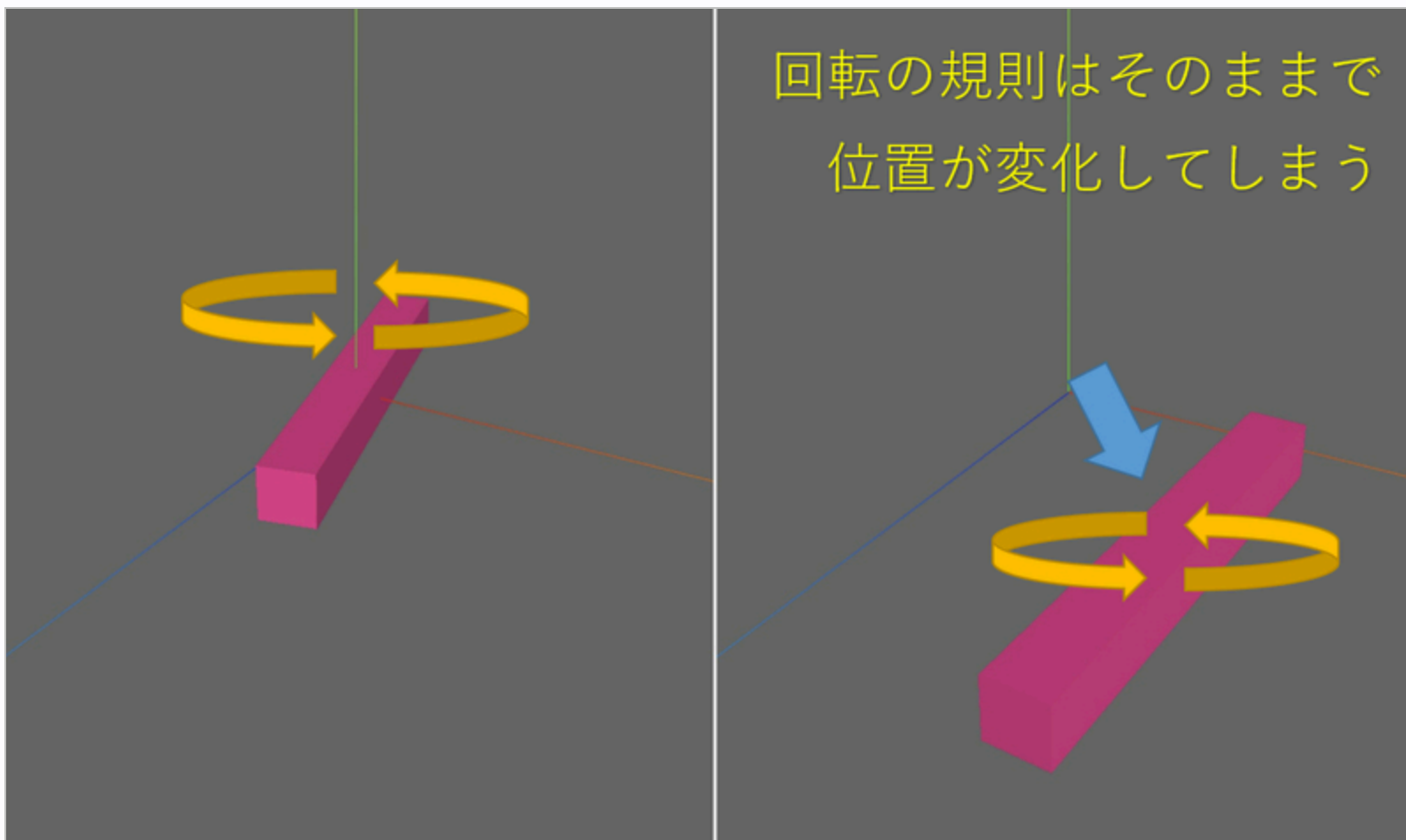
次のページのリンクから閲覧できるサンプルでは、スペースキーが押されている間 は `position` プロパティを変化させて、オブジェクトを移動させています。具体的には、以下のようなコードです。

```
// 以下を見ると、コード上は「移動 → 回転」という順序で
// コードが書かれているように見えるが.....
if (this.isDown === true) {
    this.box.position.set(2.0, 0.0, 2.0);
} else {
    this.box.position.set(0.0, 0.0, 0.0);
}
this.box.rotation.y += 0.02;
```

## 回転サンプル その1

常に回転し続けているオブジェクトがスペースキーの押下によって移動した際、どのような挙動になるのか確認しましょう。

特に「どこを軸にして回転しているのか」を見極めるのがポイント



“コードとは逆に回転してから移動しているような動きになる”

時計の針のように、特定の位置を中心にして回転させたいという場合、先程のようにオブジェクトの本来の中心位置でしか回転が行えないということになると困ってしまいますよね。これこそが回転表現の持つ難しさです。

一見、回転させるだけなんて非常にシンプルなことのようにも思えるのですが、実は結構ややこしいのです。

しかし実際に実行結果を見ても、これがどういうことなのか、どうしてこのような結果になってしまうのかはいまいちわかりにくいと思います。

そこで、このような挙動を理解するために非常に重要な回転処理におけるポイントを覚えましょう。

## 1. 回転が先か、移動が先か

まず、3D に限らず 2D の場合でも「回転が先に起こるか平行移動が先に起こるか」の違いは非常に重要です。先程のサンプルでは、`position` を変化させたとしても、回転する際の軸は変化しませんでした。

つまり three.js の世界では「常に」回転が先に起こる ということです。

## 2. 回転は世界全体を原点を中心に回転させる

これも 3D の場合に限りませんが、回転は 対象となる世界全体 に影響を与えます。回転の基点となる位置は常に、座標が XYZ で  $(0, 0, 0)$  となる原点です。

three.js の場合は、世界全体を回転させる処理を各オブジェクトごとに細切れに実行しています。



つまり、three.js の内部では、移動と回転を行う処理を実行すると以下のような流れで処理が行われています。

- 処理の対象となるオブジェクト（Object3D のインスタンス）を中心とした空間を考える
- 原点を中心に対象となる世界全体を回転させる（常に回転が先）
- オブジェクトを移動させる（回転したあと移動）
- 次のオブジェクトを処理するときは、またそれ中心の空間を別途持ち出してきて考える（他のオブジェクトのことは互いに関与しない）
- 原点を中心の世界全体を回転させる.....（以下繰り返す）

なんでわざわざオブジェクトごとに世界を区切って、それ全体を回転させるの？ と思うかもしれませんが。でもこれは、数学的な背景がわかってくると自ずとそうせざるを得ないことが理解できるようになってきます。

そのあたりはネイティブな WebGL のコードを書くようになると必然的にさらに深く理解しなければならなくなりますので、ここではいったん、そういうものなのだと雑に理解してしまっても問題ありません。

# 回転処理の勘所まとめ

- 回転と移動は、どちらが先かによって意味が変わってくる
- three.js では 回転が常に先 に起こる
- 回転を加えると世界全体が一気に回転してしまう
- three.js では Object3D のインスタンスごとに細切れに回転を処理している

ちなみに、scale（拡大縮小）は回転や移動よりも前、最も早い段階で適用されるのが一般的です（つまり拡大縮小 → 回転 → 平行移動の順）

なお、ここでしたような移動・回転・拡大縮小などの効果は、より厳密には行列と呼ばれる概念によって実現されています。

行列はとてもむずかしい数学の概念、みたいに思っている人が多いかもしれませんが次回もう少し深掘りする予定です。以下のブログ記事は図解も多く非常にわかりやすく読みやすいので、もし気持ちに余裕があったら読んでみるとおもしろいかもしれません。

線形代数の知識ゼロから始めて行列式「だけ」理解する

**時計の秒針をどうやって実現するか**

さて、回転がやや複雑に作用することがわかりましたが、いかんせん、時計の針のような回転処理をどのように行えばいいのかはわかりにくいですね。

three.js は先述のとおり常に回転を先に行ってしまいます。これはもう、three.js のなかの処理の流れがそのように定義されているので、この原則そのものを変えることは基本的にはできません。

ここでポイントになるのは、three.js が Object3D のインスタンスごとに、細切れに世界を区切って回転処理を行っているという点です。

つまり、Object3D ごとに回転が行われるのなら、あらかじめ移動した状態のオブジェクトを別の Object3D のインスタンスで包み込んでしまい、それに対して回転を掛けてやればいいのです。

“言葉で書くとわかりにくいですが、HTML の入れ子構造や、PowerPoint などシェイプのグループ化をするイメージを持つとわかりやすい”

three.js には `Group` と呼ばれるクラスがあります。

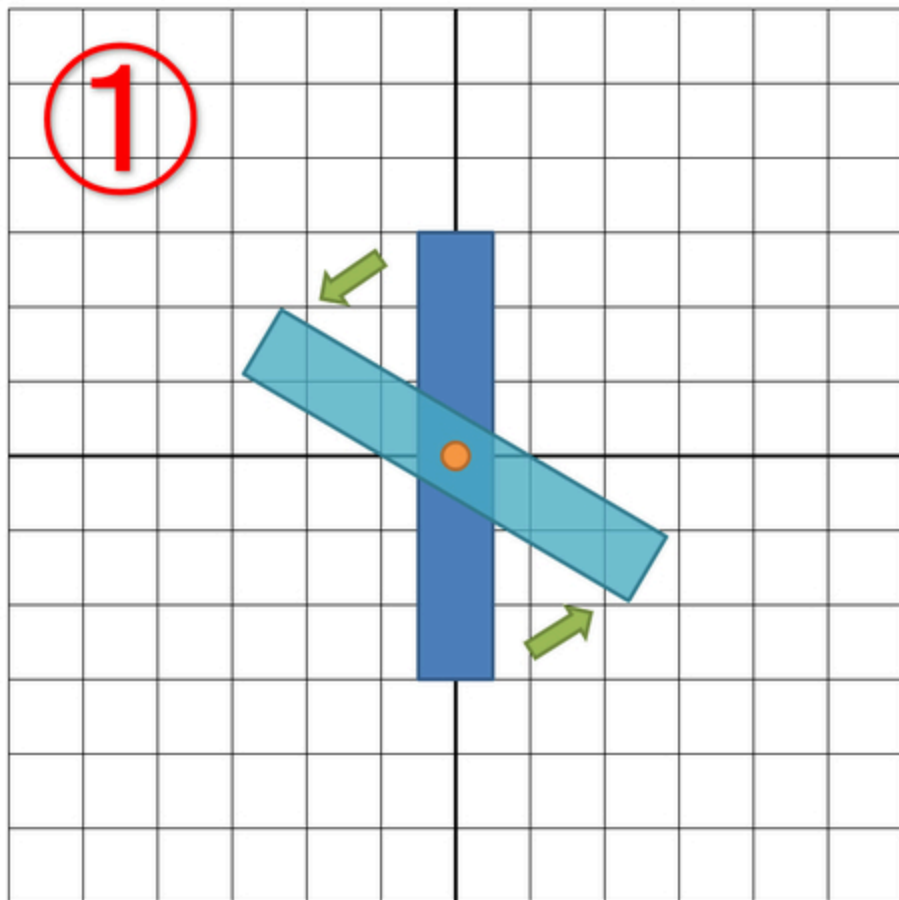
これはグループ、という名前からもわかるとおりオブジェクトをグループ化してひとつの `Object3D` のインスタンスとして扱うことができるクラスです。



そして、グループ化が行われた場合、そのグループ内では世界が閉じられ親子関係が常に維持されます。

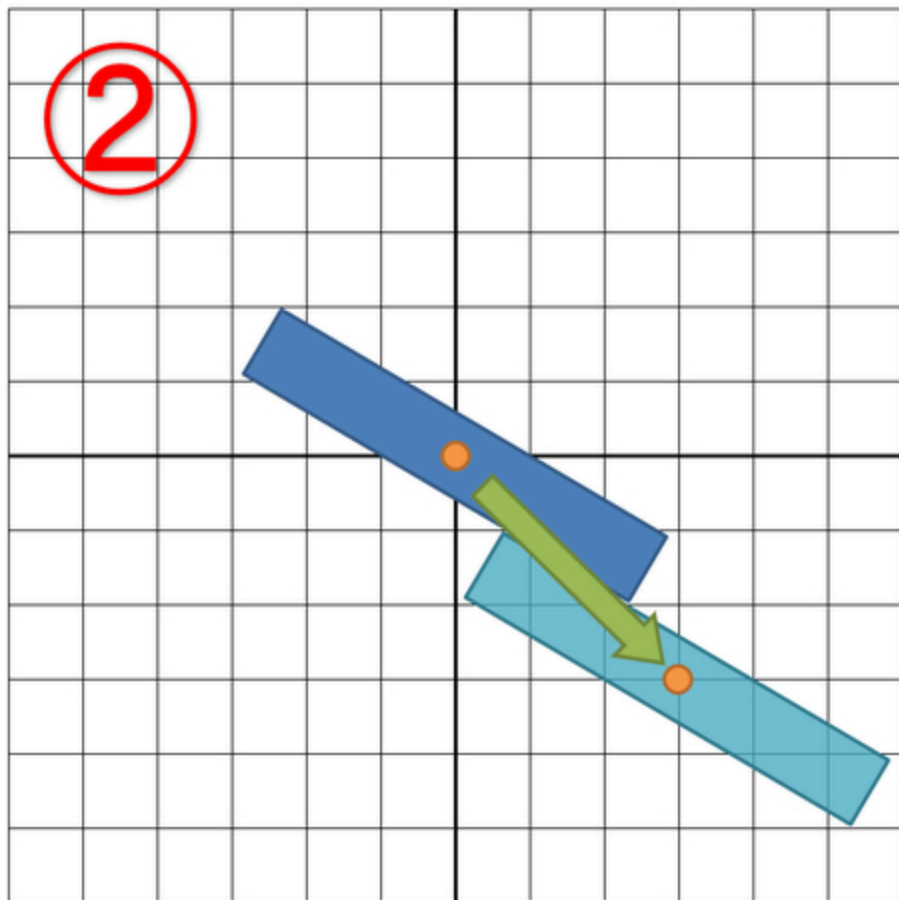
言い換えると、たとえば「`position` で移動したオブジェクトが含まれたグループ」を回転させると、先に移動が行われた状態に対してグループ全体が回転する、という処理を実現できます。これは図解しながら考えてみるとわかりやすいでしょう。

まずはグループを使わない場合の回転のおさらいから



three.js では、普通にオブジェクトを回転させると Object3D ひとつひとつに対して回転が先行して行われるので……

常にオブジェクトの中心位置（オレンジ点）を軸にした回転処理が行われる。※①



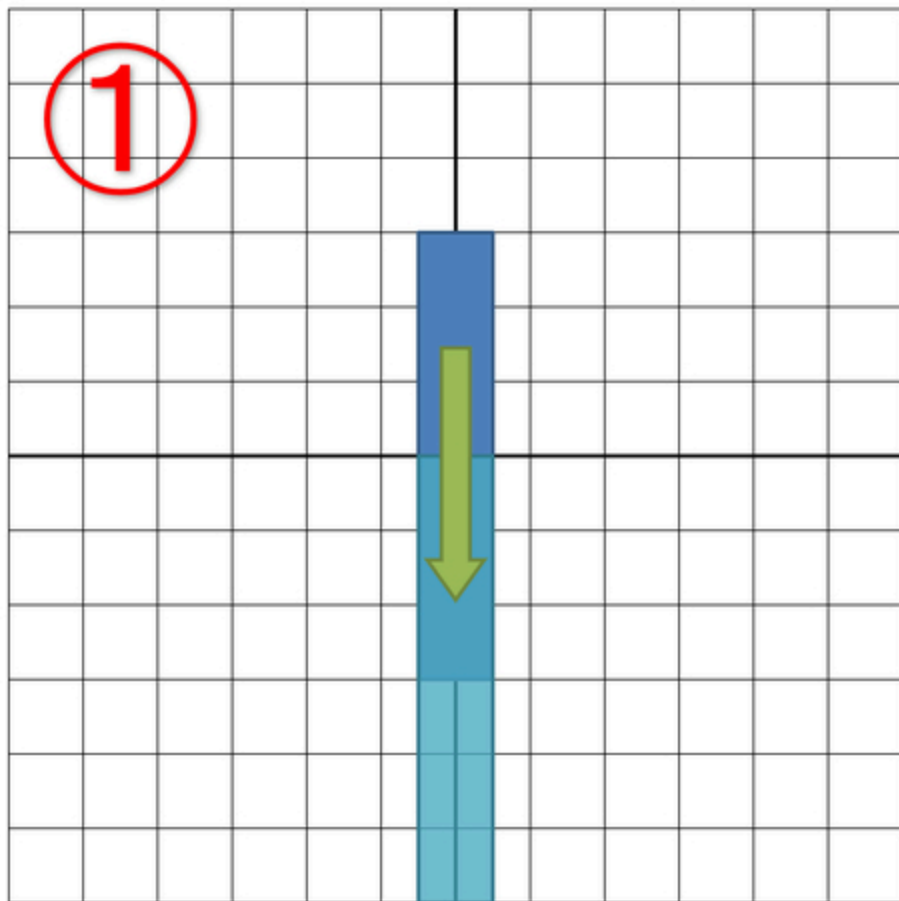
three.js では、普通にオブジェクトを回転させると Object3D ひとつひとつに対して回転が先行して行われるので……

常にオブジェクトの中心位置（オレンジ点）を軸にした回転処理が行われる。※①

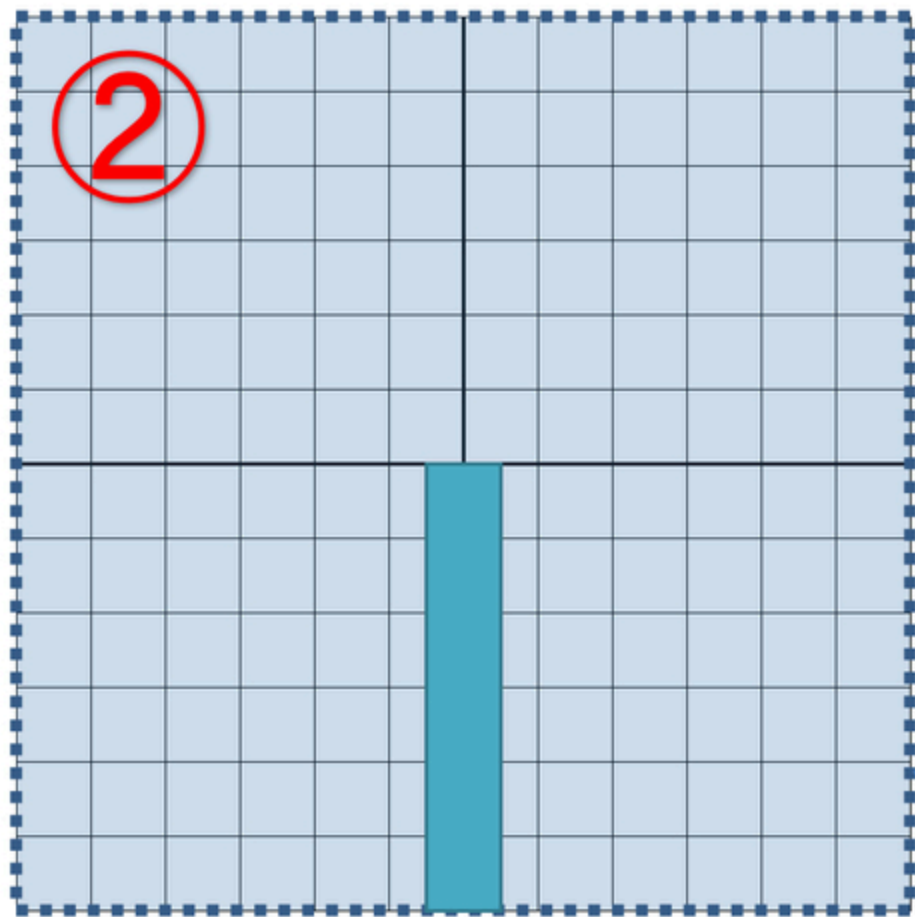
移動は、回転後に行われる。※②

ここまでが、これまでの挙動です。three.js の原則どおり、先に回転が行われた場合ですね。

では、次に `Group` を導入した場合はどうなるのでしょうか。



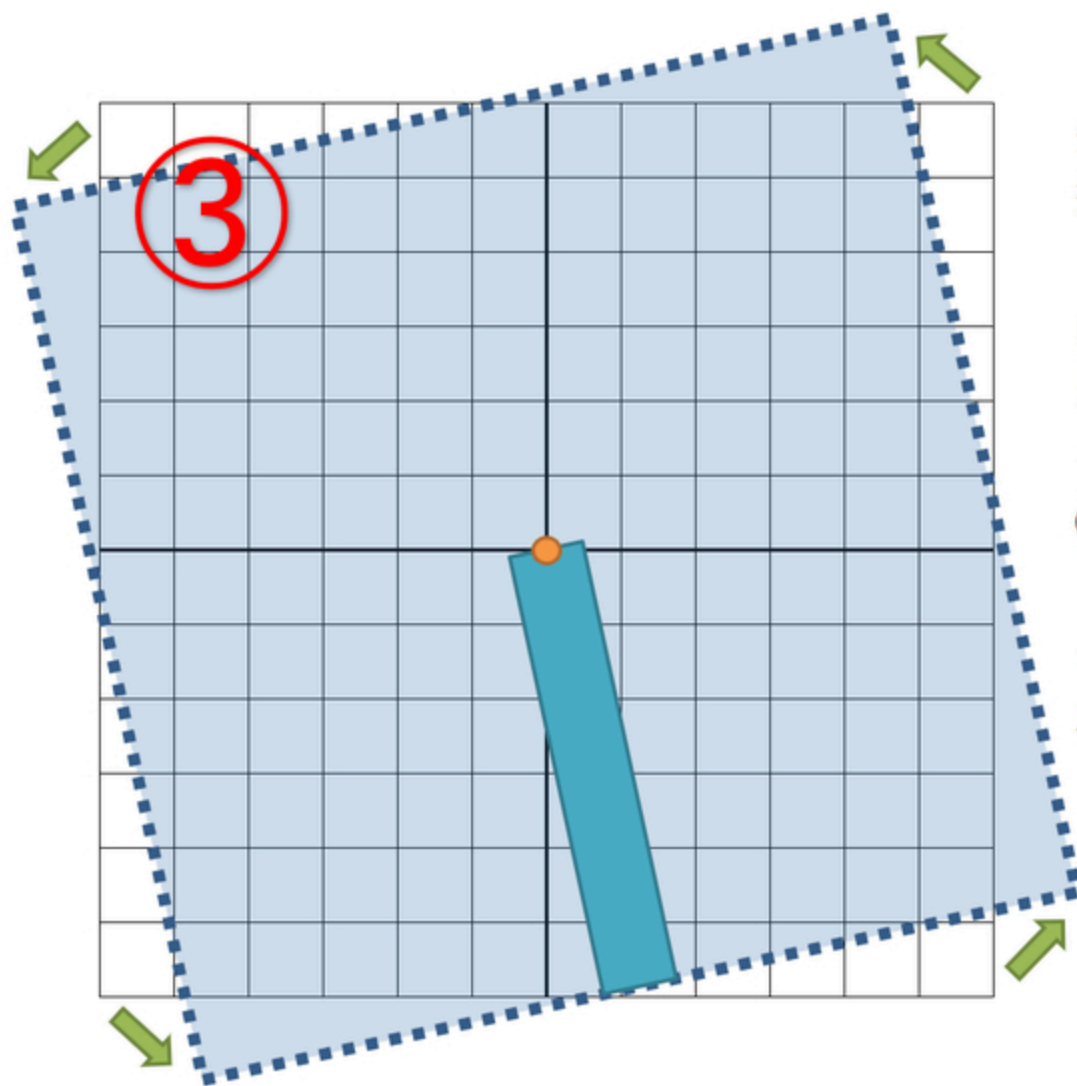
先にオブジェクトの position プロパティを  
変化させ、移動させておく。※①



先にオブジェクトの position プロパティを  
変化させ、移動させておく。※①

目には見えない（スクリーンには出ない）  
が、グループを作り、そこにオブジェクト  
を加えてグループの一員にしてしまう。※  
②

Group



先にオブジェクトの position プロパティを  
変化させ、移動させておく。※①

目には見えない（スクリーンには出ない）  
が、グループを作り、そこにオブジェクト  
を加えてグループの一員にしてしまう。※  
②

グループ全体をひとつのオブジェクトに見  
立てて回転させる。※③

さあどうでしょうか。先程、回転は世界全体を一気に回してしまう、という話をしたときにすんなりとそれをイメージできた人は少なかったと思います.....

このグループ化されたオブジェクトの挙動を見ると、回転は注目している対象の世界全体に対して行われる、ということがイメージしやすいと思います。



実際に、グループを使ってオブジェクトを包み込み、グループ単位で回転させた例が以下のサンプルです。

回転サンプル [その2](#)

three.js に限らず、また 3D か 2D かにかかわらず、座標を回転させるような処理を行う際には、今回見てきたような回転処理の原則が常につきまといます。最初は、すごく不思議な感じがすると思います。（私はしました）

でも、大事なことは結構シンプルにまとめることができます。

“ 次のページにある回転の原則はぜひ覚えておきましょう ”

# 回転の原則

- 3D プログラミングにおける回転とは、世界全体を回転させること
- 回転は常に世界の中心（原点）で行われる
- 移動してから回転するのと、回転してから移動するのでは、意味がまったく違ってくる
- 3D では一般に、回転を先に行うことが多い
- three.js でも、やはり回転が先に処理される
- three.js で移動を先に行いたい場合、グループ機能を活用する

ここで得られた回転処理の原則をイメージしながら、グループを利用したサンプルのコードを見てみましょう。

グループは、シーンと同じように `add` というメソッドを持っていますので、これを使ってオブジェクトをグループに加えていきます。

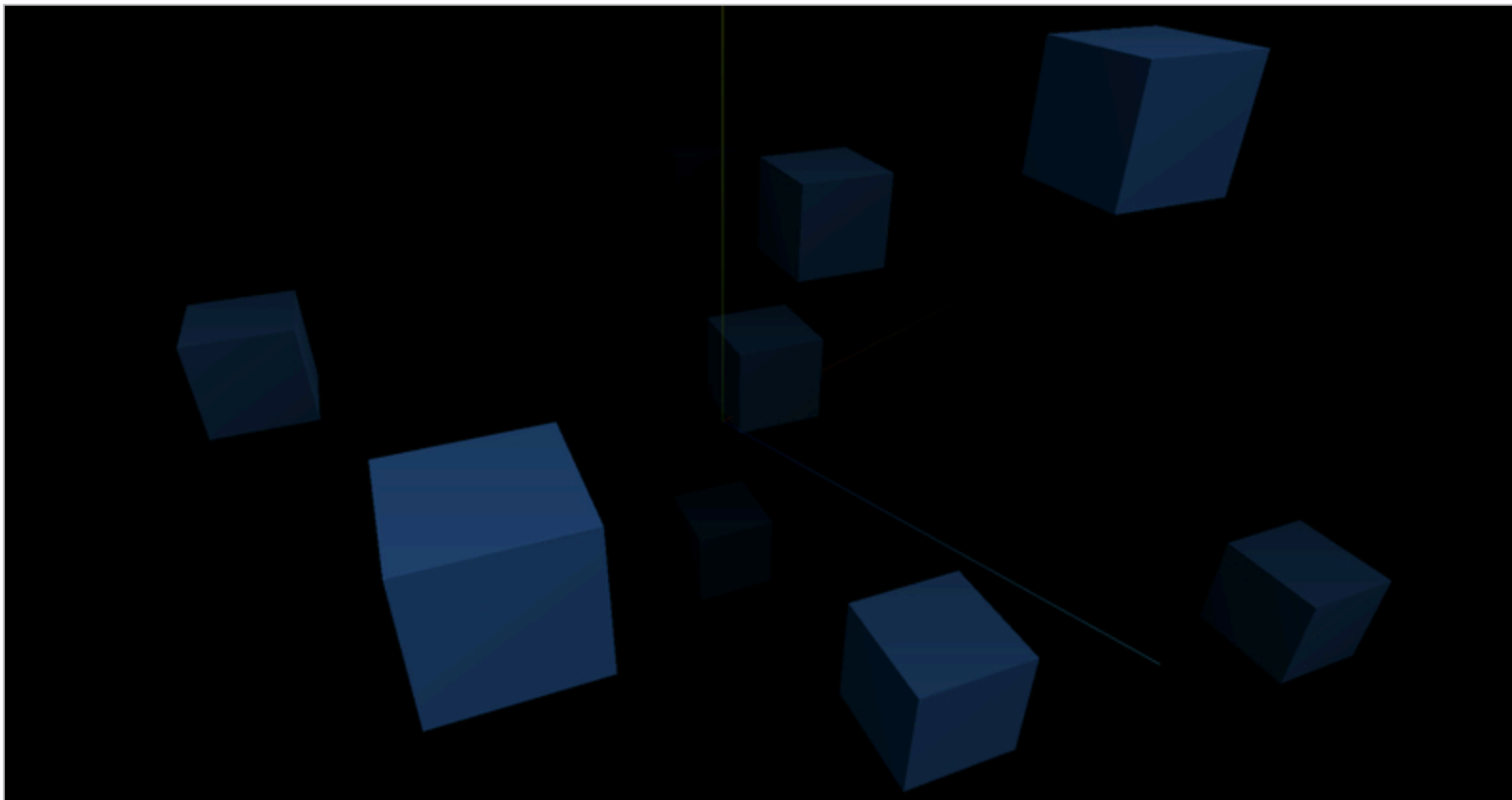
# 011

- グループをシーンのように見立ててオブジェクトを `add` していく
- グループに加えられたオブジェクトは、その時点での状態でグループ内に固定される
- グループそのものをシーンに対して加えることでグループ全体が描画される
- グループに対して行われた回転でグループ全体が回転（や拡大縮小・移動）する

フォグ

続いて取り組むのはフォグです。

フォグは、直訳すると「霧」を表す言葉で、シーン全体に霧が掛かったような効果を実現することができます。



“ フォグの色は自由に設定できるので黒い霧とかもできます ”

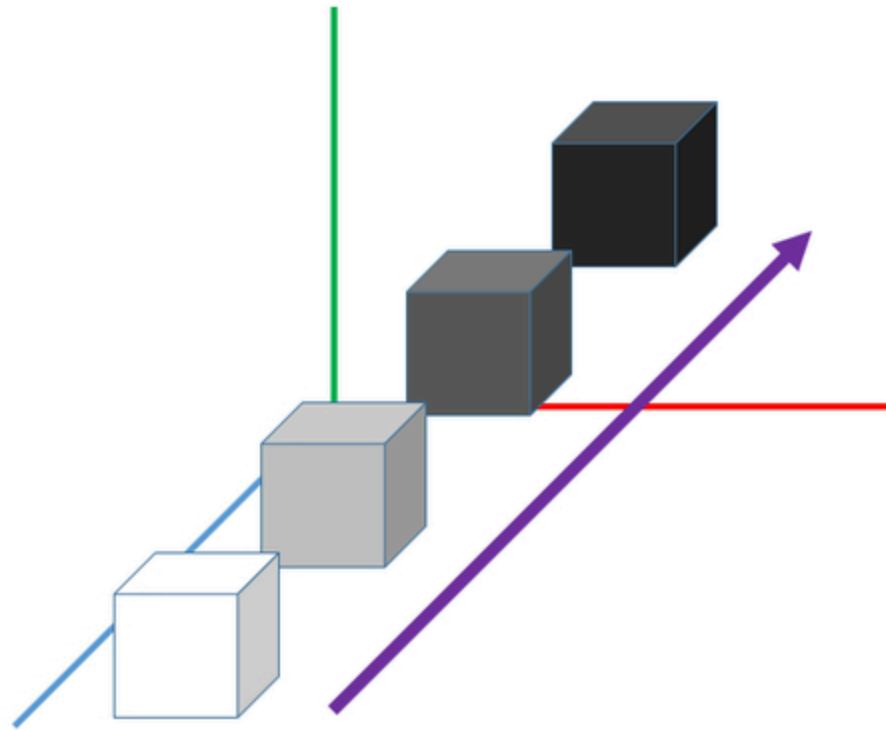


three.js では、フォグを用いるのは非常に簡単です。

フォグがどのような数学的背景で実現できるのかといった細かい話はまたの機会に説明しようと思います。まずは three.js ではいかにしてフォグを適用するか、その方法から覚えるのがよいでしょう。

# 012

- three.js ではシーンに対してフォグを適用する
- シーンの `fog` プロパティに `Fog` インスタンスを設定する
- フォグには、開始位置、終了位置、そして色のパラメータを用いる
- フォグが適用されるのはあくまでも 描画されるオブジェクトに対してのみ
- 自然に見せるには背景色とフォグの色を揃えておくこと



カメラからの距離に応じてフォグの適用される強さが変化

現実世界の霧とは違い空間に色が着くわけではない点に注意！

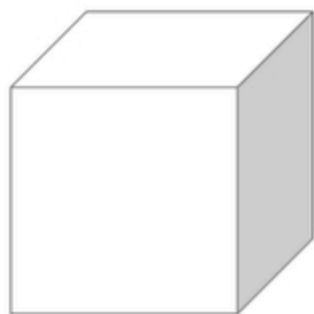
# テクスチャ

さて、どんどんいきます。

次は、テクスチャです。3D の文脈でテクスチャという言い方をした場合、大抵はそれは オブジェクトに貼り付けるビットマップ を指していることが多いです。

3DCG では、様々な形状を（three.js で言うところの）ジオメトリによって表現することができますが、その表面に複雑な模様や柄を貼り付け、より豊かな表現を得るためなどにテクスチャが用いられます。（もちろん他にもいろいろな用法があります）

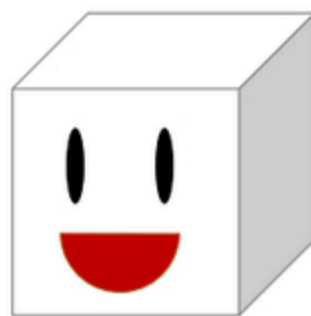
ゲームなどの場合は、服装や、キャラクターの表情のほか、地面や地形、建物などの色・柄なども、多くの場合テクスチャによって実現されています。



ジオメトリ  
(面を持つメッシュ)



画像などのビットマップ



面にビットマップが  
貼り付けられた状態

WebGL でももちろんテクスチャは使うことができ、テクスチャの元となるリソースには JPEG や PNG などの画像が用いられる場合が多いです。

ここであえて「多いです」と断定しない書き方をしているのは、必ずしもリソースは画像である必要は無く、動画データや Canvas2D で描画したビットマップデータなど、様々なものをテクスチャ用のリソースとして使うことができるからです。

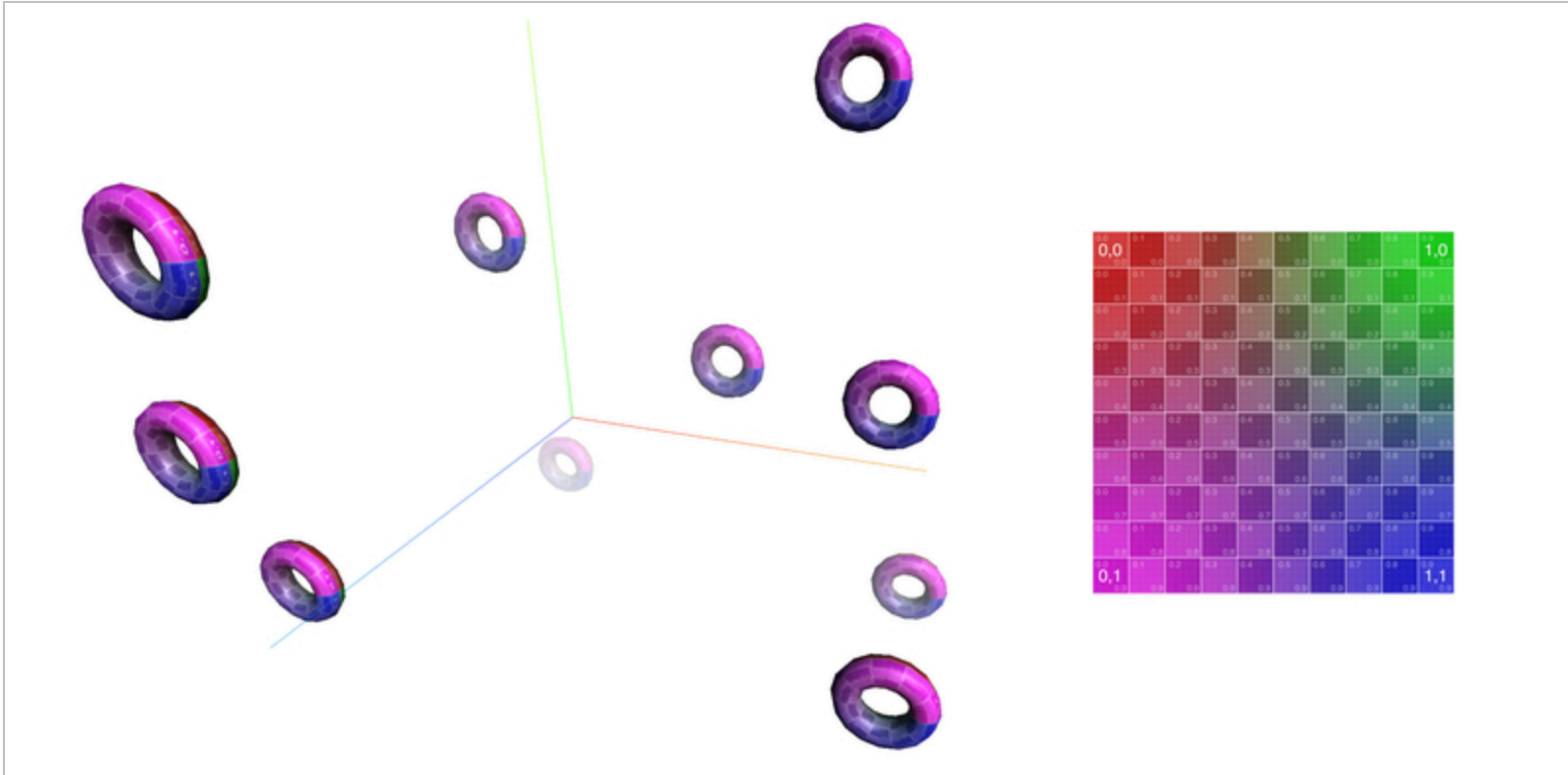


テクスチャのリソースとして使えるものの例。

- JPEG, PNG などの画像
- Canvas2D で描画したビットマップ
- 動画ファイル
- ウェブカメラから取得した映像
- WebGL でレンダリングした結果を再利用する
- バイナリデータから直接生成することもできる

とは言え、やっぱり一番ユースケースとして多いのは画像ファイルをテクスチャとして使う、というパターンです。

まずは基本中の基本となる、画像をテクスチャとして使うやり方を覚えましょう。



読み込んだ画像をオブジェクトの表面にテクスチャとして貼り付ける（マッピングする、とも言う）

テクスチャを使う場合、実は結構覚えなくてはいけないことが多いです。

ただ、three.js の場合はある程度そのあたりも考慮されて設計されていて比較的簡単にテクスチャを利用することができます。

# テクスチャと非同期読み込み

three.js を含む WebGL の実装では、テクスチャに画像由来のデータを利用する場合などに JavaScript が非同期で動作することを考慮する必要があります。

もう少し簡単に言えば、通常のウェブサイトなどがそうであるように「ロードが完了していない画像やデータは使えない」ので、非同期でデータがロードされるまでは、実行を待機するような処理を行う必要があるわけです。

three.js には、このような非同期処理に対応するため、ローダーと呼ばれるデータを取得するためのヘルパー実装があります。

テクスチャの場合も、やはりこのローダーを使ってやることで、簡単にテクスチャの読み込みを制御できます。

# 013

- ブラウザでは画像は非同期で読み込まれる
- ローダーを使うことで非同期処理に簡単に対応できる
- ローダーは、ロード完了と同時にコールバックを呼んでくれる
- 初期化したテクスチャはマテリアルの `map` プロパティに対して設定する
- そのマテリアルを設定されているオブジェクトにテクスチャが自動的に貼られる



# マテリアルをもう少し深掘り

テクスチャを使うことで、一気にシーン内に視覚的な情報が増え表現力がアップします。

せっかくテクスチャの話をしたので、ここでもう少し、three.js のマテリアルについて詳しい話をしておきます。

three.js に限らず 3DCG 一般の話として、透明度を扱う場合の難点について理解しておくことは重要です。

もしかしたらすでに、マテリアルに対して透明度を扱えるような設定を自身で行った事がある人もいるかもしれませんが、そのときなにかおかしいなと感じなかったでしょうか。

“ 実際にサンプル 014 をよく観察するとおかしい点に気が付くとおもいます ”



“なんかトーラスの内側が変な感じで見えたり見えなかったり.....”

この現象を正しく理解するには、CG の基本的な原理である「深度テスト」について理解する必要があります。

深度とは、カメラから見たときの深さ、つまり奥行きのことですね。

そもそも、3DCG では「さも当たり前のように物体の前後関係が正しく描画されます」が、これはどうしてなのでしょう。

たとえばキャンバスにシールをぺたぺた貼っていく様子を思い浮かべてみると、後ろに物がある状態をロジカルに描画するのって実は結構難しいのが想像できるのではないのでしょうか。



花や木のシールを貼ったがこれだけでは  
ちょっと寂しいので背景がほしいな.....



でもいまさら背景のシールを上から  
貼るわけにもいかない。詰んだ.....

あらかじめ前後関係を理解したうえで順番にシールを貼らないといけない

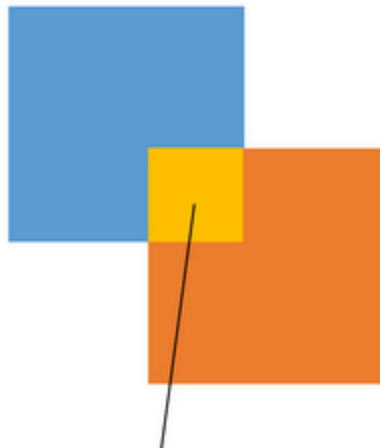
このような実はちょっとややこしい「物体の前後関係を正しく描画する」を実現するために、多くの CG では「深度テスト」が行われます。

ちょっと違った言い方をすると「深度テストに合格したピクセルだけが、画面上に描かれる」という仕組みを用いることで、三次元空間に配置されるオブジェクトの前後関係をさも正しく処理されているかのように画面に描画させます。





2枚の板、描画順は  
橙が先で、青が後



なぜこの部分は青く  
塗りつぶされない？



実は深度テストによって  
ピクセルが捨てられている

“ 深度テストに合格できなかったピクセルは捨てられてしまうが、それによっ  
て結果的に自然な描画結果となる ”

先程はシールを例に出しましたが、この深度の問題は「ジオメトリを構成するすべてのポリゴンひとつひとつ」について起こる問題なので、トーラスの一部は裏が透けるが一部は透けない、みたいなことが1つのジオメトリの中でも複雑に発生してきます。

これが、CGで透明度（半透明な質感）を扱うのが難しい理由です。

この問題は CG 一般でよく知られた基本的な知識なので、three.js ではこれに対応するために「オブジェクトの前後関係を並び替えて、奥にあるものから順番に描画してくれる」ようになっています。

ただ、さすがに「ジオメトリを構成する頂点の並び」をその都度リアルタイムに並び替えることはできないので、結局 100% すべてのケースで問題が起こらないようにはできません。

また、サンプル 014 をよく観察すると、板状のポリゴンが画面に映らなくなってしまう場合があることにも気がつくかもしれません。

これは「バックフェイスカリング」と呼ばれる機能により引き起こされる現象です。

バックフェイスカリングは、ポリゴンの裏面を非表示にしてしまうことで描画負荷を抑えることができる機能で、three.js に限らずあらゆるCG シーンで一般に使われる概念です。

地球から見上げた月の裏面って、絶対に目に見えないですよ？ だったら描画処理自体を省略しちゃったほうが効率がいいよね～という考え方ですね。



仮に青い部分が一切描画されていなくても  
どのみち目には入らない（どうせ見えない）

カメラから見て裏面になる部分をバツサリと「描画処理の対象外」にしてしまう

three.js ではバックフェイスカリングは既定で有効化されているため、ペラペラな板状のポリゴンは、裏から見たときに姿が見えなくなってしまう。

バックフェイスカリングの設定は three.js の場合はマテリアルに統合されており、`side` というプロパティに何を設定しているかによって振る舞いが増えます。

# 014

- 3DCG で透明度を扱う場合は罨がいっぱい！
- 深度テストの原理を想像しながら事象を観察する
- 基本的には後ろにあるものから描画する、しか解決方法は無い
- バックフェイスカリングの設定次第で面は消えることがある

CG 特有の難しい部分でありハマりポイントなので、まずは最低限の知識としてこういうことが起こるということ自体を知っておくことが大切



# ポストプロセス

さて今回は最後に、ちょっとだけ見た目がド派手になるやつをやって終わりにしましょう。

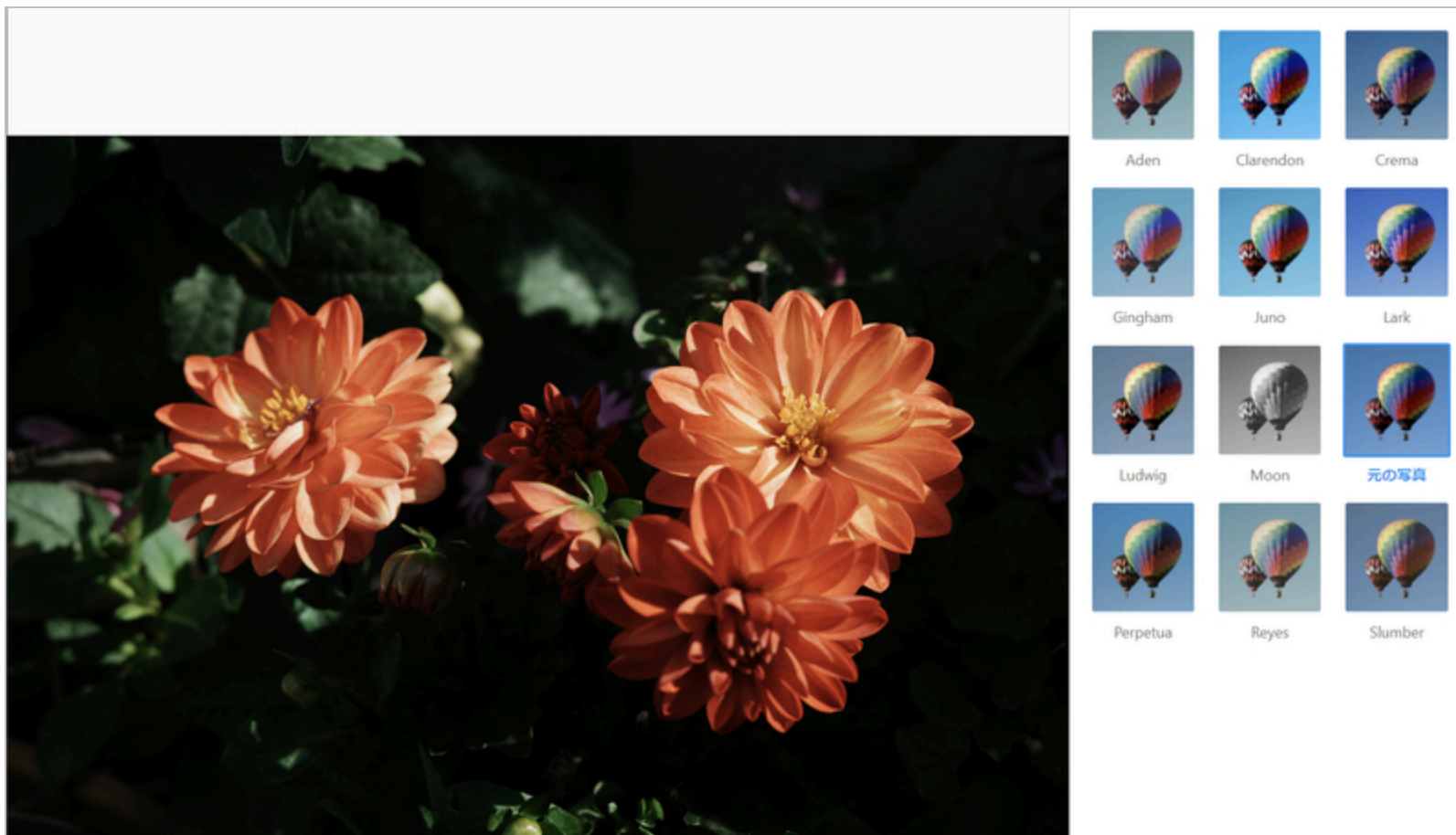
ここでは、ポストプロセスについて扱ってみます。













ポストプロセスというのは、簡単に一言で言ってしまうと 事後処理 のことです。

3DCG では、この事後処理ことポストプロセスは絵作りにおいて非常に重要な役割を持っています。

たとえば、カメラで撮影した写真画像を、photoshop などを使って加工したり.....といったことは結構みなさんも想像がしやすいと思います。

最近では、スマートフォンなどで撮影した画像をその場で加工できるアプリなどもありますし、事後処理を施して加工する、という行為そのものは結構身近ですね。

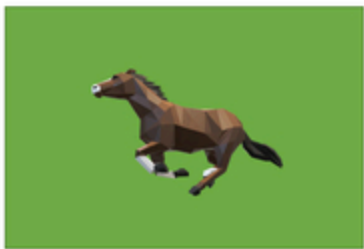


|   |   |   |
|---|---|---|
|  |  |  |
| Aden  | Clarendon   | Crema   |
|  |  |  |
| Gingham   | Juno  | Lark  |
|  |  |  |
| Ludwig  | Moon  | 元の写真  |
|  |  |  |
| Perpetua  | Reyes   | Slumber   |

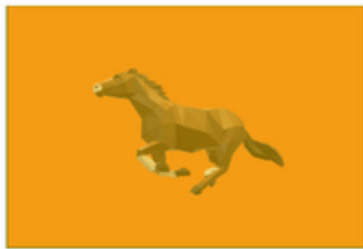
“ 実は結構身近な概念になっているポストプロセス ”

ポストプロセスとは、まさにこの 一度出来上がった絵を事後的に加工することを言い、3DCG の文脈では一度レンダリングした CG のシーンを、そのままその場でダイレクトに、かつリアルタイムに加工します。

そして、three.js はもちろんこのポストプロセスをサポートする機能を持っています。



まずバックグラウンドで  
CG をレンダリングする



一枚絵になった CG に  
何らかの加工を加える



加工後の状態を最終的な  
出力として画面に出す

“ その場でリアルタイムにフォトタッチするようなかんじ ”

three.js では、エフェクトコンポーザーを用いることでポストプロセスを比較的簡単に実現できます。

たとえば、まずは A というシーンを描画し、続いてその描いたばかりのシーン A を、B という方法で加工しながらスクリーンに描画する、といった描画順序に関する手続きをよしなに管理してくれます。



このとき、エフェクトコンポーザーには「パス（pass）」と呼ばれる単位で描画処理を指定することができ、パスを追加した順番通りに、連続してエフェクトが適用されていきます。

パスには、普通に 3D モデルを利用してシーンをレンダリングするパス、グリッチエフェクトを掛けるパス、ぼかすようなエフェクトを掛けるパス、といったように様々なパスがあります。

レンダリング



モノクロ化



グリッチ



特定の効果を持つ Pass を連続で適用していくイメージ

“ いくつかのキャンバスを次々経由していくような感じで出力結果が変化していく ”

サンプル 015 では、まず最初にそのまま（これまでどおり）シーンをレンダリングするパスである `RenderPass` と、グリッチエフェクトを掛けるパスである `GlitchPass` のふたつを使っています。

処理の順番を、落ち着いてイメージしましょう。

# 015

- エフェクトコンポーザーを導入して複数回描画する
- シーンのレンダリング結果をそのまま使う `RenderPass`
- その結果にさらに `GlitchPass` でグリッチエフェクトを掛ける
- どの段階の描画結果を画面に出すのか `renderToScreen` で指定

# 016

- コンポザーにパスをさらに追加
- エフェクト後にさらにエフェクト！
- 組み合わせれば表現は多彩になるがその分負荷は高くなる
- 処理順序にも十分注意しよう

さいごに

覚えることが大量に出てきた第二回講義ですが、今回はこのあたりにしておきましょう。

一度に全部を漏れなく把握することは誰にとっても難しいことです。講義の配信動画は何度でも見返すことができますので、焦らずひとつずつ、自分のペースで身につけていくことが大事です。

さて、今回の課題ですが.....

今回は、three.js の `Group` を駆使して「首振り機能付きの扇風機」を実現してみましょう。簡単そうに見えて、実は複数の回転を組み合わせる必要がある題材なので、グループ機能や回転を扱う練習としてなかなかおもしろいと思います。

“ 形状は適当でも構わないですし複雑じゃなくてよいので、とにかく「回転する羽」と「首振り」という2つの現象を再現してみましょう ”