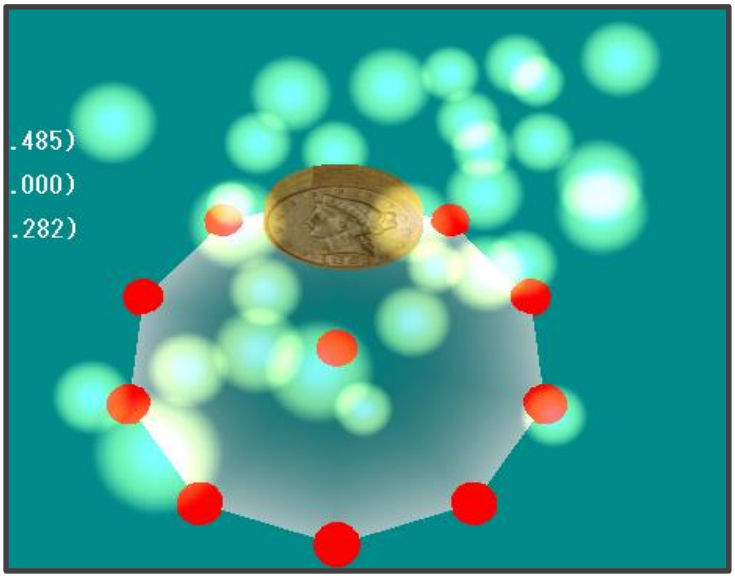
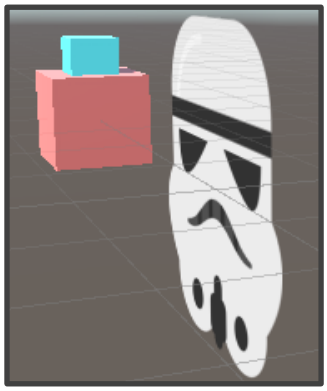
3Dでのエフェクト表現

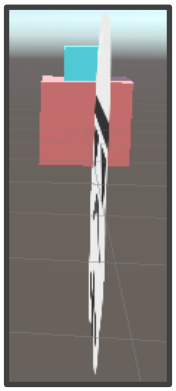
3D空間でエフェクトを表示する手法として、
今回は、ビルボードとパーティクルを題材にして作成していきます。



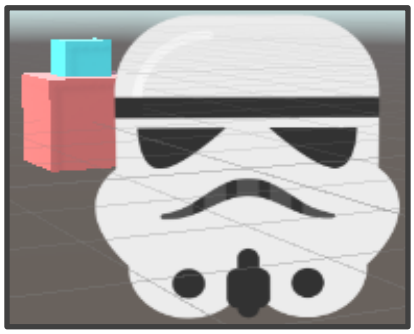
ビルボード

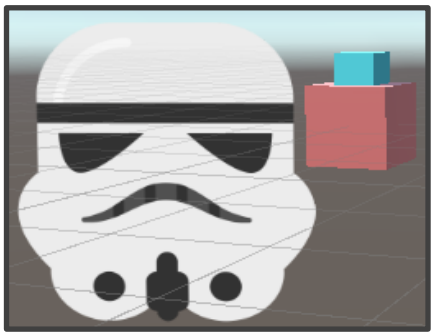
ポリゴンを、常にカメラに向ける技術のことです。





板のようなポリゴンに
テクスチャを貼り付け。
カメラを回転させて、
横から見ると、ペラペラ。

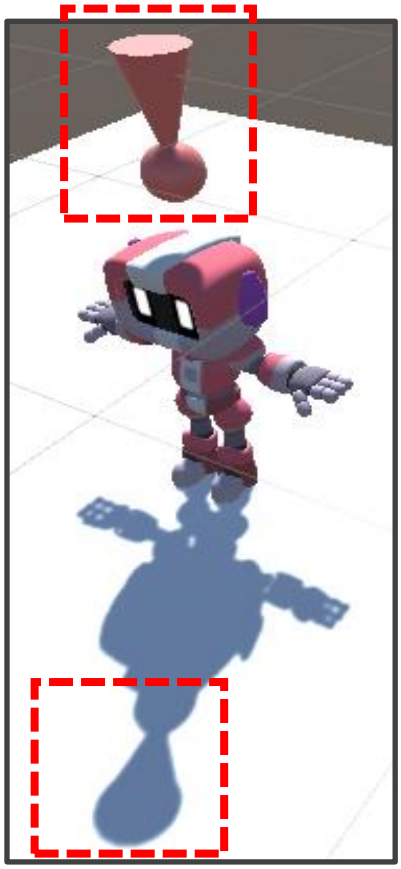


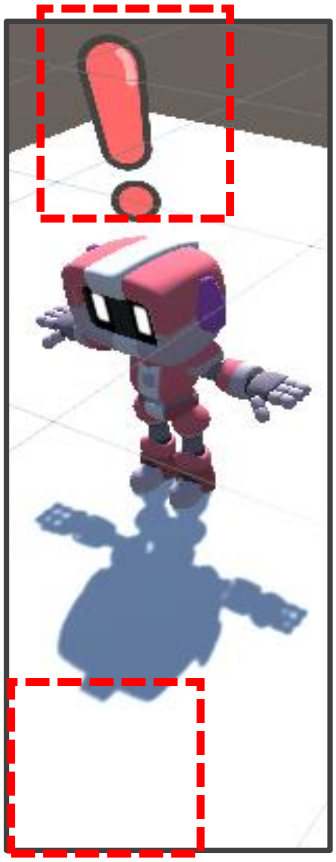


ビルボードを使用した描画。
ずっとコッチを見て欲しい時に使用します。

3Dよりも、2Dの方が処理負荷がかかりませんので、
3D空間での2D表示の実現だけではなく、負荷軽減のメリットもあります。

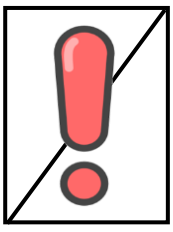
例えば。。。
キャラクターはもちろん３Ｄモデルで表示するとして、
そのキャラクターの感情を表現するためによく使われる感情アイコン。





ペラペラ。

どうしても３Ｄで表現したい場合は仕方がないのですが、
ビルボードの方は、板ポリになりますので、
どんなに複雑な画像であっても、頂点数は４つ。
(頂点インデックス使用時)
三角ポリゴンが２つ。
３Ｄモデルの計算量からすると、遥かに軽量です。



ビルボードを使う前に素の３Ｄ描画をおさらい

これまでは、３Ｄモデルのファイルを読み込み、
モデルを描画してきましたが、
ポリゴン情報をプログラム内で生成して描画してみましょう。

[Dxlib関数]
DrawPolygon3D
(VERTEX3D *Vertex, int PolygonNum, int GrHandle, int TransFlag) ;

VERTEX3Dという構造体が出てきましたが、
今回は、頂点座標(VECTOR pos)と、
ディフューズカラーのみを使用していきます。

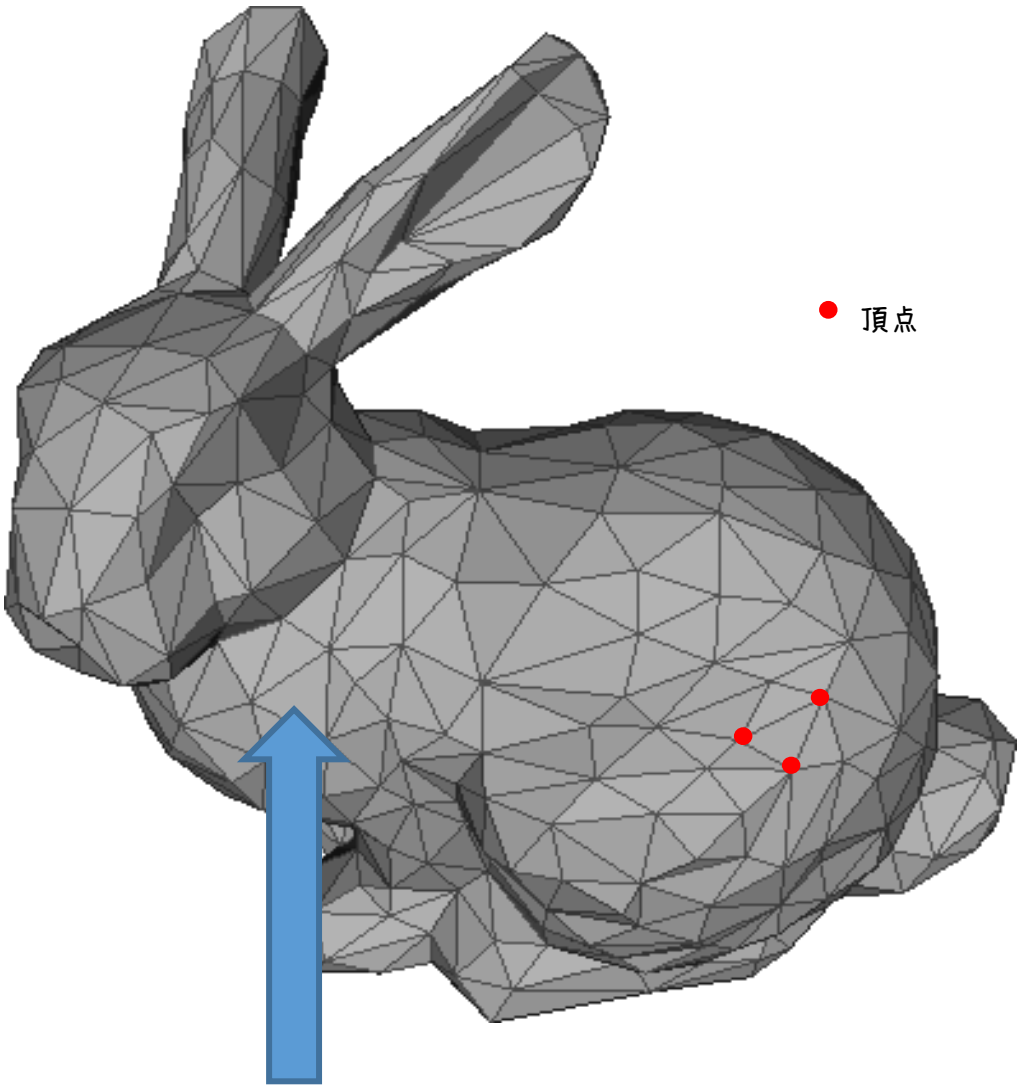
```
struct VERTEX3D
{
    // 座標
    VECTOR pos ;
    // 法線
    VECTOR norm ;
    // ディフューズカラー
    COLOR_U8 dif ;
    // スペキュラカラー
    COLOR_U8 spc ;
    // テクスチャ座標
    float u, v ;
    // サブテクスチャ座標
    float su, sv ;
} ;
```

第１引数
VERTEX3D *Vertex　　:　　三角形ポリゴンを形成する頂点の情報群。
　　座標と基礎色を今回は使用。
　　ディフューズ = 拡散反射光。

第２引数
int PolygonNum　　:　　描画する三角形ポリゴンの数。

頂点？三角ポリゴン？

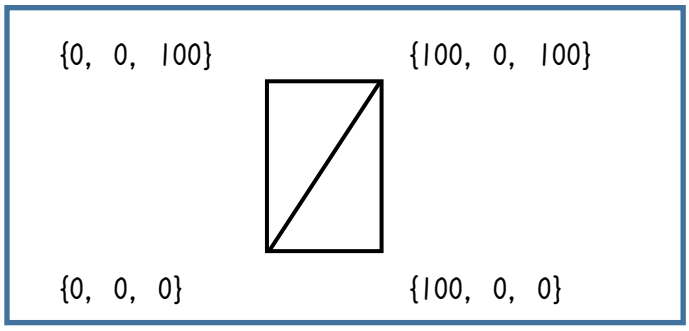
３Ｄモデルは、三角ポリゴンを繋ぎ合わせることで、
形成されています。

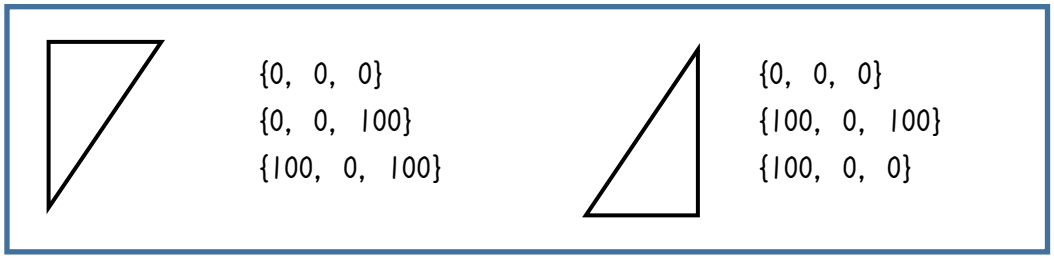


たくさんの三角ポリゴンの集合体が、３Ｄモデルとなっている。
もっと滑らかなモデルにするには、もっと三角ポリゴンを細かくして繋げる必要がある。
グラフィックは良くなるが、ポリゴンの数が多いほど、
３Ｄの計算量が多くなるため、処理負荷が高くなる。

メッシュ　　・・・　　ポリゴンの集まりのこと。

ひとまず、XZ面に四角を描画してみましょう



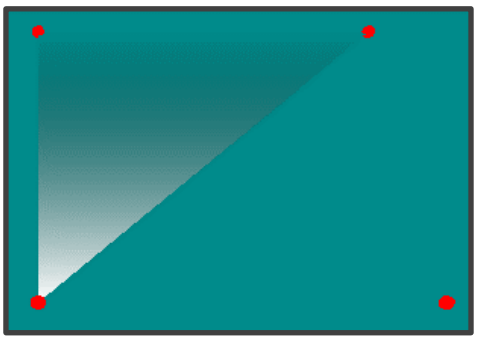



```
mVerticesSquare[0].pos = { 0, 0, 0 };
mVerticesSquare[1].pos = { 0, 0, 100 };
mVerticesSquare[2].pos = { 100, 0, 100 };
```

頂点の数は、ポリゴン(面)の数×３。
頂点を繋ぎ合わせて、三角ポリゴンを作る。

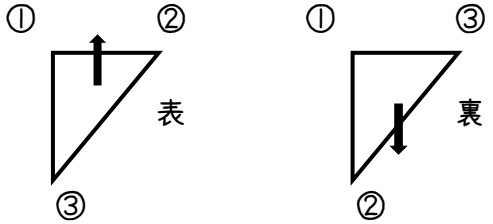
ディフューズカラー(拡散反射光)は、
最初の頂点だけに設定してみて、
どんな見え方になるか試してみましょう。

```
mVerticesSquare[0].dif = GetColorU8(255, 255, 255, 255);
                                    R、G、B、α
```

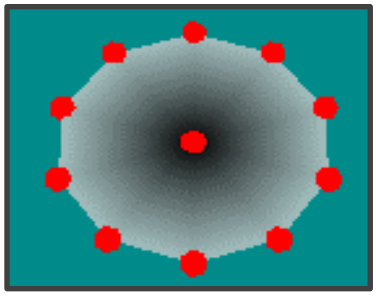


どっちが表？裏？

左手座標系のDxLibは、時計回りが表。

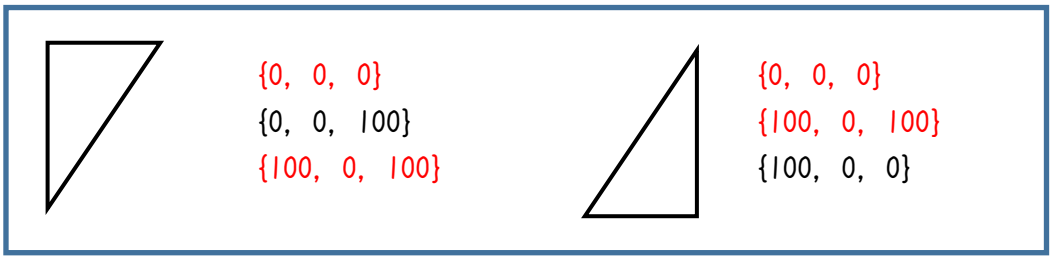


次は、XZ面に円形を描画してみましょう



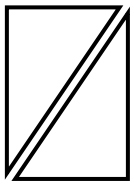
今度は、こちらの関数を使って貰います。
[Dxlib関数]
DrawPolygonIndexed3D
(VERTEX3D *Vertex, int VertexNum, unsigned short *Indices,
int PolygonNum, int GrHandle, int TransFlag)

先ほどのポリゴンデータの形成方法だと、
無駄な情報量が多くできてしまいます。



頂点は、必ず、別の頂点と繋がっていますので、
上図の色のついた座標のように、頂点座標が重複されます。

これを緩和する仕組みとして、
頂点インデックスというものがあります。



VERTEX3D[0] {0, 0, 0}
VERTEX3D[1] {0, 0, 100}
VERTEX3D[2] {100, 0, 0}
VERTEX3D[3] {100, 0, 100}

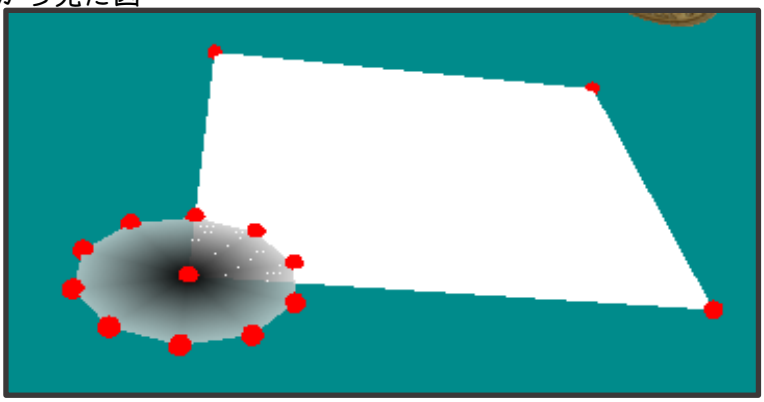
第３引数 short *Indices

[0]->[1]->[3]	Indices[0] = 0	Indices[0] = 0
[0]->[3]->[2]	Indices[1] = 1	Indices[1] = 3
	Indices[2] = 3	Indices[2] = 2

カメラの移動処理を加えて、自作ポリゴンを観察

復習も兼ねて、まずはカメラの移動処理を実装しましょう。
そして、自作したポリゴンを観察しながら３Ｄ用語も覚えていこう。

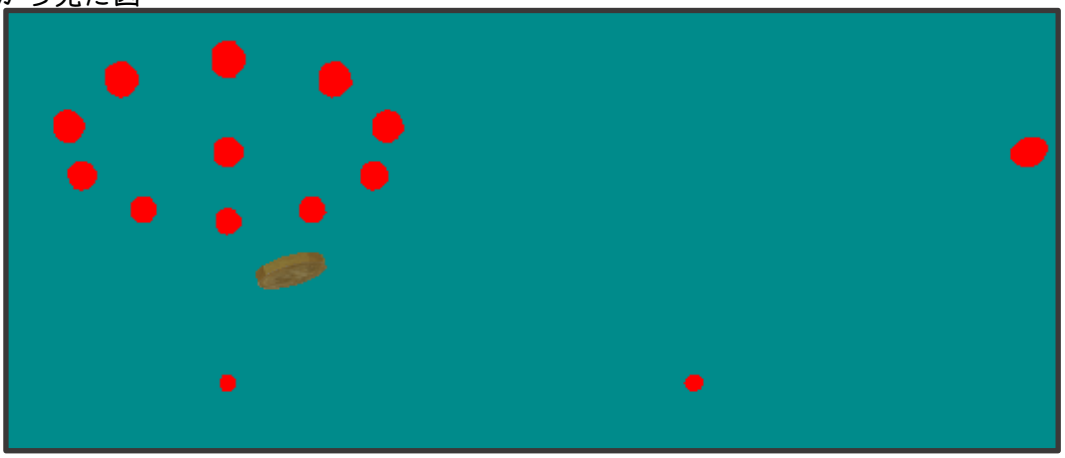
上から見た図



横から見た図

ペラペラ。

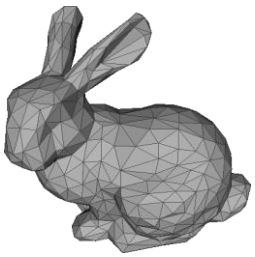
下から見た図



ディフューズカラーがのってない。

カリング

ポリゴンの表裏のどちらかを描画しないようにすること。
処理負荷を軽減できる。



通常、モデルのポリゴンの裏側は、
（皮膚の裏？）
描画する必要がない。

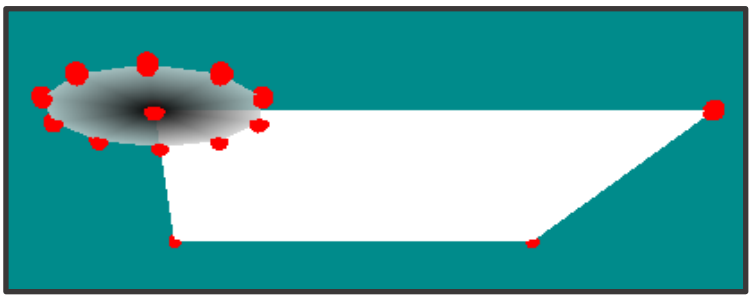
本プロジェクトでは、３Ｄ関連の初期設定で、
バックカリング（裏側を描画しない）を有効にしている。

```
SetUseBackCulling(true);
```

そのため、ポリゴンの裏側からカメラで見ると、
何も表示されなくなる。

```
SetUseBackCulling(false);
```

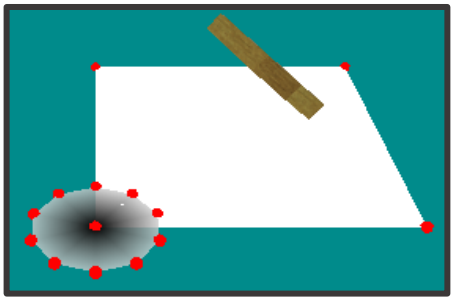
カリングをオフしてみて、見え方が変わるか確認してみましょう。



描画されました！
でも不要ですので、バックカリングを有効に戻しておきましょう。

Zバッファ

描画対象のオブジェクトの前後関係を、
カメラからの距離を元に計算し、奥側にあるオブジェクトよりも、
手前のオブジェクトの方を優先して、描画するようにする仕組み。
深い内容なので、もっと理解したい方は、個別で調べてみましょう。

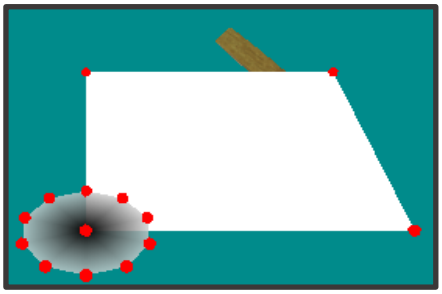


前後関係的には、
コインの方が手前にあるので、
描画優先されている。

但し、

```
// Zバッファを有効にする
SetUseZBuffer3D(false);
// Zバッファへの書き込みを有効にする
SetWriteZBuffer3D(false);
```

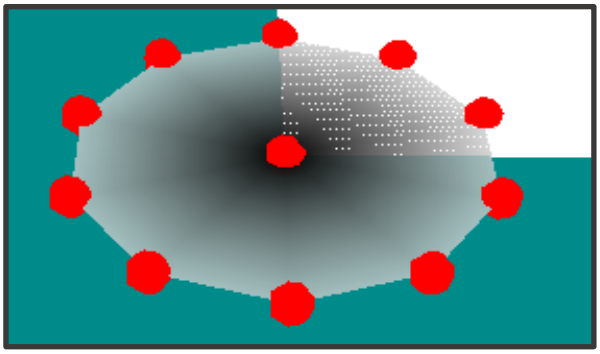
Zバッファを無効にすると、



描画の実行順で、描画順が
決まってしまうので、
コインよりも、
自作ポリゴンの方が、
描画優先されてしまっている。

```
void DemoScene::Draw(void)
{
    mCoin->Draw();
    mCoinMove->Draw();
    mCoinComplete->Draw();
    mSphereGenerator->Draw();
}
```

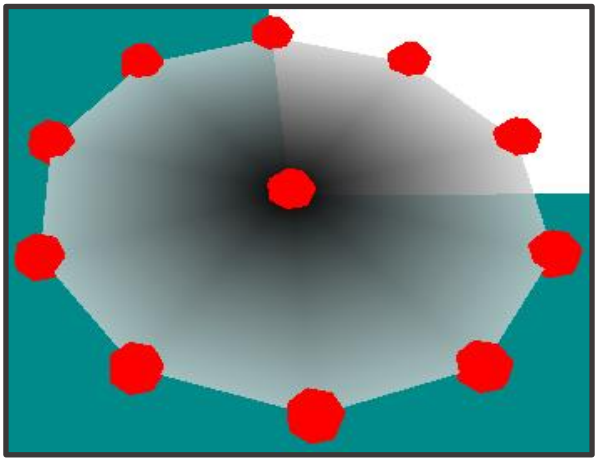
Zファイティング



自作ポリゴンが重なっている部分がチラチラしています。
これは、前述のZバッファの奥行き数値(深度 = デプス)が
同じになっているため、どっちを優先して描画して良いかわからず、
喧嘩して、チラチラとした表示になってしまいます。
これをZファイティングといいますが、解決方法としては、
デプス以外にも、描画順を決める要素を作り、管理するか、
ゲーム上、問題ないようでしたら、位置を少しズラしてあげることに
より、簡単に解決できます。

```
VECTOR cPos = mPos;
cPos.y += 0.5f;

//mVerticesCircle[mCntVertex++].pos = mPos;
mVerticesCircle[mCntVertex++].pos = cPos;
```



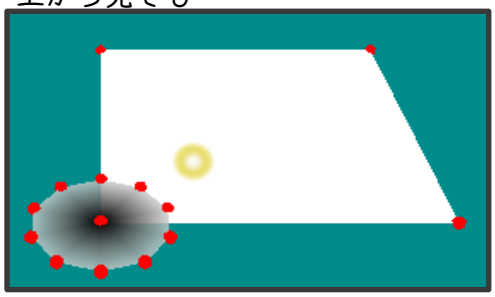
これで解決！

いよいよ、ビルボードで描画

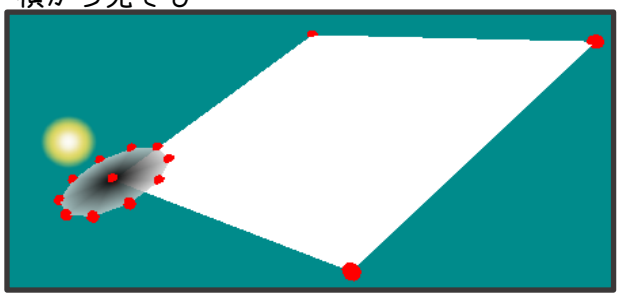
画像ファイル“Light.png”を読み込んで、描画してみましょう。


```
DrawBillboard3D(mPos, 0.5f, 0.5f, mSize, mAngle, mImgLight, true);
```

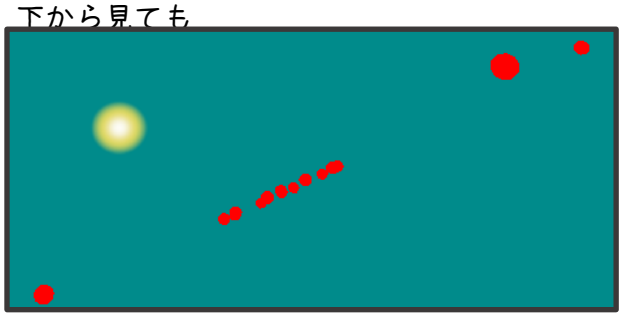
上から見ても



横から見ても



下から見ても

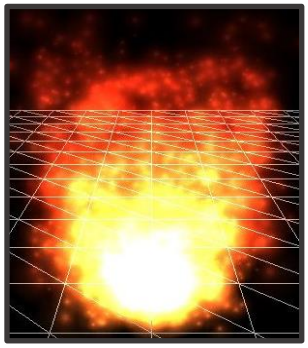


ペラペラにならず、ずっと正面を向いてくれます。

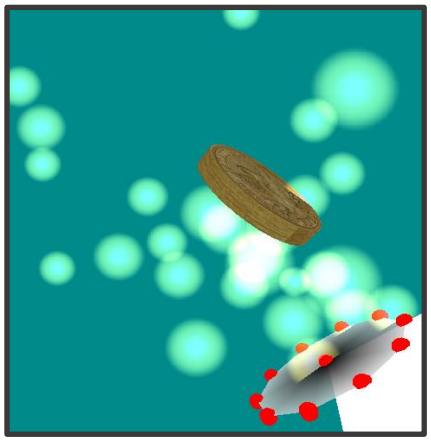
超簡易的なパーティクルエフェクト

たくさんのパーティクル(粒子)を制御して、
エフェクトを表現していくシステムをパーティクルシステムと呼びます。



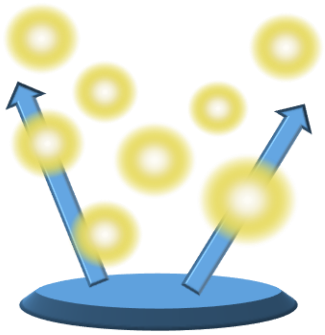


ゲームエンジンやエフェクトソフトには、
非常に強力なパーティクルシステムが実装されており、
様々なエフェクト表現ができます。
Dxlibだと、Effekseerを使用するのが無難だと思いますが、
今回は、３Ｄの学習も兼ねて、
超簡易的なものを自作していきたいと思います。



基本的な設計

円状の範囲から、ランダムにパーティクルを『 30個 』発生させ、
発生位置から円の外側方向かつ、
円の垂直方向にパーティクルを移動させる。
パーティクルはランダムな経過時間後、消滅させるが、
同時に、新しいパーティクルを生成させて、途切れないようにする。



基本的な設計が実装できていれば、詳細は自由にして頂いて
大丈夫です。が、後の解説内容と比べ、不味いと思った場合は、
修正をお願いします。

詳細な設計の例

Particleクラス

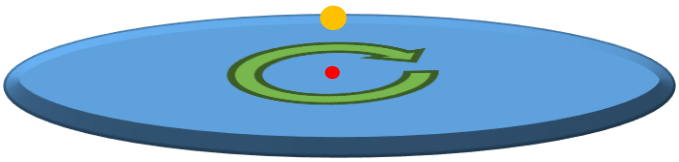
描画、移動、生存時間の減少。

ParticleGeneratorクラス

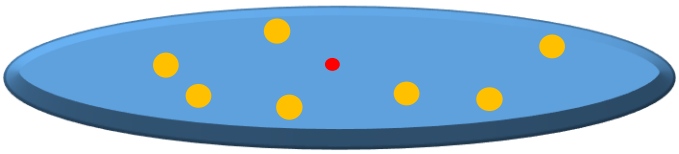
パーティクルをランダムに生成する。
生存時間が過ぎたパーティクルを再配置する。

大きさ(ランダム)
移動速度(ランダム)
生存時間(ランダム)

発生位置(ランダム)



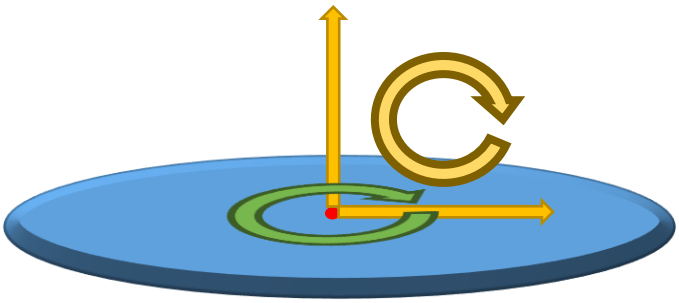
中心(赤)からオレンジまでの距離をランダムとする。(0～半径)
上記で求めた座標をY軸にランダム回転(0～360)すると、
円範囲内のランダムな位置が求められる。



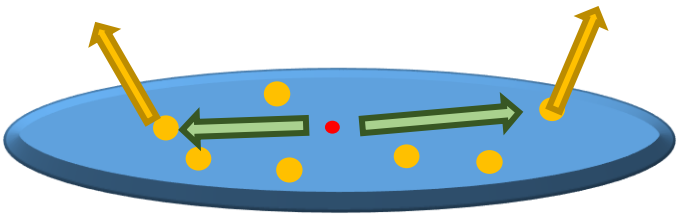
後々、発生範囲をドーナッツ状にしたくなった場合は、
ここの計算方法を改良する。

移動方向(ランダム)

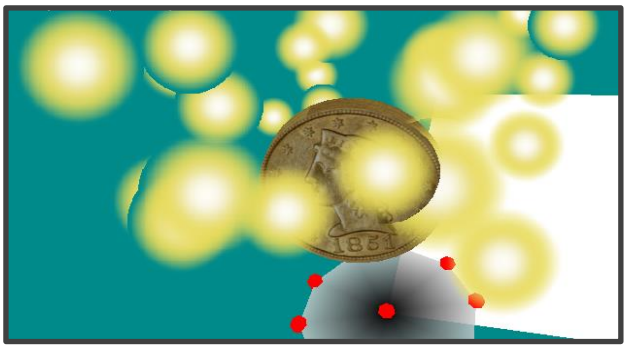
発生位置を計算したY軸の回転に加え、X軸の回転を使うと良い。

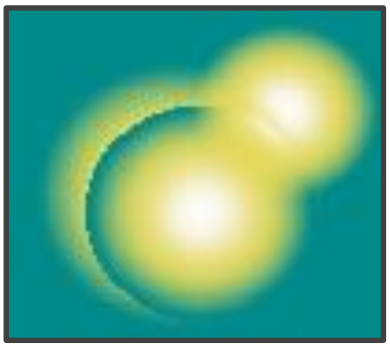


↑を向かせるためには、X軸のマイナスであることに注意。
60～80度ぐらいのランダム値が良いでしょう。



できた！！・・・いや、よく見ると。。。



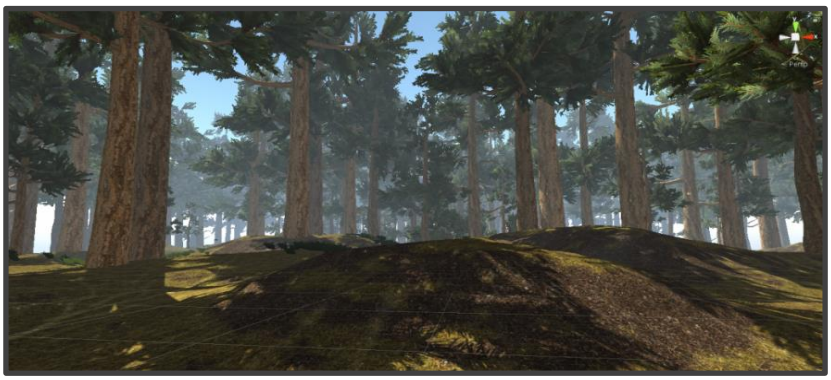


透過は上手くできていますか？
透過、半透明処理は実はやっかいでコストが高いです。
不透明→透明→半透明の順にやっかいです。

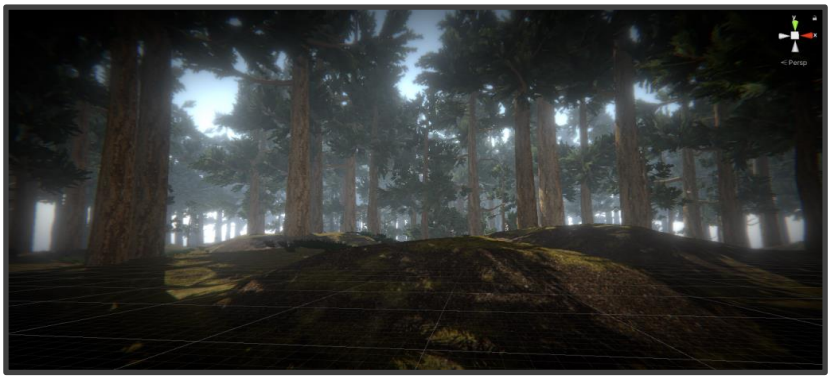
一般的なレンダリング順はこんな感じです。

①	Background	背景
②	Geometry	不透明なオブジェクト
③	Transparent	透過が必要なオブジェクト
④	Overlay	ポストエフェクト

↓　ちなみに、比較

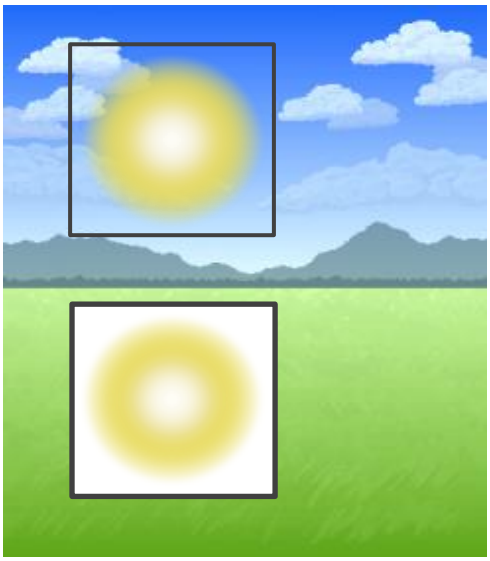


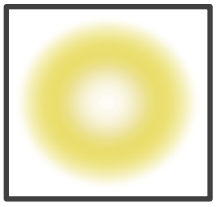
ポストエフェクト
無し



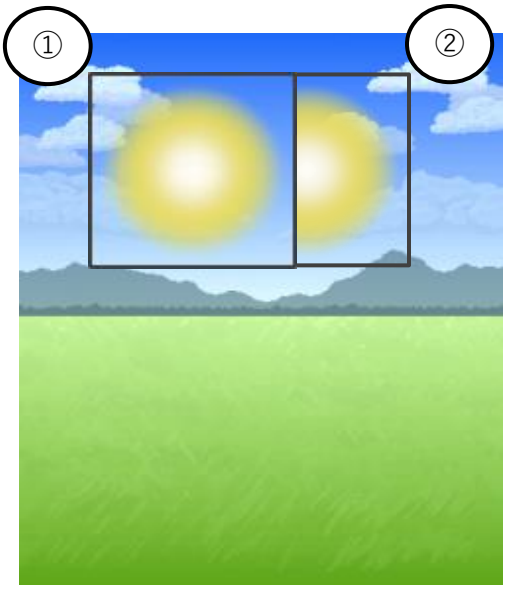
ポストエフェクト
有り

Zバッファの弱点として、描画順によって、
半透明が不完全な状態になります。





これは不透明。
草原が見えない。



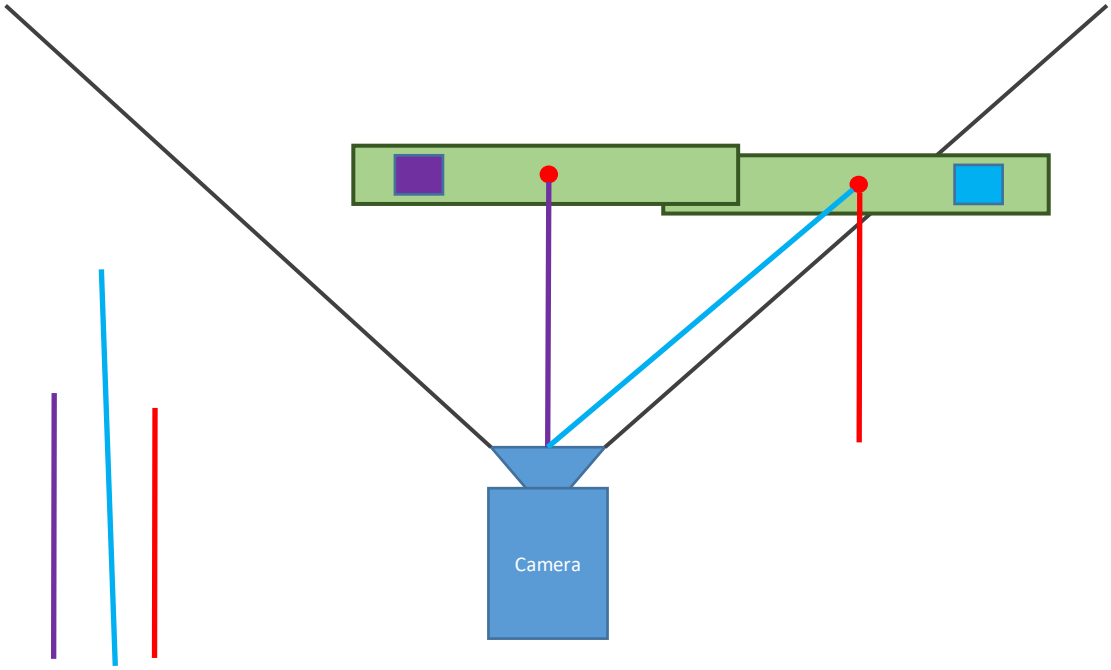
透過・半透明により、
光の玉の長方形枠内に、
背景の空がキチンと見える。
但し！！
Zバッファも書き込まれる。

よって、
① 手前
② 奥側
①、②の順で描画すると。。。
←のような表現になる。

これを改善するためには。。。
カメラから見て、奥側から順に描画していく必要がある。

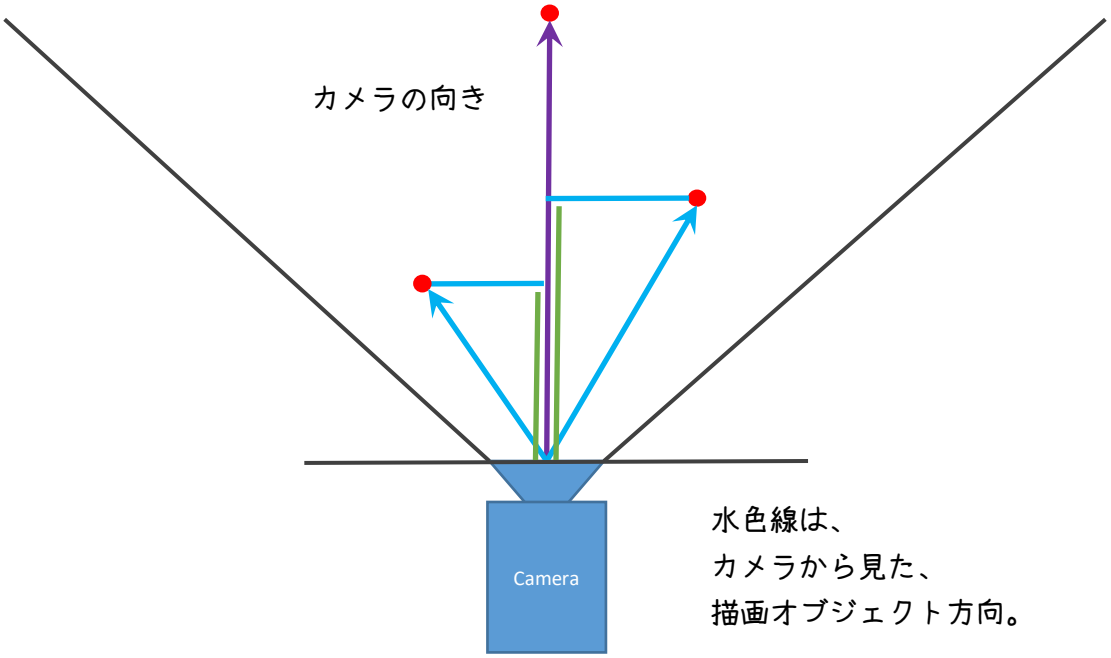
これを『 Zソート 』といいます。

このZソート、気を付けて貰いたいのが、
カメラからの純粋な距離順ではなくて、
奥行き順であるということ。

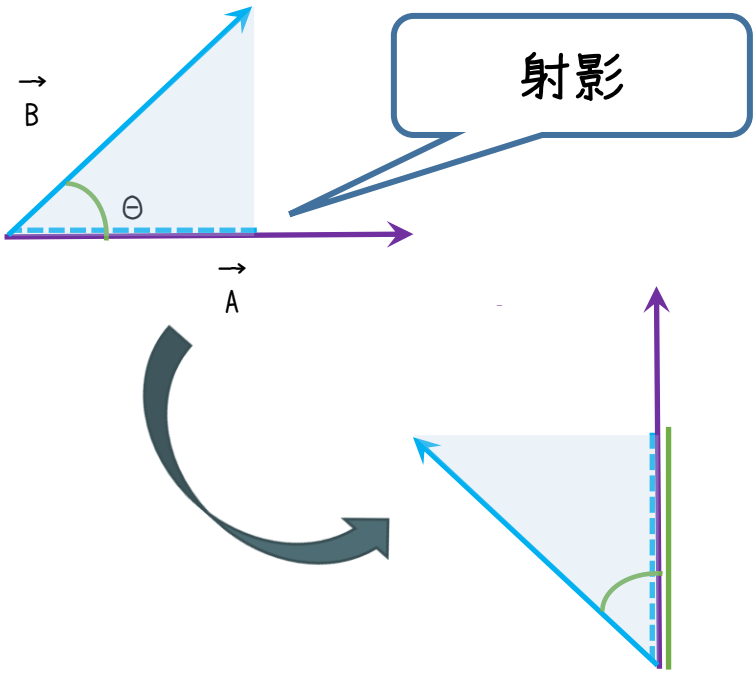


カメラからの純粋距離が短いのは紫の線で繋がっている長方形ですが、
『 カメラからの奥行き 』は、水色の線の方が近いですので、
描画の優先度としては、紫より水色の方が高いです。
奥行きから順に描画していく必要がありますので、
紫→水色の順で描画していきます。

Zソートのやり方

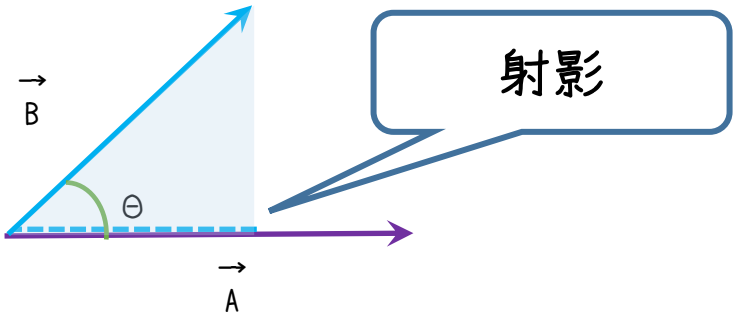


カメラの位置から、描画オブジェクトの奥行き距離が知りたい。。。
この図、どこかで見たような。。



そうだ！　1年生の時に習った『 内積 』を使ってみよう！

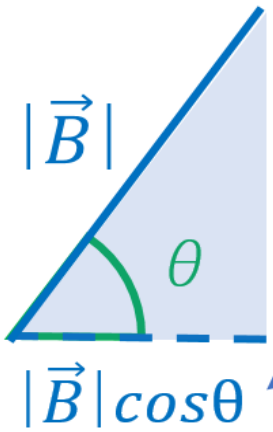
内積(dot)　　$\vec{A}\cdot\vec{B}$



3次元ベクトルの内積　$\vec{A}=(x_a, y_a, z_a)$、$\vec{B}=(x_b, y_b, z_b)$

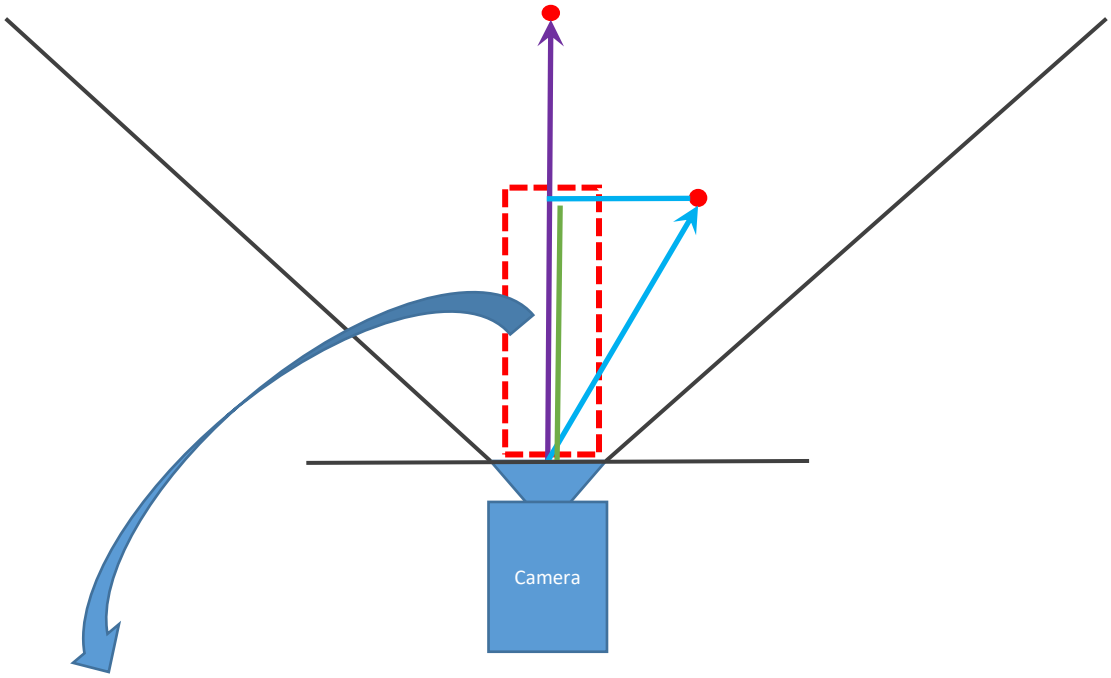
$$\vec{A}\cdot\vec{B} = x_a x_b + y_a y_b + z_a z_b$$

$$\vec{A}\cdot\vec{B} = |\vec{A}|\,|\vec{B}|\cos\theta$$



$$|\vec{B}|\cos\theta = \frac{\vec{A}\cdot\vec{B}}{|\vec{A}|}$$

by　真島先生作　数学基礎Ⅰ



$$|\vec{B}|\cos\theta = \frac{\vec{A}\cdot\vec{B}}{|\vec{A}|}$$

距離ではなく、比較を行うだけであれば、
分母のベクトルAは省略して良いが、今回は式に当てはめ、正規化して使用する。

カメラ方向　・　カメラから見たエフェクト方向

正規化されたカメラ方向

カメラ方向	・・・	カメラ座標と注視点から求めても良いし、 カメラ姿勢(角度やクォータニオン)から 求めても良い。
エフェクト方向	・・・	エフェクト位置 － カメラ位置

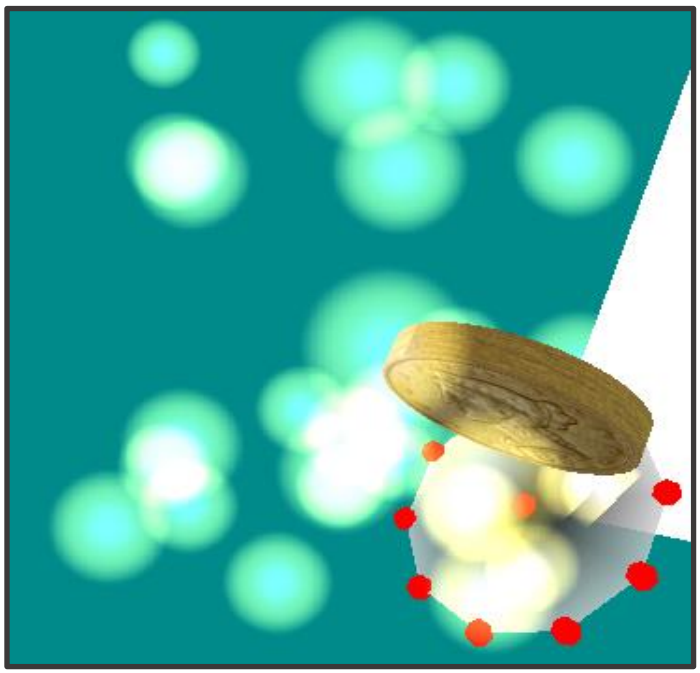
Dxlibの内積関数VDot。

```
/ ベクトルの内積
__inline float      VDot( const VECTOR &In1, const VECTOR &In2 )
{
    return In1.x * In2.x + In1.y * In2.y + In1.z * In2.z ;
}
```

奥行き(デプス)距離を使用してソートする。
今回は、わかりやすいように事前にメンバ変数にセットしておく。
複数パーティクルは、コレクション(vectorなど)で管理しておくと、

```
// Zソート
sort(mParticles.begin(), mParticles.end(),
    [](Particle* x, Particle* y) { return x->GetZLen() > y->GetZLen(); });
```

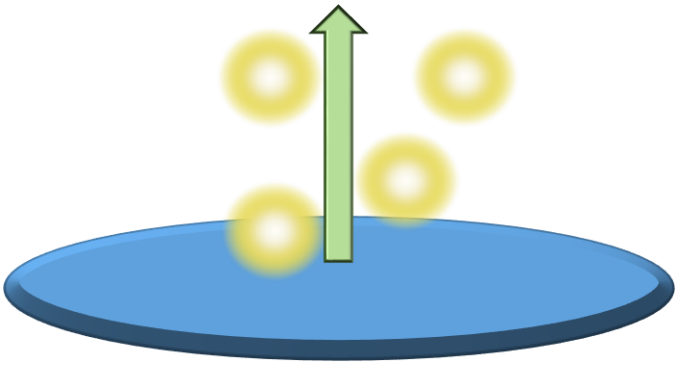
上記のような形でソートできる。
奥行きを他の用途で全く使用しないのであれば、
メンバ変数に保持する必要もないし、比較ソートもcompareで行うと、
無駄なメモリが削減できるが、今回は、わかりやすさのため、メンバ変数を設ける。



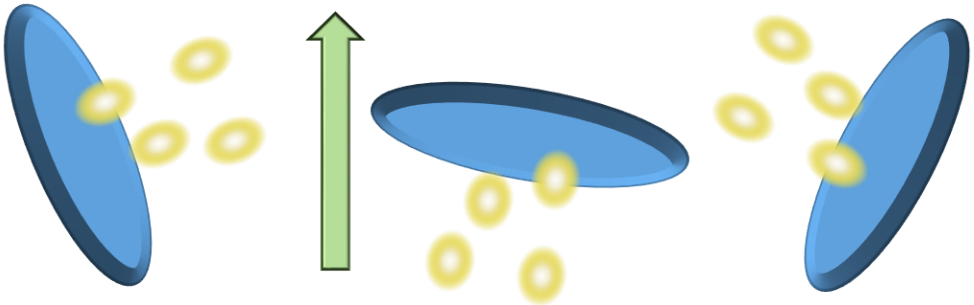
左は加算ブレンド使用

やっとできました。お疲れ様でした。。。。？

いや、更に進化させます。
ParticleGenerator自身を回転させて、
自由自在にエフェクトを発生させましょう。



今は、ParticleGenerator自身は、Yの正方向に固定してしまっている。
ParticleGenerator自身に回転情報を持たせて、

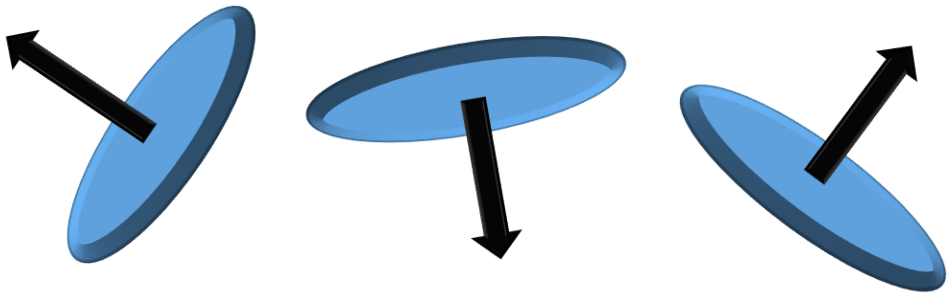


自由自在にエフェクトを制御できるようにしていきます。

回転といったら、Quaternion。
回転状態がわかりやすいように、回転状態に応じた向きを可視化させましょう。
今回は、回転無し(0, 0, 0)をZの正方向(0, 0, 1)とします。

```
VECTOR v1 = VNorm(VTransform({ 0.0f, 0.0f, 1.0f }, mAnglesQ.ToMatrix()));
DrawLine3D(mPos, VAdd(mPos, VScale(v1, 100.0f)), 0x000000);
```

あとは、IJKLなどの余ったキーを使って、
キーの押下している間、X軸、Y軸に回転するようにしてみましょう。
デバッグで表示させている線がグルグル回るはずです。

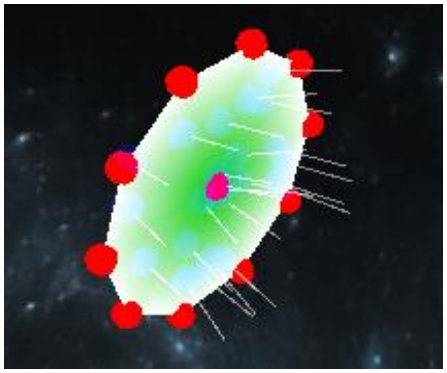


ここまでは問題ないかと思います。

この回転の影響を受けるのは、
パーティクルの発生位置と移動方向です。

パーティクルを発生させるGenerate処理を修正して、
正しい位置と移動方向になるようにしましょう。

ヒントは、
Generatorが親とすると、パーティクルは子供です。
子供が親の回転を引き継ぐように、回転を合成してみましょう。

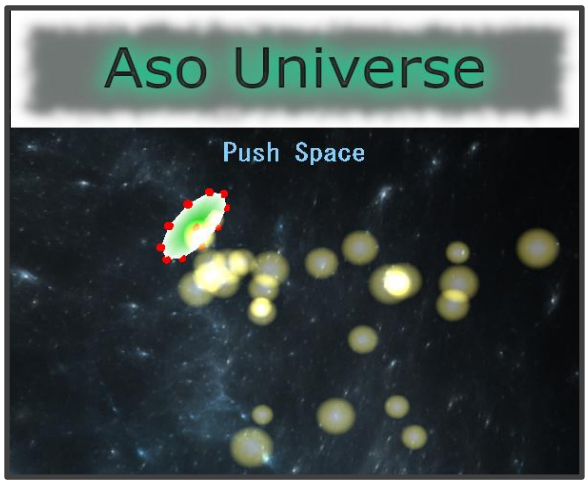


確認用に、一旦ビルボードの表示と、パーティクルの移動処理をコメントして、
発生場所に球体を表示して、移動方向に線を引いて見ると良いでしょう。

プラスアルファで、
パーティクル生成時に、いっぺんにパーティクルが生成されると、
途中でパーティクルが途切れ、見栄えが悪くなりますので、
徐々に生成する処理を追加しても良いかもしれません。

Unityのように、透明の状態から徐々に表示され、
生存時間に応じて、徐々にフェードしていっても良いかもしれません。

ついに完成！



お疲れ様でした。
３Ｄシューティングの方に持っていって、
自機の疾走感を演出したり、エンジンの噴出演出に使用してみたりしましょう。

ポイント

- パーティクルの大きさは小さめに調整
- 移動速度も遅めに
- 生存時間も短めに
- 生成間隔も短めに
- 移動方向は円周から広がるように
- 発生位置のドーナツ化