three.js のさらなる活用

より理解を深めていこう

はじめに

前回は数学の基本を押さえる内容で、後半はちょっと難しかったかもしれません。

このあたりは１日勉強して即座に身につく、みたいなものではないので、じっくり取り組んでいきましょう。

さて今回は、three.js を利用して解説を行うところとしては仕上げ的な内容です。

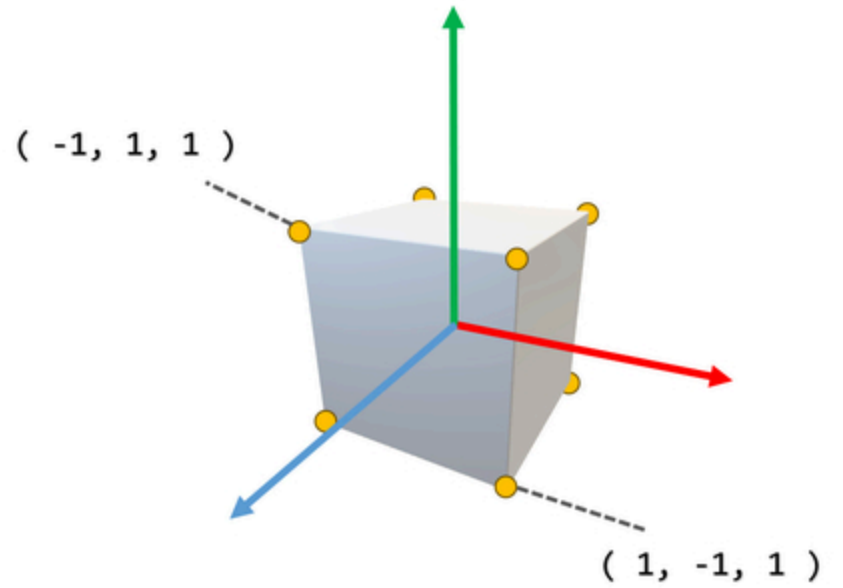
言い換えれば、three.js の基礎は今回の講義でほぼ網羅できた感じになると思います。今回もいろんな概念が出てきますが張り切って進めていきましょう。

3D シーンにマウスで干渉する

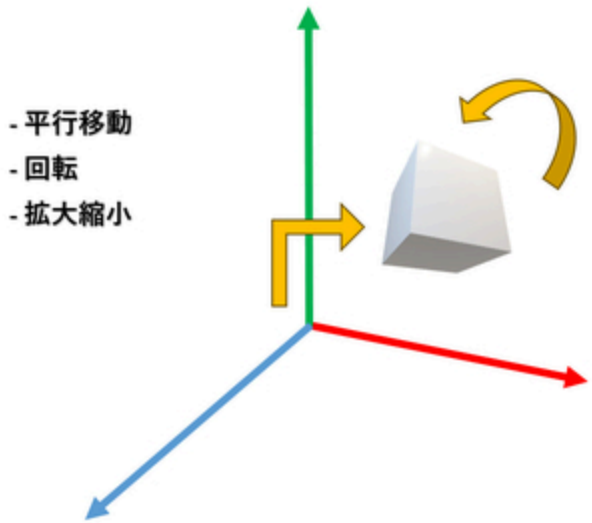
一般的な 3D 用語として、メッシュなどのオブジェクトが置かれている三次元空間のことを「ワールド空間」または「ワールド座標系」のように呼びます。

語感的にもわかるとおりで、まさに「オブジェクトが置かれている空間・世界」を表す三次元空間に対する呼称です。

> three.js 風に言えば Scene がワールド空間全体を表していると言えます



ローカル座標とは、そのジオメトリを構成する頂点がそれぞれ原点から相対的にどういう位置関係であるかを表した座標。
これはある意味設計図のようなものであり、このような三次元空間のことをローカル空間と呼ぶ。



一方でワールド空間とは、ジオメトリが実際に配置される世界そのものであり、ワールド空間上で平行移動や回転、拡大縮小などを行った後の、移動後の最終的な頂点の座標はワールド座標（またはワールド空間上での頂点座標）と呼ぶことができる。

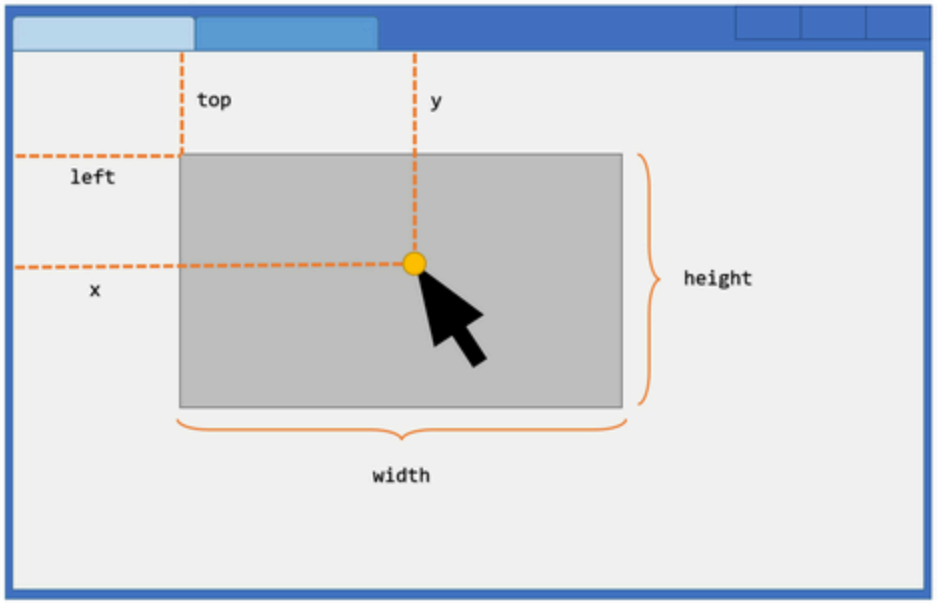
ローカル座標はいっさい変換されていない生の頂点座標とも言える

つまり three.js の Object3D が持つ `position` や `rotation` などを操作すると、オブジェクト（より正確にはそれを構成する頂点）が「ワールド空間上で動く」ということが起こっているわけです。

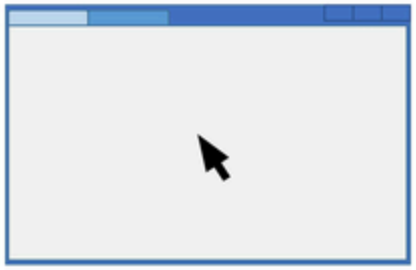
しかし、当たり前と言えば当たり前なのですが、最終的にワールド空間上にあるオブジェクトが描画されて一枚の絵になるとき、それが描かれるのは二次元の平面であるスクリーンです。

そうなると困るのが「オブジェクトをマウスカーソルでクリックしたい」といった場面に出くわした場合です。

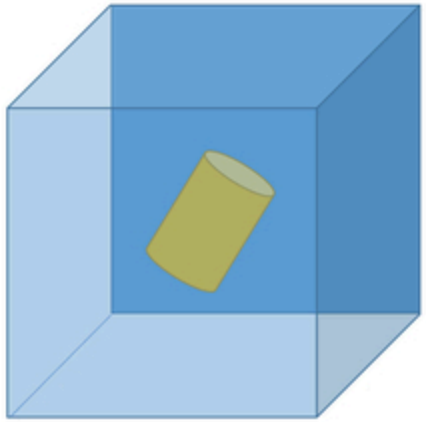
ワールド空間とスクリーン空間ではそれぞれ座標系が異なるため、いわゆる二次元平面上での衝突判定のようなシンプルな四則演算だけでは、両者の衝突判定を行うことはできません。



一般に、2D での衝突判定は、XY 座標や幅、高さなどがわかっていれば、比較的簡単な計算だけで求めることができる



二次元のスクリーン座標系



三次元のワールド座標系

それぞれはまったく異なる座標系なので、両者を対応付けするための
なにかしらの座標を変換する計算を行わなくてはならない

座標の概念自体がまったく別物

ここではワールド座標系とスクリーン座標系という対比で話をしましたが、基本的に 座標系が異なる場合は、座標系を揃えるような計算・変換処理 を行ってやらないと衝突判定を行うことはできません。

このような座標系の変換処理は、3D や、あるいは数学に慣れ親しんでいないとなかなか計算するのが難しい分野のひとつだと思います。

> ただし残念ながら、これら座標変換の概念がめちゃくちゃたくさん登場するのが 3D プログラミングでもあります…… 😂

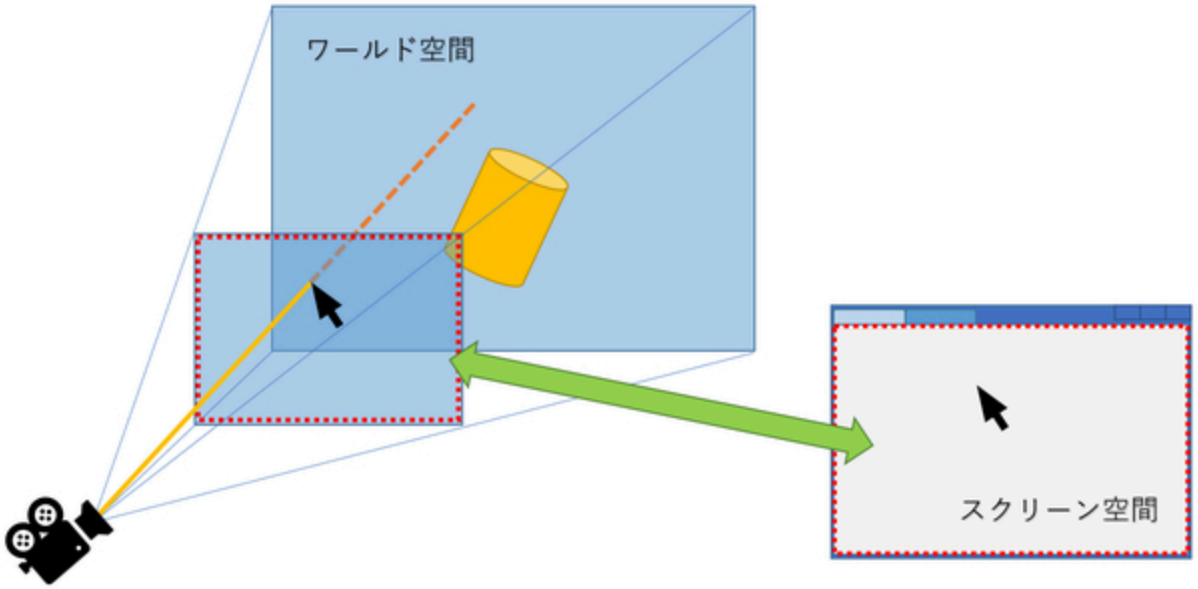
とはいえ、そこは three.js、きちんと「オブジェクトをマウスでクリックしたりする処理」を実現する方法が用意されています。

具体的には Raycaster（レイキャスター）を使うことによって、これを実現します。

Raycaster - three.js docs

レイキャスターは、マウスカーソルのスクリーン空間上での位置を元に、3D のワールド空間に対してレイ（Ray）を飛ばして判定を行ってくれます。

レイは直訳すると「光線」などの意味になりますが、文字通り、目には見えないレーザーのようなものを照射したと仮定して、そのレーザーとオブジェクトが衝突（交差）しているかどうかを調べてくれます。



三次元空間上に、マウスカーソルの位置を指標にしてレーザーをまっすぐ飛ばしてやり、オブジェクトがレーザーと交差するかどうかを判定するのが Raycaster のお仕事

three.js の Raycaster は、レイがオブジェクトと衝突した場合、そのオブジェクトや衝突した（ワールド空間上での）交点などを教えてくれます。

使い方もそれほど難しくないので、いつか自分の作成しているアプリケーションで「マウスで 3D 空間に干渉したい」というときのために使い方を把握しておくとよいでしょう。

024

- ウェブブラウザ上のマウスカーソルはスクリーン座標系に存在する
- 一方で、3D オブジェクトはワールド座標系に存在している
- Raycaster は、マウスカーソルの位置から逆算して……
- ワールド座標系にレイ（光線）を飛ばして交差判定を行ってくれる
- 判定結果から対象となるオブジェクトを調べることができる

独自のジオメトリを定義する

さて続いては、ちょっと原点回帰の意味も込めて three.js のジオメトリについてもう少し詳しく見てみましょう。

three.js のジオメトリには、球体やトーラス、ボックスなど、様々な種類があります。これら全てのビルトインのジオメトリは、元となる `THREE.BufferGeometry` クラスを継承・拡張して実装されています。

当然ながら、ビルトインのジオメトリと同じような状況を作ってあげれば、自分自身でジオメトリを組み立ててそれを利用することもできます。

自分でローカル空間上における頂点の座標を定義し、設計図そのものを作るわけですね。

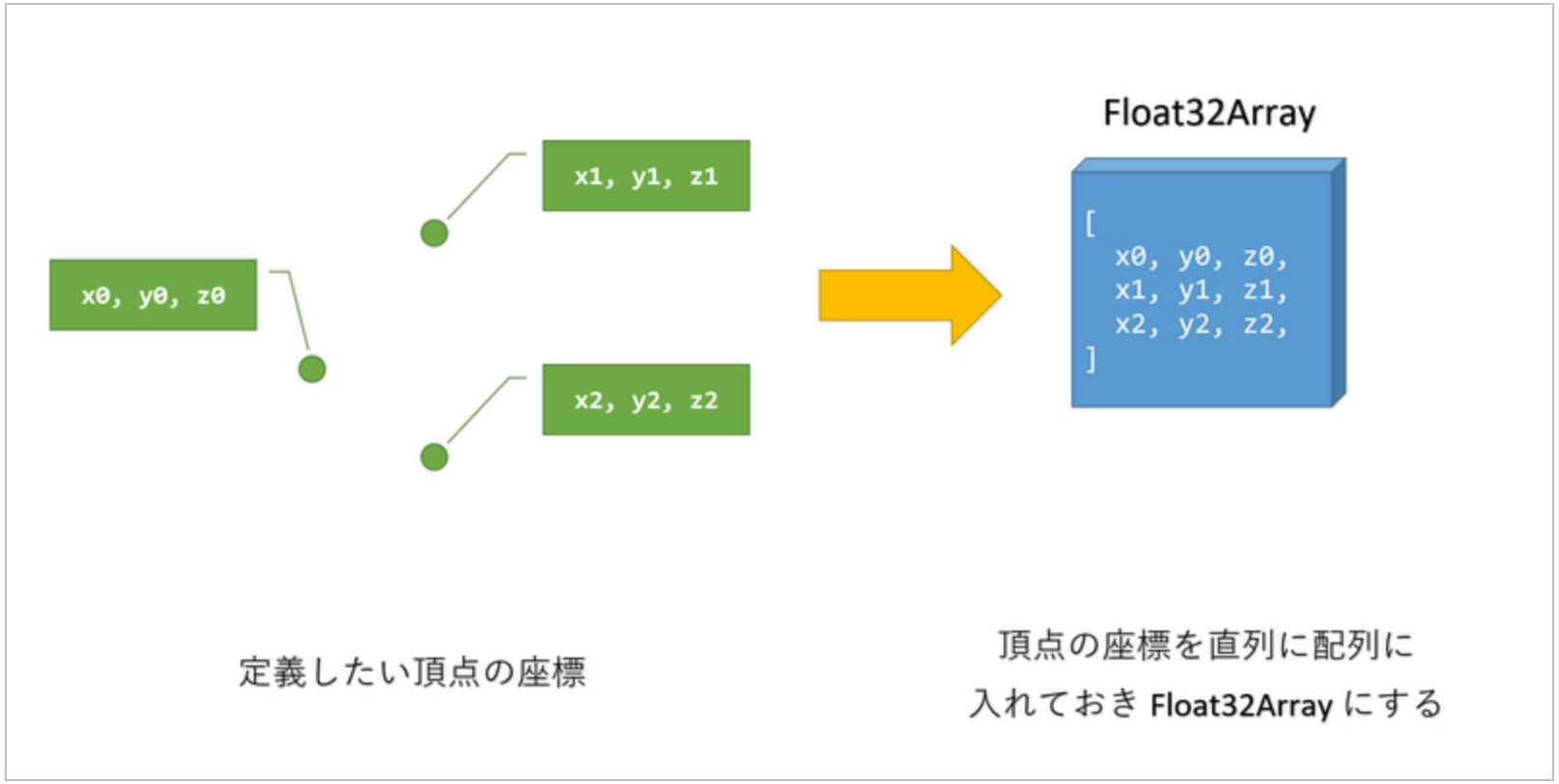
ジオメトリを自分で組み立てる必要性は、特に 3D に不慣れなうちにはあまり感じないかもしれません。しかし、自由自在にジオメトリを定義できるようになっておくとその応用範囲を大きく広げることができます。

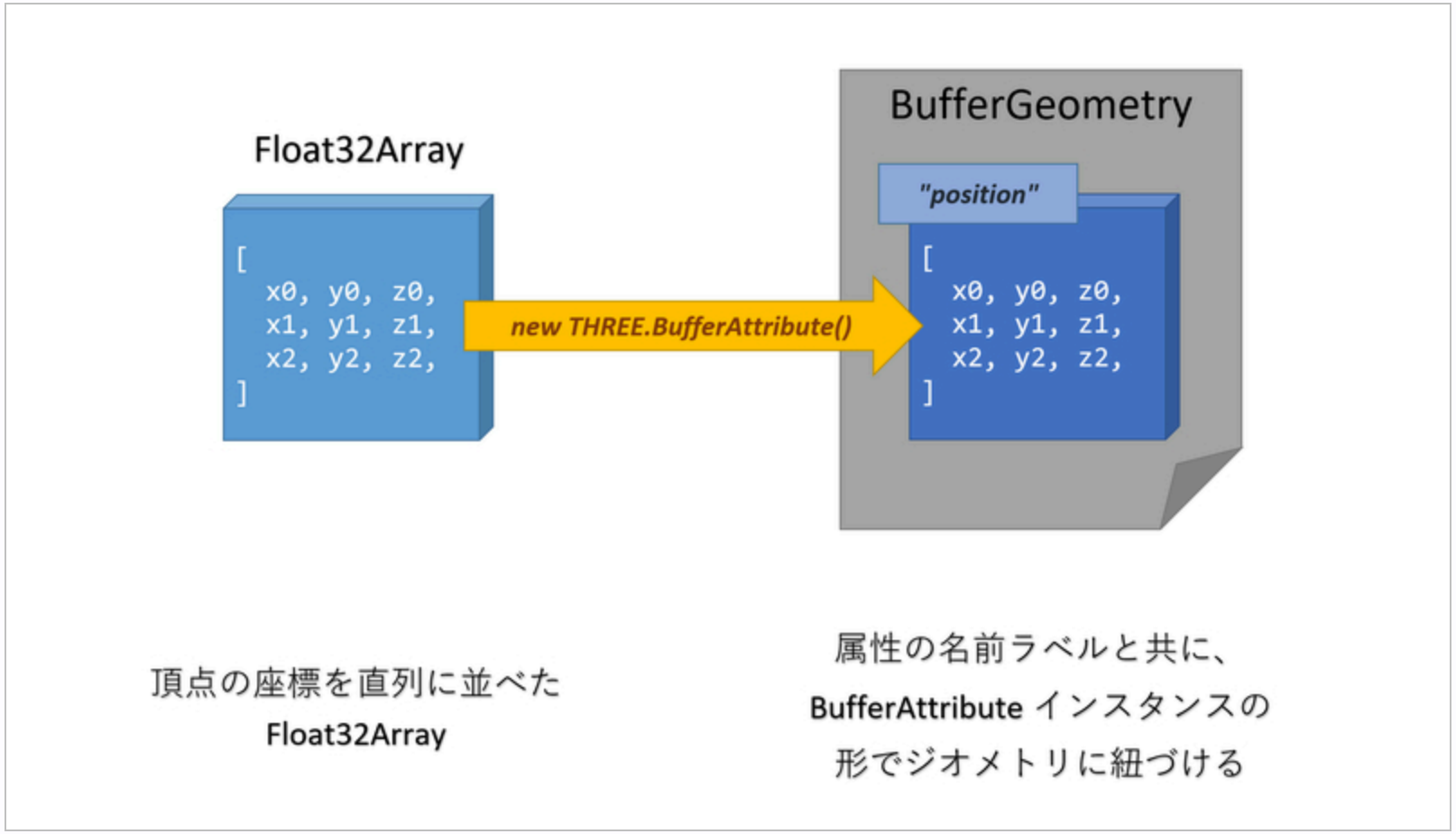
ここでは、まずはそれを簡単に体験する意味でシンプルなパーティクルの描画を行ってみましょう。

> ラインや面（ポリゴン）はちょっとだけ複雑なので、まずは点の描画からトライしてみましょう

three.js のあらゆるジオメトリの元になっている `THREE.BufferGeometry` クラスには、あらかじめどのような形も定義されていません。（ローカル空間における、いかなる頂点も定義されていない）

このジオメトリに、自分で定義した頂点をどんどん追加していき、それを点として描くことでパーティクル描画を実現してみます。


x0, y0, z0
x1, y1, z1
x2, y2, z2
Float32Array
[
x0, y0, z0,
x1, y1, z1,
x2, y2, z2,
]
定義したい頂点の座標
頂点の座標を直列に配列に
入れておき Float32Array にする


Float32Array
[
x0, y0, z0,
x1, y1, z1,
x2, y2, z2,
]
new THREE.BufferAttribute()
BufferGeometry
"position"
[
x0, y0, z0,
x1, y1, z1,
x2, y2, z2,
]
頂点の座標を直列に並べた
Float32Array
属性の名前ラベルと共に、
BufferAttribute インスタンスの
形でジオメトリに紐づける

025

- パーティクルとは粒子のことで、点として描くことが多い
- マテリアルに `sizeAttenuation` を設定すると点の大きさが自動的に調整され、遠近感のある描画結果が得られる
- ジオメトリを生成したら、配列を使って情報を定義していく
- 型付き配列とストライドを指定し `THREE.BufferAttribute` を生成し、名前と紐づけるかたちでジオメトリに割り当てる
- ジオメトリが定義できたら `THREE.Points` で点群を作る

> 点に遠近感が出るのは three.js が内部的に処理を行ってくれているからです

ポイントスプライト

three.js を使っている場合に限らず、WebGL や OpenGL などの 3D API で「頂点を点として描く」処理を行うと、点は正方形の姿で描かれます。

これはこれで、まあ悪くはないし使えないということでもないのですが…… せっかくなら、単なる四角形ではなく任意の模様や形状を表現したいと考えるケースが圧倒的に多いと思います。

そのような要望に対する１つの答えが「ポイントスプライト」です。その名のとおり、点（ポイント）として描かれる頂点を、スプライト（二次元のビットマップが付与されたオブジェクト）として扱ってしまおうというのがポイントスプライトの趣旨です。

three.js では、単純に「点に対して設定したマテリアルにテクスチャを割り当てる」だけでこれを実現することができ、とても手軽です。

> map プロパティにテクスチャを設定するだけ！

026

- 点として描かれる頂点にテクスチャを貼る
- あくまでも点なので、常にカメラに正対しているような見た目になる
- このような挙動（カメラに正対する姿勢を取ること）を指してビルボードと呼ぶことがある
- 正方形ではない画像を貼ると歪んでしまうので注意

> ビルボードはポリゴンでも実現することはできますが、ポイントスプライトはより手軽に同様のことを実現できる手段だと言えます

マテリアル設定再び

ポイントスプライトはテクスチャをマテリアルに設定するだけで手軽に実現ができましたが……

とは言え、このままだと見た目上ちょっとややこしい問題が起こってしまいます。

> 透明を扱う設定になっていないので、透明な部分が正しく処理されていない

以前もマテリアルと透明度について解説したことがありましたが、
3DCG における透明や半透明は、扱いが難しくハマりやすいです。

ちょっとおさらいすると、透明を扱いたい場合は「奥にあるものから描画されるように気をつける」などの対処をする必要があったのでした。

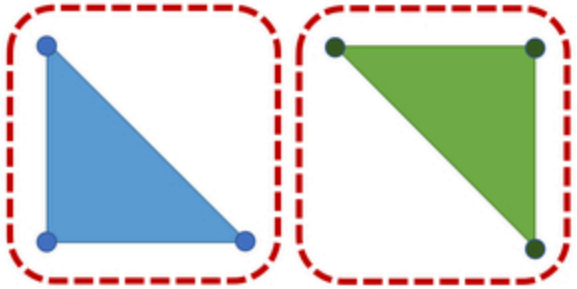
> three.js が自動的に Z ソートをしてくれる、という話も以前したかと思いますが、万能ではありません

パーティクル（を模したポイントスプライト）を今回のような手順で実装した場合、１つのジオメトリの中に大量の頂点がある状態なので three.js の Z ソートもあまり意味を成しません（Z ソートはメッシュなどの Object3D の単位で行われるため）。

かといって頂点を１つ１つ別のオブジェクトとして Scene に add してしまうと、CPU 側（つまり JS 側）でパーティクルの個数分ループを回すことになるので負荷が高くなってしまいます。

x 1

１つのジオメトリ（から作られたメッシュ）は、頂点が何個含まれていてもまとめて一度に処理される。



x 2

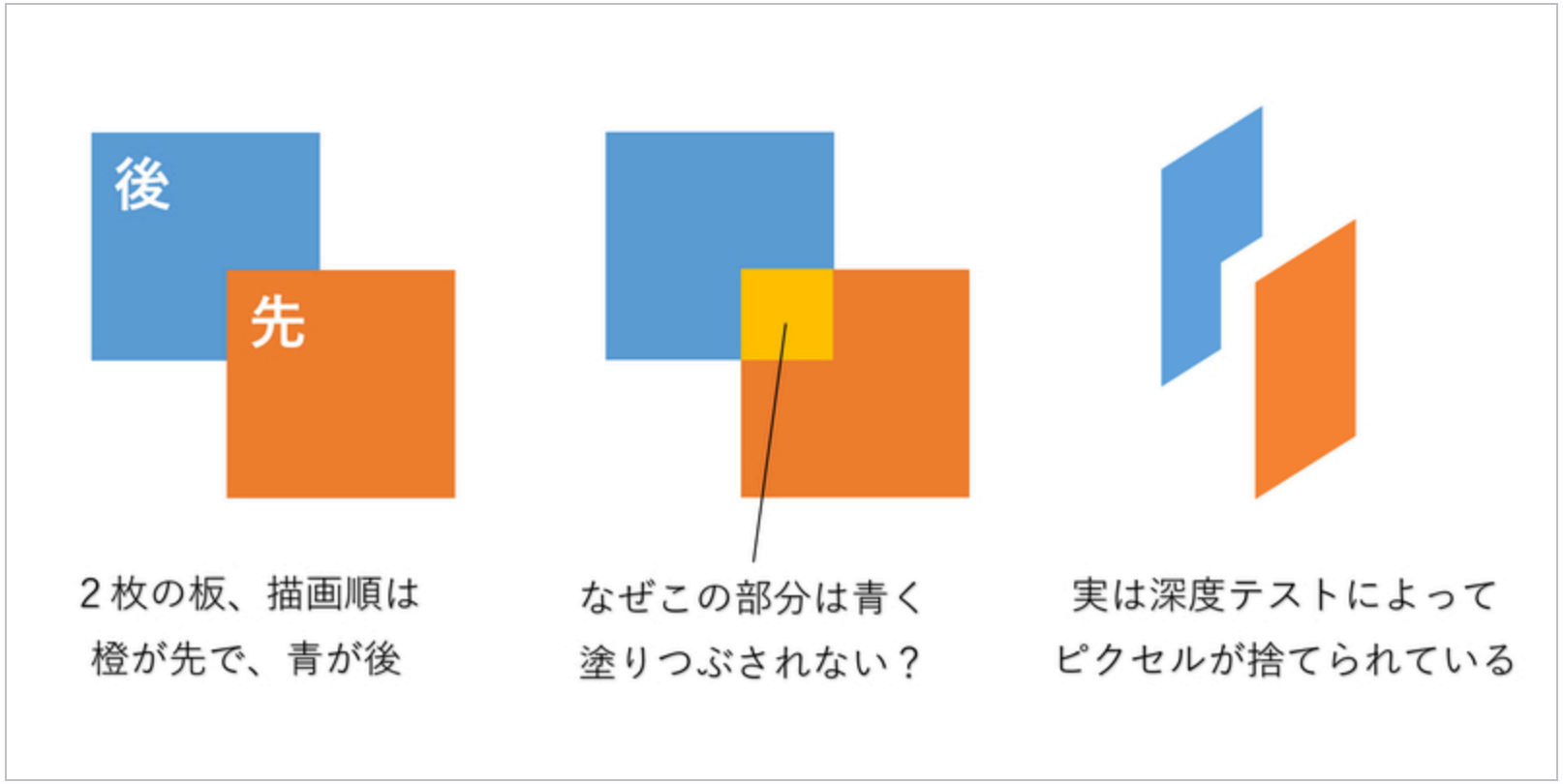
合計した頂点の数が同じでも、メッシュが別々の場合はそれぞれに対して順番に処理が行われる。

このような原理からわかるかと思いますが、１つのジオメトリにできるだけまとめて頂点を定義したほうがパフォーマンスは良くなります

ここでは、マテリアルの設定を以下のように指定することで、結果的に「そこそこよい描画結果」を得ています。

- 透明を扱いたいので `transparent: true` にし……
- さらに `opacity: 0.8` として透けさせる
- 色のブレンド方法は `AdditiveBlending` とし……
- さらに 深度テストを切る ことで すべてのピクセルを強制的に描画 させる

参考：Materials – three.js docs



深度テストについては第二回で扱ったこの画像のイメージを思い出しながら考えてみましょう。深度テストが有効な場合、ピクセル単位で描画すべきかどうかがチェックされます。

027

- マテリアルの世界は奥が深い
- 透明度の扱いなどは特に罠が多いので注意
- 原則として奥から描画するのが正解だが……
- ジオメトリ内の頂点を個別にソートしたりはできないので……
- マテリアル設定でうまく対応するしかないことも多い

> こうすれば絶対うまくいく、といった正解は無くて深度テストやバックフェイスカリング、シェーダなどを駆使して対応していくしかないので経験が求められる領域と言えます

glTF

three.js にはビルトインのジオメトリがあらかじめ多数用意されているため、それらの組み合わせによって多くのものを手軽に表現することができます。

しかし、ちょっと複雑な形状を作るとなると、幾何学形状の組み合わせだけでそれを実現するのが難しい場合が多くなってきます。

そこで、画像などを読み込むのと同じように、モデリングされた 3D データを読み込んで利用することができるようになっておくと便利な場合があります。

3D モデルのデータは一般に無償・有償問わず様々なものがあり、配布を行っているウェブサイトもたくさんあるので、ライセンスなどに気をつけながら配布されているデータを調達して使うことでよりリッチな表現を行うことができます。

> もちろん自分でモデリングする、というのも一つの手です

3D モデルデータの配布・販売サイト例

- Sketchfab - The best 3D viewer on the web
- 3D Models for Professionals :: TurboSquid
- Free3D.com

> もちろんこれですべてではないですが、代表的なもの

このような 3D モデルデータを配布しているサイトを見ていると気がつくかと思いますが、3D モデルデータの形式には様々な種類のものがあります。

obj, stl, fbx, max, 3ds, c4d, blend, dae... などいろいろあります。これらはウェブで例えるなら JPEG や PNG のような、いわゆるファイルフォーマットの違いです。

正直なところ、3D モデルデータのフォーマットにはベンダーの独自フォーマットなどが多数存在するので、かなり混沌としています。

そこで「3D モデルデータの中間フォーマットとして、統一された規格があったほうが便利だよね」というところから仕様が策定されているのが **glTF** です。

glTF の仕様は GitHub 上で公開されていて、three.js ふうに言うとジオメトリとマテリアルの情報を同時に持つことができ、さらにはモデルを取り囲むシーン全体の情報（カメラやライト）も定義することができるよう仕様が決められています。

近年の WebGL 実装では glTF を用いることが多くなってきています。glTF は「ウェブにおける画像フォーマットで言うところの JPEG を目指している」ので、将来的にもこの傾向が続くと考えられます。

KhronosGroup/glTF: glTF – Runtime 3D Asset Delivery

028

- three.js には glTF 用のローダーがあるので手軽に試せる
- 読み込みからパースまで全部ローダー側でやってくれる
- パースした結果のオブジェクトは `scene` プロパティを持つ
- `scene` プロパティ以下をシーンに追加すると、内包されているメッシュなどが描画され画面に出てくる

029

- アニメーションミキサーを利用するとモーションを再生できる
- 手順がやや複雑なので落ち着いて１つ１つ確認する
- アニメーションを複数含む gITF もある
- ウェイトという概念を用いてアニメーションを合成する

なお、glTF のファイルを開いて中身を確認したい場合は、Babylon.js のビューアを使うと機能も豊富で使いやすいです。

Babylon.js Sandbox - View glTF, glb, obj and babylon files

glTF 形式ではないファイルであっても、Blender などのツールを経由することで glTF 化できる場合があります。

また three.js にはそもそも glTF ローダー以外にもたくさんのローダーがあらかじめ実装されているので、無理に glTF 化しなくても良い場合もあったりします。そのあたりはリソースの状態や自分自身の Blender スキルなどと相談してアプローチを決めるのがよいでしょう。

余談：3D データを圧縮する Draco

glTF は大量の頂点を含んでいたり、容量の大きなテクスチャ用画像を含んでいたりするので、得てしてファイルサイズが大きくなりがちです。

これを圧縮して、よりウェブで使いやすいサイズ感に変換・復号する技術に、Google が開発している Draco があります。

google/draco

Draco は非常に圧縮効率が良く、驚くほどデータが軽くなりますが利用にはいくつか留意しておくべきことがあります。

場合によっては、むしろ導入を控えたほうがよいケースもあります。

効率よく圧縮される対象

Draco では、頂点の定義（座標など）が最も効率よく圧縮されます。つまり、定義されている頂点が多ければ多いほど圧縮効果は高くなります。

一方で、頂点の数自体が少ない場合は、あまり圧縮されないと考えたほうがよいでしょう。

Draco 自体の容量

Draco は原理上、あらかじめ Draco で圧縮した状態の軽いファイルをブラウザで読み込み、それをブラウザ上でデコードしてもとに戻す、という挙動をします。

このとき、デコードを行うためのデコーダが必要で、これが約 700KB ほどあります。せっかく圧縮してデータを軽くしているのにデコーダ自体がそれ以上に容量を食っている、といったことにならないように気をつけましょう。

ドロップシャドウ

さて、続いてはドロップシャドウについて見ていきます。ドロップシャドウとは、地面に落ちる影、のことです。（日陰などの「陰」ではなく、落ちるほうの「影」）

three.js では影を落とすためにいくつかの設定が必要になりますが、自前で影を実装することに比べるとかなり簡単に実装できます。

> ネイティブで影やるのは大袈裟じゃなく１００倍くらい大変……

three.js ではレンダラーを始めとするいくつかのオブジェクトに対して「影を有効化する」という設定を行った上で……

「影を落とす設定」と「影を受ける設定」という２つの観点から、各種オブジェクトに対して設定を行っていきます。このあたりはちょっとややこしいのですが、サンプルで `@@@` コメントがついている場所を中心に確認しながら、１つ１つ設定していきましょう。

030

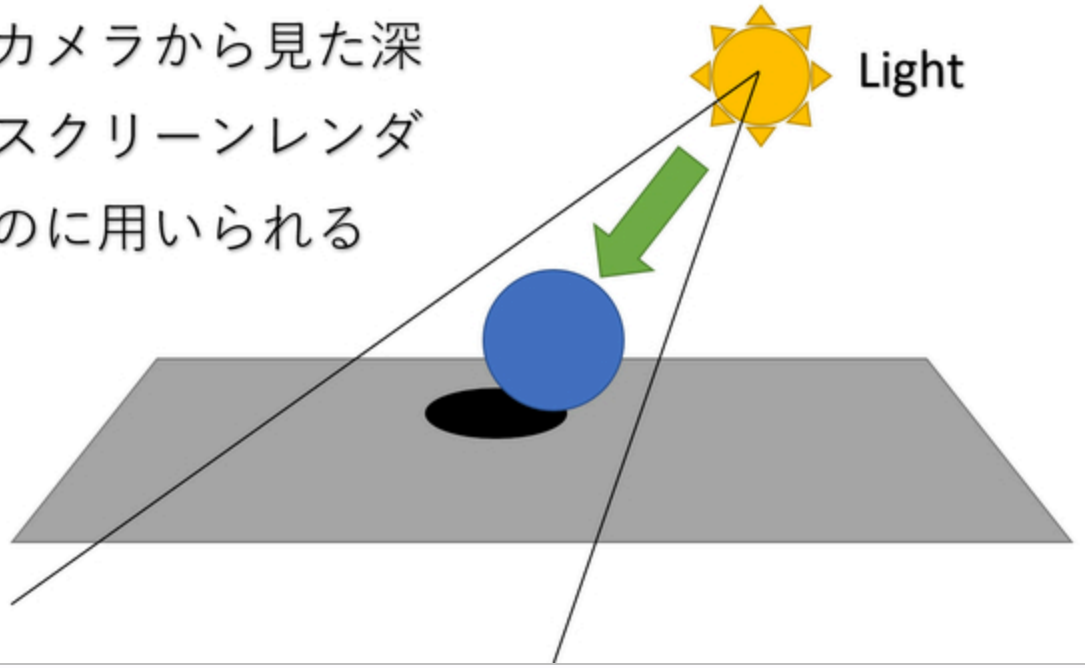
- レンダラーに対して影の有効化などを設定
- ライトに対して影を落とすよう設定
- ライトに内包されているカメラの設定に注目する
- 影用のバッファのサイズ次第で影のクオリティが変わる
- バッファサイズが大きいと負荷も比例して高くなる
- `Object3D.traverse` を活用して Mesh を操作する
- 影が落ちるのは影を受ける設定をされたマテリアルだけ

余談：影を描画する方法（深度影）

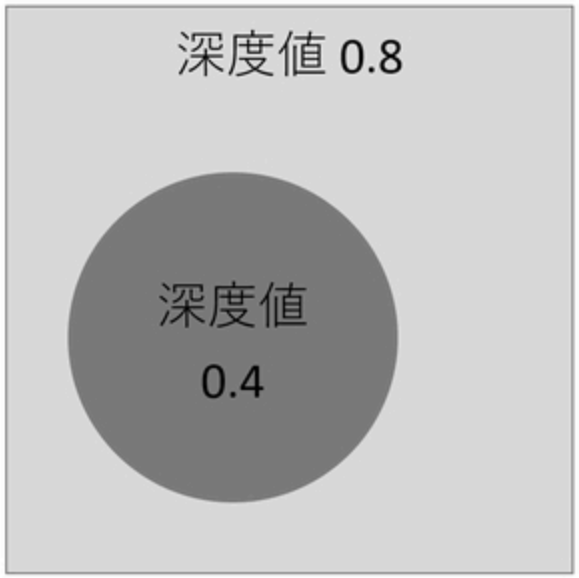
一般的な CG でよく用いられる影は、その手法から Depth buffer shadow、直訳すると深度バッファ影というように呼ばれます。（細かく言うといろんな手法の影描画方法があります）

その原理は、ライトのある位置から見える風景を写真に収めておき、実際に CG を描画するタイミングで写真と見比べることで、対象のピクセルの前に障害物があるかどうかを確認するというやり方です。

ライトにはカメラが内包されていてそのカメラから見た深度値をオフスクリーンレンダリングするのに用いられる



オフスクリーンレンダリングに関してはこのあと詳しく説明します



ライトに内包されるカメラ
から見た深度マップの状態

※ ここでは深度値は大きいほど
遠くにあるとして考えてみよう

このような深度を焼き込んだ描画結果は「深度マップ」と呼ばれ、これを参照しながら最終的なシーンを描画することで影を落とします。

深度バッファを用いたドロップシャドウの原理は、言葉や図解で説明されても、おそらく理解するというか腑に落ちた状態になるのにちょっと時間が掛かると言いますか…… 少し理解するのが難しい概念かと思います。

私も最初の頃、言っていることの意味がよくわからなくてすごく苦労しました……

three.js 的なお作法で言うと「影を受けると設定されたマテリアルにだけ影が落ちる」という仕様になっている理由は、原理から逆算するとわかりやすいかもしれません。

「深度マップを参照しながら深度値を比較し影を落とすかどうか判定する処理」は若干ですが負荷も上がりますので、影を受ける設定になっているマテリアルに対してのみ、限定的に行われるようになっているわけですね。

オフスクリーンレンダリング

現代の 3DCG に欠かせない技術に「オフスクリーンレンダリング」があります。

これは読んで字の如くで、スクリーンには映らないレンダリング、という意味があります。

以前の講義で EffectComposer を利用したポストプロセス（ポストエフェクト）について扱ったことがありましたが、あれも「オフスクリーンでレンダリングした結果に対して、加工した結果を画面に映す」ということを行っているため、オフスクリーンレンダリングを活用している技術の１つです。

このように、オフスクリーンレンダリングをうまく活用することでより柔軟かつ多彩な表現を行うことができるようになります。

過去の講義で取り上げたポストプロセスの概念図で言うと、左端の「バックグラウンドで CG をレンダリング」の部分がまさにオフスクリーンレンダリング。

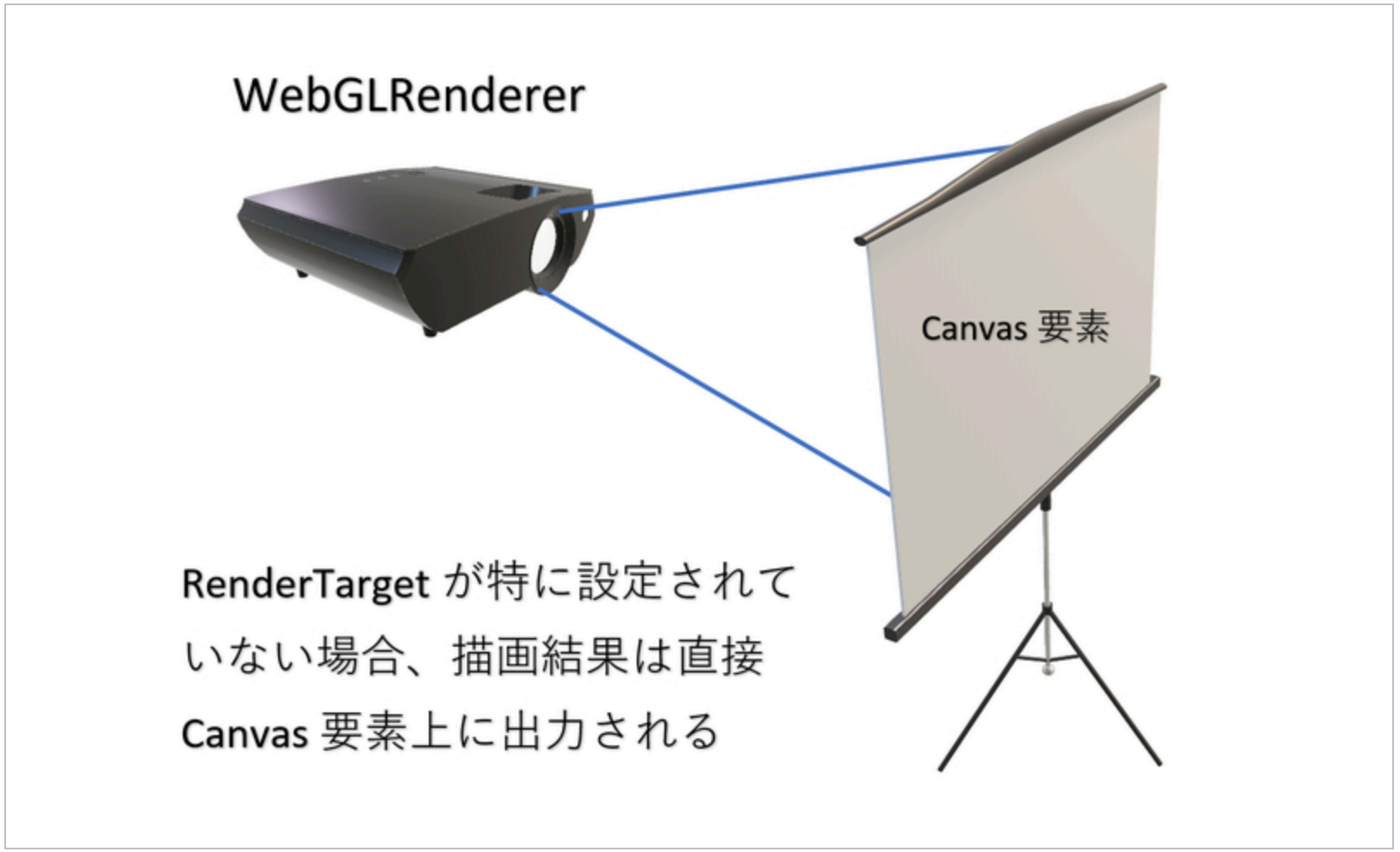
オフスクリーンレンダリングの主な用途

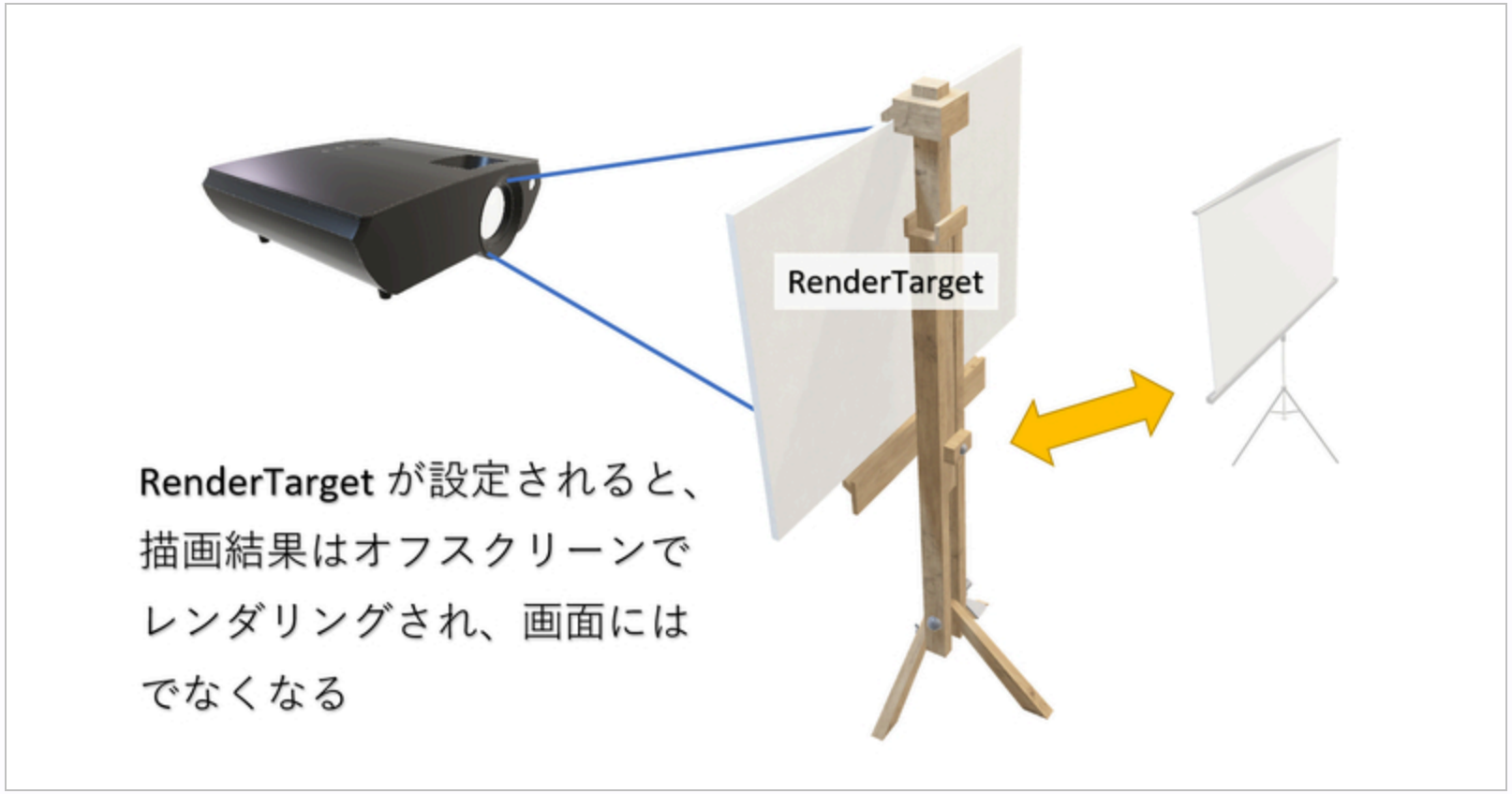
- 各種ポストエフェクト
- 鏡やツルツルした面への映り込みの表現
- ワイプのような表現
- 影を落とす処理
- 水や光学迷彩などの透明な質感の表現

> とにかく用途が多すぎて、あらゆる表現の下地となっている技術であるため
> 現代の CG には欠かすことのできない超重要な概念だと言える

three.js でオフスクリーンレンダリングを行うためには、
`THREE.WebGLRenderTarget` というオブジェクトを利用します。

今までにもずっと利用してきた `THREE.WebGLRenderer` というオブジェクトがありましたが、これに対して `THREE.WebGLRenderTarget` を割り当てることでレンダリング先が「画面から、割り当てられた RenderTarget に切り替わる」というイメージです。


WebGLRenderer
Canvas 要素
RenderTarget が特に設定されていない場合、描画結果は直接Canvas 要素上に出力される



仕組み上、Renderer に対して RenderTarget は１つしか適用できない。（より発展した技術に MRT がありますがここでは割愛）

031

- レンダラーに対してレンダーターゲットを適用することができる
- 単純に、出力先がそのまま入れ替わるようなイメージ
- レンダーターゲットに `null` を指定し適用を解除すればもとに戻る
- レンダーターゲットのサイズやカメラの設定に注意が必要
- レンダリングを行う手順が複雑になるので混乱しないように注意
- レンダリングされた描画結果はテクスチャとして取り出せる

> シーンやカメラは必ずしも別にする必要はないが、齟齬が生まれないように設定内容に注意

さいごに

今回まで全４回に渡り three.js を扱ってきましたが、three.js の持つ機能はまだまだたくさんあります。

機能を網羅的に把握していくのは大変ですが、総じて言えることとして「ネイティブな WebGL で実装することに比べたら大きく難易度・手間が軽減される」ので、本当に素晴らしいライブラリだと思います。

一方で、three.js が（良くも悪くも）隠蔽している部分にこそ、実は本質的な基礎・原則が潜んでいることも多くあります。

それらの知識を正しく身に着けていることで初めて実現できる技術があったり、むしろ理解が促進されたりすることも、実際あったりするんですよね。次回からは、そのあたりをじっくり取り上げていく感じになります。

さて最後に今回の課題ですが、Raycaster と Plane（板）を使ってなにか作ってみましょう。

たとえば「サムネイルをリンクのように扱う」であったり「画像のビューアを 3D で作る」といったなにかしらのテーマを決めると作りやすいかもしれません。以下、アイデアの参考に。

- CLOU architects