DirectX 資料②

スプライト編

ゲームエンジニア科２年

1. 読み込みのタイミング

今までは、画像ファイル等の読み込みは各ユニットクラスのコンストラクタで行っていた。

（例えば、タイトル画面の画像はタイトルクラスのコンストラクタで）

しかし Direct3D を使う場合は、**Direct3D デバイスの準備ができてからでないと画像の読み込みはできない**。

①　UnitBase に関数追加

そこで、各ユニットクラスにコンストラクタとは別に読み込みようの関数を追加する必要がある。

全ユニットに必要なので、UnitBase に追加しよう。

<UnitBase.h>

```
class UnitBase
{
protected:
    //ユニットの位置
    POINT   position;

public:
    //読み込み処理
    virtual void Load();

    //更新処理
    virtual void Update();
```

<UnitBase.cpp>

```
//―――――――――――――――――――――――――――――――
//  読み込み処理
//―――――――――――――――――――――――――――――――
void UnitBase::Load()
{

}
```

②　SceneBase に関数追加

先ほど作った Load 関数は各シーンから呼ばれる。

ということで、SceneBase クラスに**各ユニットの Load を呼ぶための関数**が必要になる。

<SceneBase.h>

```
class SceneBase
{
        :
        :
public:
    //デストラクタ
    virtual ~SceneBase();

    //読み込み処理
    void Load();
```

<SceneBase.cpp>

```
//―――――――――――――――――――――――――――――――
//  読み込み処理
//―――――――――――――――――――――――――――――――
void SceneBase::Load()
{
    //全ユニットを処理する
    for (WORD i = 0; i < unit.size(); i++)
    {
        unit[i]->Load();
    }
}
```

③　Game に関数追加

先ほど作った各シーンの Load 関数は、ゲームクラスから呼ばれる。

ということで、Game クラスに**各シーンの Load を呼ぶための関数**が必要になる。

<Game.h>

```
自分でやって！！
```

<Game.cpp>

```
//―――――――――――――――――――――――――――――――
//  読み込み処理
//―――――――――――――――――――――――――――――――
void Game::Load()
{
    scene[g_gameScene]->Load();
}
```

④　Load 関数を呼ぶ

ゲームクラスの Load 関数は、Direct3D の準備が完了したタイミングで呼べばいい。

<Main.cpp>

```
    case WM_CREATE:
        //ランダムの準備
        srand((unsigned)time(NULL));

        //ゲームオブジェクトを作成
        game = new Game;

        //Direct3D の初期化
        game->InitD3d(hWnd);

        //読み込み処理
        game->Load();

        break;
```

2. スプライトクラス

ゲームで使用する 2D 画像のことを**スプライト**という。

画像を表示するための新しいクラスを作成しよう。

ツール開発で作った Picture クラスの Direct3D バージョンと考えればよい。

とりあえず必要最低限の、読み込み処理と描画処理だけ用意しておこう。

<Sprite.h> **（コメントは各自で！）**

```
#pragma once

#include "Global.h"

class Sprite
{
public:
    Sprite();
    ~Sprite();

    void Load(char* fileName);
    void Draw();
};
```

読み込み関数は、ファイル名を指定できるようにする。

描画関数の引数は、とりあえず無しにしておこう。

<Sprite.cpp> **（コメントは各自で！）**

```
#include "Sprite.h"


Sprite::Sprite()
{
}


Sprite::~Sprite()
{
}

void Sprite::Load(char* fileName)
{

}

void Sprite::Draw()
{

}
```

3. 読み込み処理

① スプライトオブジェクトを作成

2D 画像は厳密に言うと**スプライトという板に画像を貼り付けてゲーム画面上に置く**というイメージになる。

3D の場合は“ポリゴン”という板に“テクスチャ”という画像を貼り付けるが、あれと似たようなもの。

そこで、まずはスプライトという板を用意する。

<Sprite.h>

```
class Sprite
{
    //スプライトオブジェクト
    LPD3DXSPRITE pSprite;
public:
```

これはスプライトを入れる変数になる。

次に Load 関数でスプライトを作成する。

<Sprite.cpp>

```
void Sprite::Load(char* fileName)
{
    // 「スプライトオブジェクト」の作成
    if (FAILED(D3DXCreateSprite(g_pDevice, &pSprite)))
    {
        MessageBox(0, "スプライトの作成に失敗しました", "エラー", MB_OK);
    }
}
```

D3DXCreateSprite というのがスプライトを作成する関数で、作られたスプライトが先ほど作った pSprite に入る。

失敗した場合はメッセージを表示するようにしてみた。

② テクスチャを作成

スプライトという板に貼り付けるための画像ファイルを読み込む。

3D と同じで、画像のことは**テクスチャ**と呼ぶ。

まずは、テクスチャを入れるための変数を用意。

<Sprote.h>

```
class Sprite
{
    //スプライトオブジェクト
    LPD3DXSPRITE pSprite;

    //テクスチャオブジェクト
    LPDIRECT3DTEXTURE9 pTexture;
```

そして、Load 関数で読み込む

<Sprite.cpp>

```
void Sprite::Load(char* fileName)
{
        :
        :
    //「テクスチャオブジェクト」の作成
    if (FAILED(D3DXCreateTextureFromFileEx(g_pDevice, fileName, 0, 0, 0, 0, D3DFMT_UNKNOWN,
        D3DPOOL_DEFAULT, D3DX_FILTER_NONE, D3DX_DEFAULT, 0xff000000, NULL, NULL, &pTexture)))
    {
        MessageBox(0, "テクスチャの作成に失敗しました", fileName, MB_OK);
    }
}
```

D3DXCreateTextureFromFileEx というのが、画像ファイルを読み込んでテクスチャを作成する関数。

引数はたくさんあるが、**第１引数に”Direct3D デバイス“、第２引数に”ファイル名“を指定するとテクスチャが作られ、最後の引数に指定した変数に、完成したテクスチャが入る**ということだけ分かっていればよい。（今はね）

例のごとく、失敗した場合はメッセージを表示するようにした。

4. 解放処理

スプライトもテクスチャも LP で始まる型なので、解放処理を忘れずに。

<Sprite.cpp>

```
Sprite::~Sprite()
{
    pTexture->Release();
    pSprite->Release();
}
```

5. 描画処理

Draw 関数を作っていく。

① 開始と終了

まずは「スプライトの描画開始」と「スプライトの描画終了」の関数を呼ぶ必要がある。

<Sprite.cpp>

```
void Sprite::Draw()
{
    pSprite->Begin(D3DXSPRITE_ALPHABLEND);


    pSprite->End();
}
```

②　必要な情報を変数で用意

まず、画像を表示する位置を **D3DXVECTOR3** 型で用意する。

とりあえず、適当な位置にしておこう。（今作っているのは二次元の処理なので Z は使わない（0 にしておく））

<Sprite.cpp>

```
void Sprite::Draw()
{
    pSprite->Begin(D3DXSPRITE_ALPHABLEND);

    //描画位置
    D3DXVECTOR3 postion = D3DXVECTOR3(100, 100, 0);


    pSprite->End();
}
```

つぎに、画像の切り抜き範囲を **RECT** 構造体で用意する。

左上と左下の座標を指定する。（幅と高さではないので注意）。

とりあえずこれも適当な値にしておこう。

<つづき>

```
//切り抜き範囲
RECT cut = { 50, 100, 200, 150 };
```

最後に基準点を **D3DXVECTOR3** 型で用意する。

基本的には（0,0,0）で構わない。（画像の中心で回転させたい場合は中心を指定するが、表示位置がずれる）

<つづき>

```
//基準点
D3DXVECTOR3 center = D3DXVECTOR3(0, 0, 0);
```

③　描画

pSprite->Draw 関数で描画する。

```
    //描画
    pSprite->Draw(pTexture, &cut, &center, &postion, D3DCOLOR_ARGB(255, 255, 255, 255));
```

引数は、貼り付けるテクスチャ、切り抜き範囲、基準点、表示位置、色。

6. とりあえず使ってみる

ためしに、タイトル画面のつもりで適当な画像を描画してみよう。

① タイトルクラスを作る

ここは今までどおりなので説明は要らないだろう。

<Title.h>

```
#pragma once
#include "UnitBase.h"

class Title : public UnitBase
{
public:
    Title();
    ~Title();
    void Load();
    void Render();
};
```

<Title.cpp>

```
#include "Title.h"

Title::Title()
{
}

Title::~Title()
{
}

void Title::Load()
{
}

void Title::Render()
{
}
```

<TitleScene.cpp>

```
//---------------インクルード----------------------
#include "TitleScene.h"
#include "Title.h"

//―――――――――――――――――――――――――――――――
//  コンストラクタ
//―――――――――――――――――――――――――――――――
TitleScene::TitleScene()
{
    //ユニットを追加
    unit.push_back(new Title);
}
```

②　Sprite クラスを使う

タイトルで画像を表示するので、Sprite クラス型の変数を作る。

<Title.h>

```
#pragma once
#include "UnitBase.h"
#include "Sprite.h"

class Title : public UnitBase
{
    Sprite sprite;

public:
    Title();
```

次に Load 関数で画像ファイルを読み込む。

画像ファイルはサーバーからとってくること。

<Title.cpp>

```
void Title::Load()
{
    sprite.Load("number.png");
}
```

そして、Render 関数で描画する。

<Title.cpp>

```
void Title::Render()
{
    sprite.Draw();
}
```

実行すると画像が表示される。

切り抜き範囲や、色などをいろいろといじってみよう。

7. Draw 関数の引数を考える

今のままでは、決まった位置に、決まった部分しか表示できない。

Picture クラスを作ったときと同じように、引数で指定できるようにしたい。

① 引数を洗い出す

どんな引数が必要だろうか。

Picture クラスと同じにした方が使いやすいだろうから

- 表示位置（X,Y,Z）
- サイズ（X,Y）
- 切り抜き位置（X,Y）

さらに追加で

- 基準点（X,Y,Z）
- 色（A,R,G,B）

と言ったところ・・・。

多すぎる！！

② 構造体にまとめる

引数が多すぎる場合は、構造体にまとめてしまうのも一つの手だ。

Sprite クラスの外側に構造体を用意しよう。

<Sprite.h>

```
#include "Global.h"

//画像の表示に使うデータ
struct SpriteData
{
    D3DXVECTOR3 pos;        //位置
    D3DXVECTOR2 size;       //サイズ
    D3DXVECTOR2 cut;        //切り抜き位置
    D3DXVECTOR2 center;     //基準点

    //色
    struct
    {
        int a, r, g, b;
    }color;
};

class Sprite
{
```

ここに作っておけば、Sprite.h をインクルードしたところでは SpriteData 型が使えるようになる。

③　Draw 関数を修正

引数で SpriteDraw 型の変数を受け取り、その情報を使うように修正しよう。

大きなデータを引数でやり取りする場合は、ポインタを使ったほうが良い。

<Sprite.cpp>（ヘッダーは自分で）

```
void Sprite::Draw(SpriteData* data)
{
    pSprite->Begin(D3DXSPRITE_ALPHABLEND);

    //描画位置
    D3DXVECTOR3 postion = [          ];

    //切り抜き範囲
    RECT cut = {
        [          ], [          ], [                    ], [                    ]
    };

    //基準点
    D3DXVECTOR3 center = [          ];
    center.z = [    ];

    //描画
    pSprite->Draw(pTexture, &cut, &center, &postion,
            D3DCOLOR_ARGB([          ], [          ], [          ], [          ]));

    pSprite->End();
}
```

（D3DXVECTOR2 型の変数は、そのまま D3DXVECTOR3 型変数に代入可能だが、Z の値はおかしくなる）

これで、一つの引数で全ての情報がやり取りできるようになった。試してみよう。

<Title.cpp>　　（左右はどちらも全く同じ意味。書きやすい方でどうぞ）

```
void Title::Render()
{
    SpriteData data;

    data.pos.x = 100;
    data.pos.y = 100;
    data.pos.z = 0;

    data.cut.x = 50;
    data.cut.y = 100;

    data.size.x = 200;
    data.size.y = 150;

    data.center.x = 0;
    data.center.y = 0;

    data.color.a = 255;
    data.color.r = 255;
    data.color.g = 255;
    data.color.b = 255;

    sprite.Draw(&data);
}
```

```
void Title::Render()
{
    SpriteData data;

    data.pos = D3DXVECTOR3(100, 100, 0);
    data.cut = D3DXVECTOR2(50, 100);
    data.size = D3DXVECTOR2(200, 150);
    data.center = D3DXVECTOR2(0, 0);
    data.color = { 255, 255, 255, 255 };

    sprite.Draw(&data);
}
```

④　初期値を設定

引数を 1 つにまとめたが、設定する項目が多いことに変わりはない。

そこで構造体にコンストラクタを追加し、初期値を設定しよう。

構造体にはクラスと同じようにコンストラクタを追加することができる。

そうすれば、値を設定しなければ初期値が使われるようになる。

<Sprite.h>

```
//画像の表示に使うデータ
struct SpriteData
{
        :
        :

    //コンストラクタ
    SpriteData()
    {
        pos =  D3DXVECTOR3(0, 0, 0);
        size =  D3DXVECTOR2(200, 200);
        cut =  D3DXVECTOR2(0, 0);
        center = D3DXVECTOR2(0, 0);
        color = { 255, 255, 255, 255 };
    }
};
```

サイズの値はとりあえず適当な値にしてみた。

こうしておけば、Draw 関数を呼ぶ時に何も値を設定しなければ左上に 200×200 のサイズで表示される。

<Title.cpp>

```
void Title::Render()
{
    SpriteData data;
    sprite.Draw(&data);
}
```

もし、表示位置だけ指定したいなら、それだけ書けばいい。

<Title.cpp>

```
void Title::Render()
{
    SpriteData data;
    data.pos.x = 100;
    sprite.Draw(&data);
}
```

⑤　サイズの取得

さきほどは、サイズを省略した場合はとりあえず 200×200 になるようにしたが、やはり省略した場合は画像全体が表示されるようにした方が便利だろう。

そこで、テクスチャのサイズを取得してみよう。

まずは、サイズを入れる変数を用意。

<Sprite.h>

```
class Sprite
{
    //スプライトオブジェクト
    LPD3DXSPRITE pSprite;

    //テクスチャオブジェクト
    LPDIRECT3DTEXTURE9 pTexture;

    //テクスチャのサイズ
    D3DXVECTOR2     texSize;
```

テクスチャを作成後、サイズを調べよう。

<Spritet.cpp>

```
void Sprite::Load(char* fileName)
{
        :
        :
    //テクスチャのサイズを取得
    D3DSURFACE_DESC  d3dds;
    pTexture->GetLevelDesc(0, &d3dds);
    texSize.x = d3dds.Width;
    texSize.y = d3dds.Height;
}
```

D3DSURFACE_DESC はテクスチャの情報を扱うための構造体で、

テクスチャクラスのメンバ関数 **GetLevelDesc** がテクスチャの情報を取得するもの。

あとは、これを初期値として使えばよい。

が・・・・

こう書くことはできない。

```
//画像の表示に使うデータ
struct SpriteData
{
        :
        :

    //コンストラクタ
    SpriteData()
    {
        pos = D3DXVECTOR3(0, 0, 0);
        size = D3DXVECTOR2(texSize.x, texSize.y); //これはエラー
        cut = D3DXVECTOR2(0, 0);
        center = D3DXVECTOR2(0, 0);
        color = { 255, 255, 255, 255 };
    }
};
```

texSize は Sprite クラスのメンバなので、それとは関係ない SpriteDate 構造体の中で使うことはできない。

そういう時は、まず初期値をありえない値にする。

```
    //コンストラクタ
    SpriteData()
    {
        pos = D3DXVECTOR3(0, 0, 0);
        size = D3DXVECTOR2(-999, -999);
        cut = D3DXVECTOR2(0, 0);
```

サイズをマイナスをすることは無いだろう。

そして、描画する場合にサイズが-999 だったら画像サイズを使うようにすればよい。

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
    pSprite->Begin(D3DXSPRITE_ALPHABLEND);

    //サイズを省略した場合は画像サイズを使う
    if (data->size.x == -999)  data->size.x = texSize.x;
    if (data->size.y == -999)  data->size.y = texSize.y;

    //描画位置
    D3DXVECTOR3 postion = data->pos;
```

これで、画像サイズを省略した場合は画像全体が表示されるようになったはず。

8. 拡大縮小（スケーリング）

画像を拡大縮小してみよう。

Direct3D では、2D でも 3D でも表示物を変形（移動、回転、拡大縮小）させるには行列を使う。

① 拡大行列を作成

まず、描画直前に行列を扱うための変数を作成。変数の型は **D3DXMATRIX**。

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
        :
        :

    //拡大行列
    D3DXMATRIX matScale;

    //描画
    pSprite->Draw(pTexture, &cut, &center, &postion,
        D3DCOLOR_ARGB(data->color.a, data->color.r, data->color.g, data->color.b));
```

つぎに、今作った行列を拡大縮小させるための行列にする。

とりあえず、適当に横 2 倍、縦 0.5 倍にする行列にしてみよう。

使う関数は **D3DXMatrixScaling** だ。

<つづき>

```
    //拡大行列
    D3DXMATRIX matScale;
    D3DXMatrixScaling(&matScale, 2, 0.5, 1);
```

Z 方向は変わらないので 1 倍にしている。

これで、画像を拡大縮小させる行列が完成した。

② 行列を適用する

今作った行列を、スプライトに適用する。

使う関数はスプライトクラスのメンバ関数 **SetTransform**。

<つづき>

```
    //拡大行列
    D3DXMATRIX matScale;
    D3DXMatrixScaling(&matScale, 2, 0.5, 1);

    //スプライトに行列を適用
    pSprite->SetTransform(&matScale);
```

実行してみると、横 2 倍、縦半分に引き伸ばされているはず。

③　拡大率の指定

好きな倍率で拡大できるようにしよう。

SpriteData 構造体に、縦と横の拡大率を扱う変数を追加すればよい。

これを省略した場合は、当然縦も横も 1 倍でいいだろう。

<Sprite.h>

```
//画像の表示に使うデータ
struct SpriteData
{
    D3DXVECTOR3 pos;        //位置
    D3DXVECTOR2 size;       //サイズ
    D3DXVECTOR2 cut;        //切り抜き位置
    D3DXVECTOR2 center;     //基準点
    D3DXVECTOR2 scale;      //拡大率

    //色
    struct
    {
        int a, r, g, b;
    }color;

    //コンストラクタ
    SpriteData()
    {
        pos =   D3DXVECTOR3(0, 0, 0);
        size =  D3DXVECTOR2(-999, -999);
        cut =   D3DXVECTOR2(0, 0);
        center = D3DXVECTOR2(0, 0);
        color = { 255, 255, 255, 255 };
        scale = [                ];
    }
};
```

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
        :
        :

    //拡大行列
    D3DXMATRIX matScale;
    D3DXMatrixScaling(&matScale, [              ], [              ], 1);

    //スプライトに行列を適用
    pSprite->SetTransform(&matScale);

    //描画
    pSprite->Draw(pTexture, &cut, &center, &postion,
        D3DCOLOR_ARGB(data->color.a, data->color.r, data->color.g, data->color.b));

    pSprite->End();
}
```

自由に拡大縮小できることを確認しよう。

9. 回転

回転もできるようになれば、もうどんな表現もできるようになる。

① 回転行列を作る

拡大縮小とやり方は同じ。

ただし、回転行列を作る関数は **D3DXMatrixRotationZ**。

Sprite.cpp

```
void Sprite::Draw(SpriteData* data)
{
        :
        :
    //拡大行列
    D3DXMATRIX matScale;
    D3DXMatrixScaling(&matScale, data->scale.x,data->scale.y, 1);

    //回転行列
    D3DXMATRIX matRotate;
    D3DXMatrixRotationZ(&matRotate, 0.785);

    //スプライトに行列を適用
    pSprite->SetTransform(&matScale);
```

引数で回転する角度を指定するのだが、単位は**ラジアン**となる。(3.14 ラジアン＝180°)

②　ラジアン ⇔ 度

これでは分かりにくいので、D3DXToRadian マクロを使い、馴染のある「度」で指定できるようにしよう。

```
    //回転行列
    D3DXMATRIX matRotate;
    D3DXMatrixRotationZ(&matRotate, D3DXToRadian(45));
```

③　行列の合成

ひとつの物体に使える行列は一つだけである。

現状すでに拡大縮小行列を使ってしまっているが、回転行列をどのように使えばいいのだろうか。

実は行列は「**かけ算」をすることで合成することができる**。

拡大縮小行列と回転行列をかけ算したものを使えばよい。

```
//回転行列
D3DXMATRIX matRotate;
D3DXMatrixRotationZ(&matRotate, D3DXToRadian(45));

//合成行列
D3DXMATRIX matWorld = matScale * matRotate;

//スプライトに行列を適用
pSprite->SetTransform(&matWorld);

//描画
pSprite->Draw(pTexture, &cut, &center, &postion,
    D3DCOLOR_ARGB(data->color.a, data->color.r, data->color.g, data->color.b));

pSprite->End();
```

④　角度の指定

好きな角度で表示できるようにしよう。

SpriteData 構造体に、角度を扱う変数を追加すればよい。

これを省略した場合は、当然０°でいいだろう。

<Sprite.h>

```
//画像の表示に使うデータ
struct SpriteData
{
        :
        :
    float      angle;     //角度

    //色
    struct
    {
        int a, r, g, b;
    }color;

    //コンストラクタ
    SpriteData()
    {
            :
            :
        angle = ;
    }
};
```

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
        :
        :

    //回転行列
    D3DXMATRIX matRotate;
    D3DXMatrixRotationZ(&matRotate, D3DXToRadian(          ));
```

これで、Draw 関数を呼ぶ時に角度を指定できるようになった。

```
void Title::Render()
{
    SpriteData data;

    data.scale.x = 2;
    data.angle = 15;    //15° 回転させる
    sprite.Draw(&data);
}
```

10. 中心で回転させる

回転するようにはなったのだが、画像の左上で回転されてしまう。

実際ゲームで使う場合は、画像の中心で回転させることが多いだろう。

① 基準点を変更

SpriteData 構造体の center で指定した位置が回転の中心となる。

そこで、center を指定しなかった場合は画像の中心になるようにしよう。

まず、初期値をありえない値にする。

<Sprite.h>

```
struct SpriteData
{
        :
        :
    //コンストラクタ
    SpriteData()
    {
            :
            :
        center = D3DXVECTOR2(-999, -999);
        color = { 255, 255, 255, 255 };
        scale = D3DXVECTOR2(1, 1);
        angle = 0.0f;
    }
```

そして、ありえない数値のままだったら、center を画像の中心（幅と高さの半分）にする。

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
            :
            :

    //基準点
    D3DXVECTOR3 center = data->center;
    center.z = 0;
    if (center.x == -999 && center.y == -999)      //未設定の場合は画像の中心
    {
        center.x = [redacted];
        center.y = [redacted];
    }
```

これで、画像の中心で回転する。

そのかわり、表示する位置までずれてしまう。

②　表示位置を修正

基準点をずらしたせいで画像が左上に移動してしまったので、正しい位置に戻すため右下に移動させる。

基準点で指定した分だけ移動させればよいが、拡大縮小してる場合はその分を考えなければならない。

移動させるには、移動行列を作って回転や拡大縮小の行列に合成する。

移動行列を作る関数は **D3DXMatrixTranslation** で、XYZ 方向の移動量を指定する。

<Sprite.cpp>

```
void Sprite::Draw(SpriteData* data)
{
        :
        :
    //回転行列
    D3DXMATRIX matRotate;
    D3DXMatrixRotationZ(&matRotate, D3DXToRadian(data->angle));

    //移動行列
    D3DXMATRIX matTrans;
    D3DXMatrixTranslation(&matTrans,
        data->pos.x + [redacted] * [redacted],
        data->pos.y + [redacted] * [redacted], 0);

    //合成行列
    D3DXMATRIX matWorld = matScale * matRotate * matTrans;

    //基準点をさらに移動
    center.x += data->pos.x;
    center.y += data->pos.y;

    //スプライトに行列を適用
    pSprite->SetTransform(&matWorld);
```

これで、好きなように画像を表示することができるようになった。