DirectX 資料③

エラー対策編

ゲームエンジニア科２年

1. 初期化処理のタイミング

現在、ウィンドウプロシージャには「WM_CREATE」と「WM_DESTROY」の処理が書いてある。

しかし DirectX の特徴は「Windows に頼らないプログラム」なので、なるべくウィンドウプロシージャに頼りたくはない。

そこで、WM_CREATE 時の処理は WinMain に移動させよう。

<Main.cpp>

```
int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst, LPSTR lpsCmdLine, int nCmdShow)
{
    MSG msg;

    //ウィンドウクラスの登録
    if (!InitApp(hCurInst))
        return FALSE;

    //ウィンドウ生成
    if (!InitInstance(hCurInst, nCmdShow))
        return FALSE;

    //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();

    // メッセージを取得
    ZeroMemory(&msg, sizeof(msg));
    while (msg.message != WM_QUIT)
    {
```

このままだとウィンドウハンドルが分からなくてエラーになるので、ウィンドウハンドルをグローバルにしてしまおう。

Main.cpp だけで使えればいいので、extern は使わなくてよい。

```
//----------------グローバル変数----------------
TCHAR   szClassName[] = "OriginalGame";   //ウィンドウクラス名
Game    *game;                            //ゲームオブジェクト
HWND    hWnd;                             //ウィンドウハンドル
```

```
//-----------------------------------------
// ウィンドウの生成
//-----------------------------------------
BOOL InitInstance(HINSTANCE hInst, int nCmdShow)
{
    //ウィンドウハンドル
    HWND hWnd;

    //クライアント領域サイズから、ウィンドウサイズを計算
```

これで、プロシージャに書いていた処理は消してしまえる。

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wp, LPARAM lp)
{
    switch (msg)
    {
    //■初期設定
    case WM_CREATE:
            :
            :
        break;

    //■ウィンドウが消された
    case WM_DESTROY:

        //ゲームオブジェクトの開放
```

WM_DESTROY の時の処理は残っているが、これはプログラム終了時のものなので、
速度的を気にするものでもないし、このままでいいかなと。

ちなみに、今回の修正は「プロシージャに書かない方が気持ちいいかな？」くらいのものなので、
そのままでも特に問題があるわけではない。

2. エラー処理

プログラムを書けば必ずエラーが出るものである。エラーの無いプログラムなんて存在しない。
大事なのは**「エラーが出たときどうするか」**。

①　わざとエラーを発生

Title.cpp で「number.png」という画像ファイルを読み込んでいる。
これを、何でもいいので違うファイル名にしてみよう。

実行してみると、そんな名前のファイルは存在しなので「テクスチャの作成に失敗しました」とメッセージが出る。

読み込み時に、失敗したらメッセージを表示するように作ってあるからだ。
これは良い。

しかし、OK ボタンを押すと、プログラムが途中で止まってしまう。
画像ファイルが存在しなかったのに、そのまま無理やりプログラムを進めて画像を表示しようとしたからだ。
これは良くない。

画像ファイルが存在しなかった時点でゲームを続行するのは不可能なので、その場でプログラムを終了するべきである。

②　失敗したことを呼び出し元に伝える

読み込めなかった時点で PostQuitMessage 関数を使えばプログラムを終了させることができる。

しかし、今後複雑なプログラムを作っていくと、**エラーが出る可能性のある処理**はどんどん増えていく。

その都度、強制終了関数を書くのはめんどくさい。

そこで、基本的には**「失敗した」という情報を戻り値で返す**だけにする。

要するに失敗したことを上司に報告し、その結果どうするかは上司に判断させる。

ここで使われるのが **HRESULT** 型だ。

つまり、Sprite クラスの Load 関数をこんな感じにする。

<Sprite.cpp>

```
HRESULT 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", "", MB_OK);
        return E_FAIL;
    }

    //「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, NULL, NULL, NULL, &pTexture)))
    {
        MessageBox(0, "テクスチャの作成に失敗しました", fileName, MB_OK);
        return E_FAIL;
    }

    //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;
    return S_OK;
}
```

戻り値の型を HRESULT 型にし、失敗したら E_FAIL、最後まで問題無ければ S_OK を返すようにする。

あくまで結果を報告しているだけで、失敗したからと言って何をしているわけでもない。

（当然、Sprite.h も修正する必要があるので自分で修正すること）

③　さらに上へ

さて、部下から「失敗した」と報告を受けた上司はどうしたらいいだろうか。

プログラムの場合、**さらに上の上司に報告して指示を仰ぐ**のが正しい。

今回 Sprite::Load を修正したわけだが、それを呼び出しているのは Title::Load だ。

この処理内で何か失敗した場合はその結果を戻り値で返す。

<Title.cpp>

```
HRESULT Title::Load()
{
    if (FAILED(sprite.Load("number1.png")))
    {
        return E_FAIL;
    }
    return S_OK;
}
```

ロードに失敗したら E_FAIL を返している。

ここで注意しなければいけないのは、この関数は UnitBase で宣言した関数なので、
UnitBase の Load 関数の戻り値も HRESULT 型に変更しておく必要がある。
これを忘れると、Title::Load がオーバーライドにならないので、シーンクラスから呼ばれなくなってしまう。
そして、戻り値を void 以外にした場合は何かを返さなければならない。
この Load 関数が呼ばれるのはロードするものが無い場合なので、問題が起きることは無いだろうから S_OK を返せばいいだろう。

④　さらに・・・

さて、Title クラスの上司は誰だろうか？
各ユニットの Load 関数は、SceneBase クラスの Load 関数から呼ばれている。

Title::Load が失敗したことを知った SceneBase::Load はどうするべきか･･･

答えは想像つくだろう。そこでは何もせず、部下の失敗を上司に報告する。

実際の会社の場合にこれでいいのかはさておき、プログラムの場合は**中間管理職に責任は負わせない**方が良い。

<SceneBase.cpp>

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

さらに、これを呼び出しているのは Game クラスの Load 関数。
もう、説明はいらないだろう。

<Game.cpp>

```
//―――――――――――――――――――――――――――――――
//  読み込み処理
//―――――――――――――――――――――――――――――――
HRESULT Game::Load()
{
    if (FAILED(scene[g_gameScene]->Load()))
    {
        return E_FAIL;
    }
    return S_OK;
}
```

⑤　最終決断

さて、Game::Load を呼んでいるのは Main.cpp の WinMain 関数だ。

WinMain はどこから呼ばれるものでもなく、言わば**社長**にあたる関数である。

社長は責任を持って対処しよう。

WinMain 関数が終わればプログラムが終わるので、部下から失敗の報告を受けたら潔く return しよう。

<Main.cpp>

```
int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst, LPSTR lpsCmdLine, int nCmdShow)
{
        :
        :

  //読み込み処理
  if (FAILED(game->Load()))
  {
      return FALSE;
  }
```

これで、ファイルが見つからなかった場合はメッセージを表示後プログラムが終了するようになった。

途中の関数ではエラーの報告をするだけに留めることで、「エラーが起きたときどうするか」という対処法を
１か所にまとめることができた。この方がシンプルで扱いやすいだろう。

また、すでにエラーが起きたことを上に伝える形ができたので、今後エラーが起きた場合は
どこからでも E_FAIL を返すだけで正しく対処してくれるようにもなった。

⑥　他のところも

最後に修正したところの上の行を見てみよう。Game::InitD3d 関数を呼んでいる。
この関数も戻り値が HRESULT 型になっていたはずだが、呼び出し側では何の対処もしていない。

部下から報告を受けても社長が無視してる状態なので好ましくない。

Direct3D の初期化に失敗した場合もゲームは続けられないので、その場合もプログラムを終了するようにしよう。

<Main.cpp>

```
int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst, LPSTR lpsCmdLine, int nCmdShow)
{
        :
        :
    //ゲームオブジェクトを作成
    game = new Game;

    //Direct3D の初期化
    if (FAILED(game->InitD3d(hWnd)))
    {
        return FALSE;
    }

    //読み込み処理
```

⑦　どうせなら全部

UnitBase.h を見てみよう。

Load 関数の戻り値は HRESULT 型だが、Update、Hit、Render の戻り値は void 型になっている。

ということは、今後 UnitBase クラスを継承するクラスを作った場合、かならず Load 関数は HRESULT、

その他は void にしなければならない。

こういう仕様だと間違える可能性が高い。

（ちなみに、型を間違えた場合はオーバーライドにならないだけなのでコンパイルエラーは出ず、

しかしその関数は呼ばれないという分かりにくい問題が起こる）

そもそも Update や Render でもエラーが起きる可能性は 0 ではないので、

いっそのこと戻り値の型は全部 HRESULT 型にしてしまった方が無難だろう。

<UnitBase.h>

```
class UnitBase
{
        :
        :
    //更新処理
    virtual HRESULT Update();

    //衝突判定処理
    //引数：pTarget 判定する相手ユニット
    virtual HRESULT Hit(UnitBase* pTarget);

    //描画処理
    //引数：なし
    virtual HRESULT Render();
        :
        :
};
```

これを変えたら、もちろん UnitBase.cpp も変更する必要がある。
何かを返さなければならないので、**return S_OK;**を追加しておこう。

そして、UnitBase を継承している Title クラスも変更が必要。
さらに、各ユニットから失敗報告を受けるシーンクラス（SceneBase）も変更し、
そこから報告を受ける Game クラスも変更し･･･

最後は Main.cpp で、エラー時は終了するようにする。

<Main.cpp>

```
int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst, LPSTR lpsCmdLine, int nCmdShow)
{
        :
        :
            //ゲームの更新
            if (FAILED(game->Update()))
            {
                return FALSE;
            }

            //衝突判定
            if (FAILED(game->Hit()))
            {
                return FALSE;
            }

            //ゲーム画面の描画
            if (FAILED(game->Render()))
            {
                return FALSE;
            }
        }
    }

    return (int)msg.wParam;
}
```

ちょっとやりすぎな感じがしないでもないが、どこかで問題が発生したら全て社長まで伝わる
「報告・連絡・相談」が徹底された風通しの良いプログラムになった。
ためしにどこかで **return E_FAIL;**を書いてみて正しく終了することを確認しよう。

最後に注意点として、**エラーが発生したからと言って必ずプログラムを終了させなければならないわけではない**。

たとえば、「コントローラーが繋がっていない」という状況があったとしたら、
その場合はキーボードでプレイできるようにすることも可能だ。
ほかにも、たまたま今はうまくいかなかったけど次のフレームでは問題ない場合もある。

そういう時までプログラムを終了させてしまうのは”おせっかい”だ。
E_FAIL を返せばプログラムが終了するようになったので、「ここで失敗した場合に E_FAIL を返すべきか」を
ちゃんと考えるように。

3. 解放忘れチェック

new を使ったら必ず delete しなければならない。

と言っても、どうしても忘れてしまうことがある。

① チェックコード

Main.cpp に次のコードを追加しよう。

<Main.cpp>

```
//---------------インクルード----------------------
#include "Global.h"
#include <time.h>
#include "Game.h"

//---------------メモリリーク検出---------------
#if _DEBUG
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#define new new(_NORMAL_BLOCK, __FILE__, __LINE__)
#endif
        :
        :
        :

//―――――――――――――――――――――――――――――
//エントリーポイント
//―――――――――――――――――――――――――――――
int WINAPI WinMain(HINSTANCE hCurInst, HINSTANCE hPrevInst, LPSTR lpsCmdLine, int nCmdShow)
{
#if _DEBUG
    _CrtDumpMemoryLeaks();
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    MSG msg;

    //ウィンドウクラスの登録
    if (!InitApp(hCurInst))
        return FALSE;
```

これは、new を使ったのに delete してない場合は教えてくれるオマジナイだ。

② 試してみよう

ウィンドウプロシージャの WM_DESTROY 時の処理でゲームクラスのオブジェクトを delete してる。

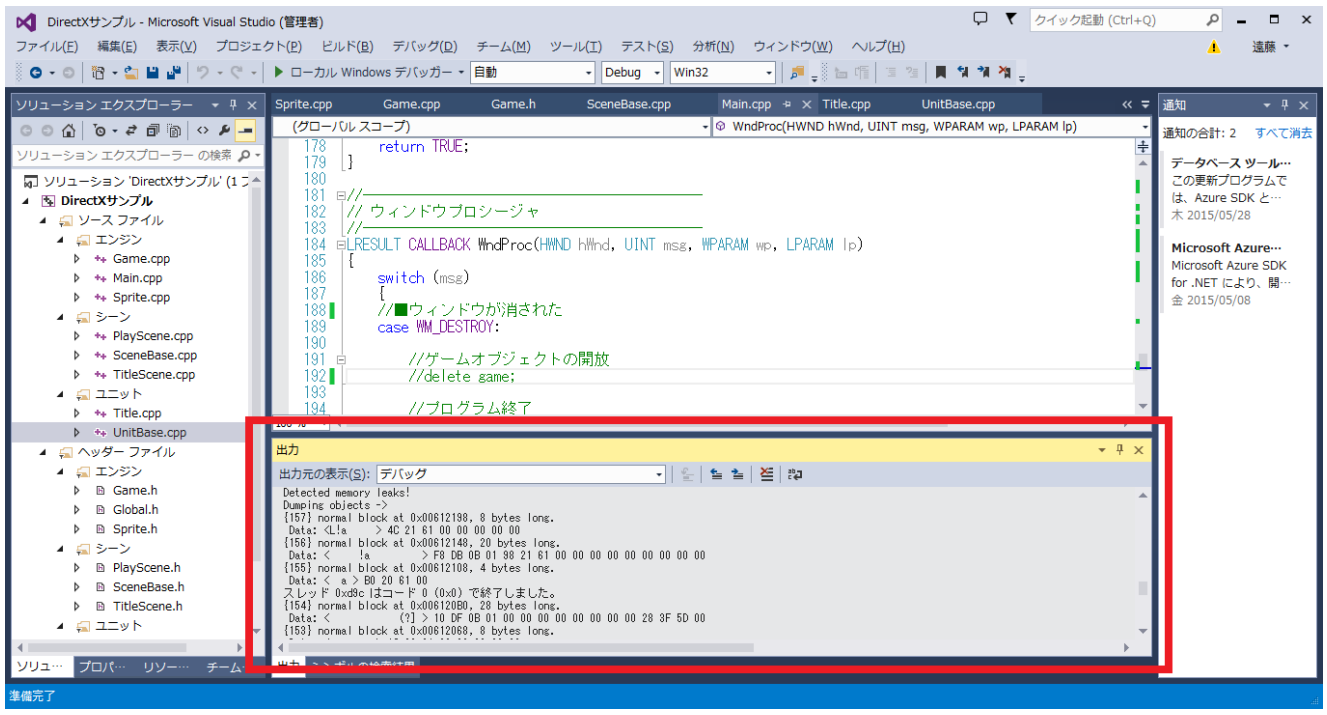
ためしにこれをコメントアウトして実行してみよう。

特に実行結果に変わったところは無いはず。

では、ウィンドウの×ボタンを押してプログラムを終了しよう。

そして、VisualSutudio の「出力」ウィンドウを見てみよう。

（出力ウィンドウがない場合は、メニューの「表示」の中の「出力」を選べば出てくる）



こんなことが表示されただろう。

```
Detected memory leaks!
Dumping objects ->
{157} normal block at 0x00612198, 8 bytes long.
 Data: <L!a      > 4C 21 61 00 00 00 00 00
{156} normal block at 0x00612148, 20 bytes long.
 Data: <     !a         > F8 DB 0B 01 98 21 61 00 00 00 00 00 00 00 00 00
{155} normal block at 0x00612108, 4 bytes long.
 Data: <  a > B0 20 61 00
スレッド 0xd9c はコード 0 (0x0) で終了しました。
{154} normal block at 0x006120B0, 28 bytes long.
 Data: <            (?] > 10 DF 0B 01 00 00 00 00 00 00 00 00 28 3F 5D 00
{153} normal block at 0x00612068, 8 bytes long.
 Data: <  a      > 1C 20 61 00 00 00 00 00
{152} normal block at 0x00612018, 20 bytes long.
スレッド 0x3960 はコード 0 (0x0) で終了しました。
スレッド 0x2280 はコード 0 (0x0) で終了しました。
 Data: <8   h a  !a  !a > 38 DF 0B 01 68 20 61 00 08 21 61 00 0C 21 61 00
D:¥ ○○○○¥DirectX サンプル¥main.cpp(65) : {151} normal block at 0x00611FD0, 12 bytes long.
 Data: <      a H!a > 00 08 A6 00 18 20 61 00 48 21 61 00
Object dump complete.
```

一番上の**「Detected memory leaks!」**というのが**「delete し忘れてるよ！」**という意味。

その後にいろいろと出てきているが、Game クラスを解放しなかったせいで各シーンが解放されず、

そのせいで各シーンに出てくるユニットも解放されず……ということでいろいろと出てきている。

最後の方に「○○○¥¥main.cpp(65)：～～～」と書いてある。

これは『main.cpp の 65 行目で作ったオブジェクトを解放してない』と言っている。

このメッセージをダブルクリックすると、new Game と書いてある行に飛ぶ。

これで delete し忘れる可能性はだいぶ減るだろう。

しかし、**このオマジナイは万能ではない**ので注意しよう。

あくまで『保険』くらいに思っておくように。

なお、コメントアウトした delete の行は元に戻しておくこと。

4. 安全な解放

① NULL ポインタ

new を使ったら delete するのは常識だが、さらに言うと delete したあとポインタに NULL を代入しておくのが安全である。

メモリの 0 番地は使えない仕様になっているためだ。

つまり――

```
（例）
int *a;
a = new int;
       :
       :
delete  a;
a = NULL;
```

これが、安全な delete の方法になる。

②　マクロ作成

しかし、「delete 後に必ず NULL を代入しよう」と決めても、絶対忘れてしまう。

そこで、次のようなマクロを作ってしまう。

<Global.h>

```
//------------------マクロ-----------------
#define    SAFE_DELETE(p)     {delete (p);    (p) = NULL;}
```

これを作っておけば、例えば次のように書いた場合――

```
SAFE_DELETE(a);
```

こう書いたのと同じ意味になる。

```
delete  a;
a = NULL;
```

今後 delete を使う場合は SAFE_DELETE の方を使うようにしよう。

さっそく、delete を使っているところを直しておこう。

<Main.cpp>

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wp, LPARAM lp)
{
    switch (msg)
    {
    //■ウィンドウが消された
    case WM_DESTROY:

        //ゲームオブジェクトの開放
        SAFE_DELETE(game);

        //プログラム終了
        PostQuitMessage(0);
```

③　配列の場合

ポインタに配列を new した場合は **delete[]**を使うはずだ。

```
 (例)
int* b = new int[10];
    :
    :
delete[]  b;
```

この場合は先ほどのマクロは使えないので、配列用の解放マクロも作っておこう。

<Global.h>

```
//-------------------マクロ-------------------
#define SAFE_DELETE(p)          {delete (p);    (p) = NULL;}
#define SAFE_DELETE_ARRAY(p)    {delete[] (p);  (p) = NULL;}
```

今のところコレを使う箇所は無いが、今後動的配列を使った場合は SAFE_DELETE_ARRAY で解放するようにしよう。

④　Release も

LP で始まる型名の変数を使ったら**○->Release()**というように解放することは説明した。

これも、その後に NULL を代入した方が安全だ。

マクロを追加しよう。

<Global.h>

```
//-------------------マクロ-------------------
#define SAFE_DELETE(p)          {delete (p);    (p) = NULL;}
#define SAFE_DELETE_ARRAY(p)    {delete[] (p);  (p) = NULL;}
#define SAFE_RELEASE(p)         {(p)->Release;  (p) = NULL;}
```

しかし、これだとちょっと問題がある。

既に解放したものをもう一度解放しようとするとプログラムが止まってしまう。

そこで、変数の値が既に NULL の場合は無視するようにしよう。

逆に言えば、**変数の値が NULL じゃない場合のみ解放する**ようにすればいい。

<Global.h>

```
//-------------------マクロ-------------------
#define SAFE_DELETE(p)          {delete (p);    (p) = NULL;}
#define SAFE_DELETE_ARRAY(p)    {delete[] (p);  (p) = NULL;}
#define SAFE_RELEASE(p)         { if(p != NULL) { (p)->Release(); (p) = NULL; } }
```

これで安心して解放処理ができる。

既に Release を使ってる場所が 4 か所あるので SAFE_RELEASE に書き換えておこう。

<Game.cpp>

```
Game::~Game()
{
        :
        :
    //DirectX 解放
    [redacted];
    [redacted];
}
```

<Sprite.cpp>

```
Sprite::~Sprite()
{
    [redacted];
    [redacted];
}
```

今後、LP で始まる型の変数を使用したら SAFE_RELEASE で解放するよう徹底しよう。