3D アクションゲームプログラミング

○評価要件

☑ローディング速度の高速化

○概要

今回はリソースマネージャーを実装します。
3D モデルやテクスチャといったデータはファイルから読み込みを行っています。
現状ではキャラクターやステージを生成する毎に 3D モデルを読み込んでいますが、本来、同じキャラクターを生成する場合、何度も同じ 3D モデルを読み込みする必要はありません。
一度読み込んだことがある場合は読み込み済みのリソースを取得できるようにします。

今回はリソースマネージャーを実装する前と実装後で違いがわかるように、スライムを複数体配置してローディング時間の違いを体感しましょう。

○リソースマネージャー

リソースマネージャーとは名前の通り、リソースを管理するものです。
ゲームではテクスチャや 3D モデルなどの大量のリソースを使用します。
リソースファイルの読み込みは処理時間が長く、リソースデータの量が多くなればばるほどメモリ領域も圧迫されてしまいます。

リソースマネージャーではリソースファイルの読み込み機能を提供し、既に読み込み済みのリソースがある場合は新規でファイル読み込みをせず、読み込み済みのリソースを返すようにします。

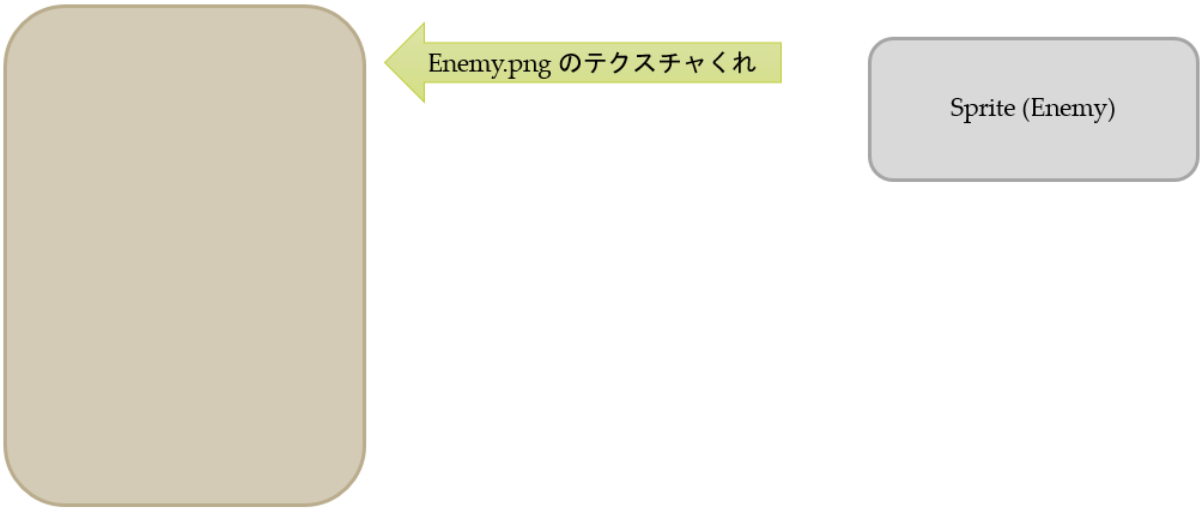
また、読み込んだリソースがゲーム内で使用されなくなった場合に自動でリソースの解放処理を行い、リソースマネージャーの管理から外れるようになるようにします。

この管理をするために「参照カウンタ」という方法を使って実装します。
オブジェクト自身の生存期間を管理するための方法です。
オブジェクトが参照されるとカウンタが増え、参照されなくなるとカウンタが減ります。
参照カウンタが 1 以上あるということは誰かがオブジェクトを参照しているということがわかり、
参照カウンタが 0 になるということは誰もオブジェクトを参照していないことがわかります。
つまり、参照カウンタが 0 になればオブジェクトを削除しても誰も困らないということです。

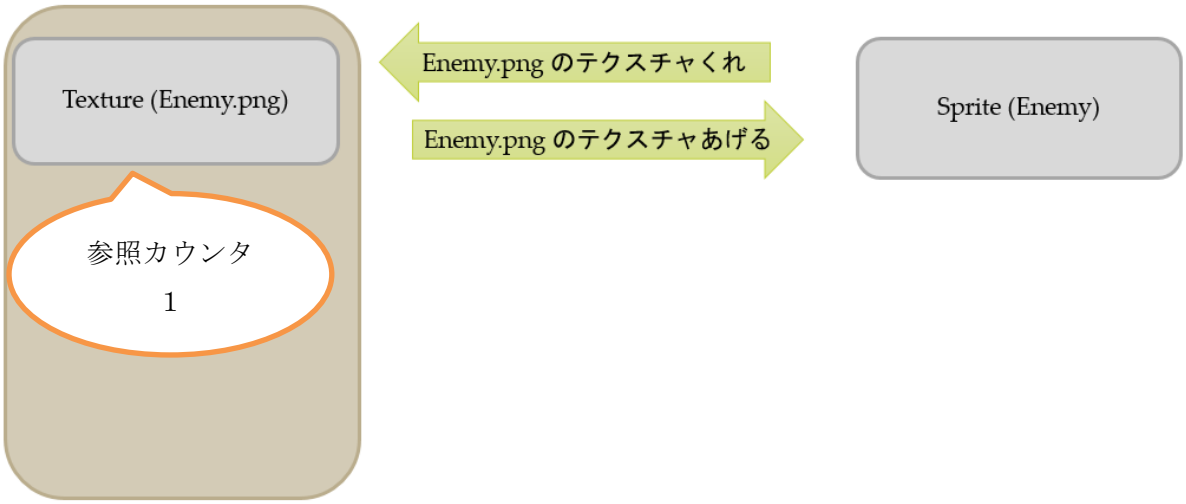
以上のことを踏まえてリソース読み込みの流れを見ていきましょう。

○リソース読み込みの流れ

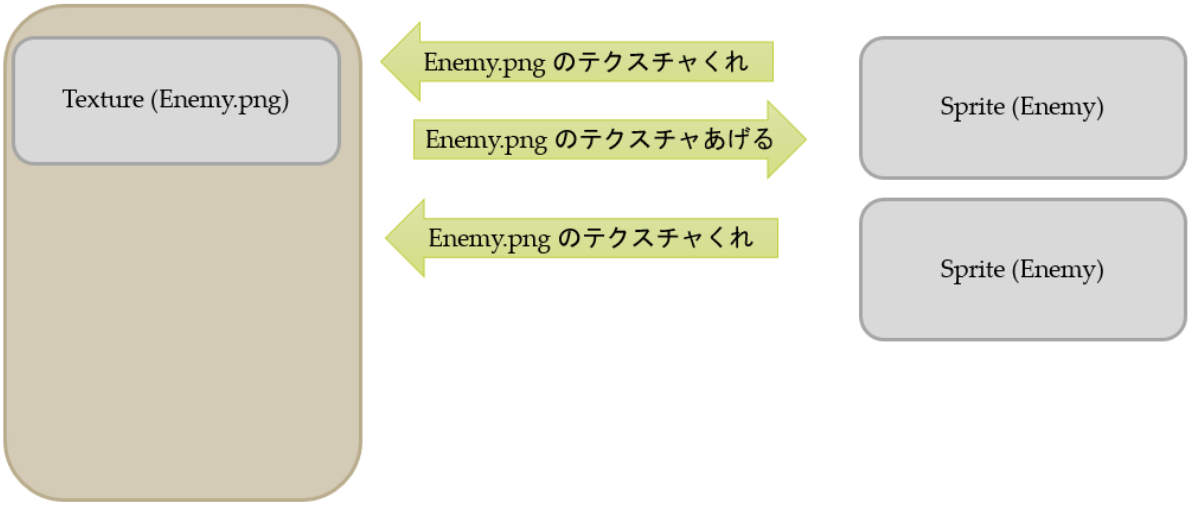
① リソースマネージャーに対して Enemy.png を要求。



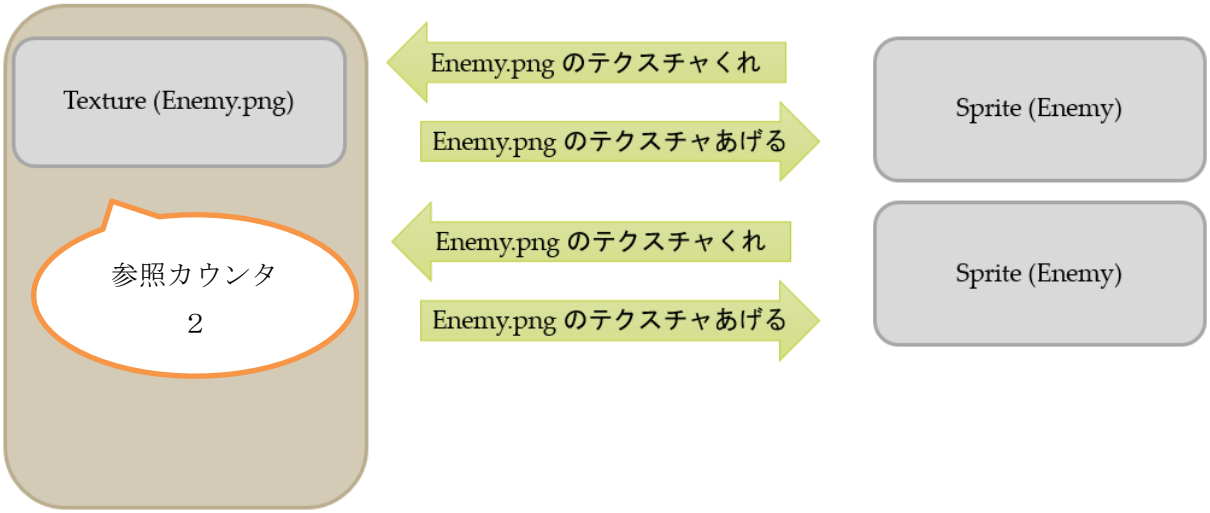
② リソースマネージャーは Enemy.png を持っていないので新規読み込みをして返す。



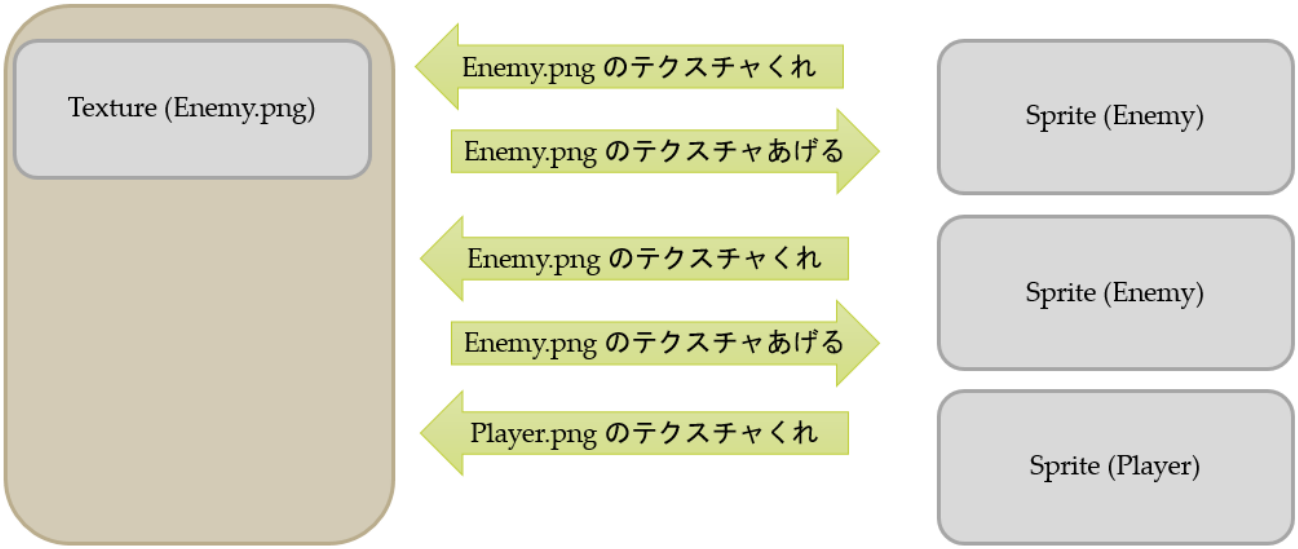
③ リソースマネージャーに対して Enemy.png を要求。



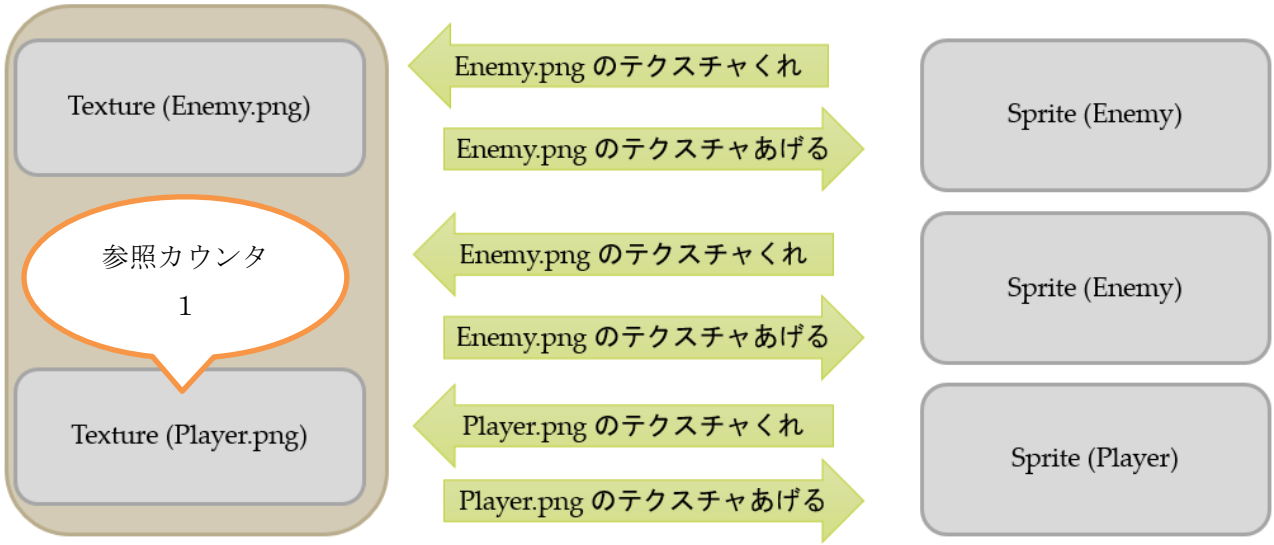
④ リソースマネージャーは Enemy.png を持っているので返す。



⑤ リソースマネージャーに Player.png を要求。

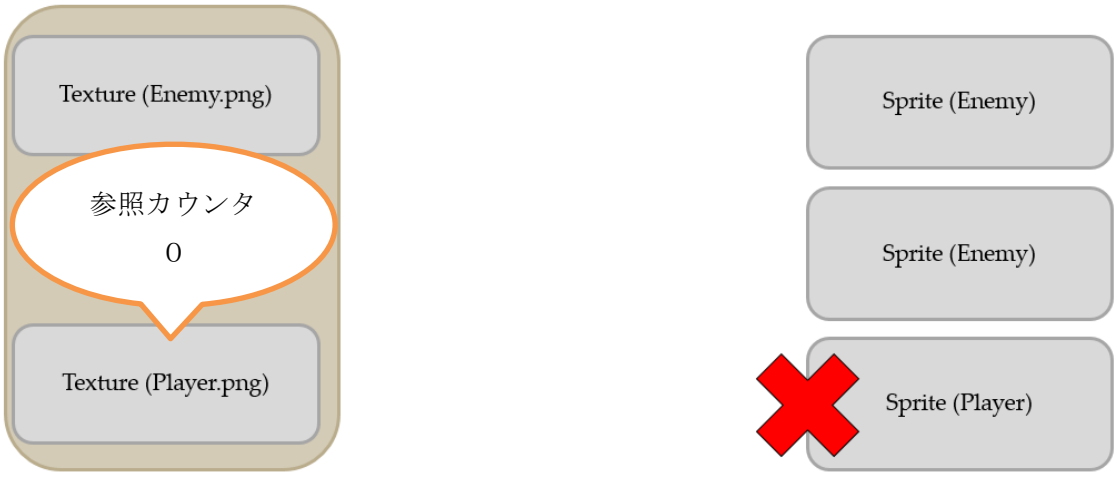


⑥ リソースマネージャーは Player.png を持っていないので新規読み込みをして返す。

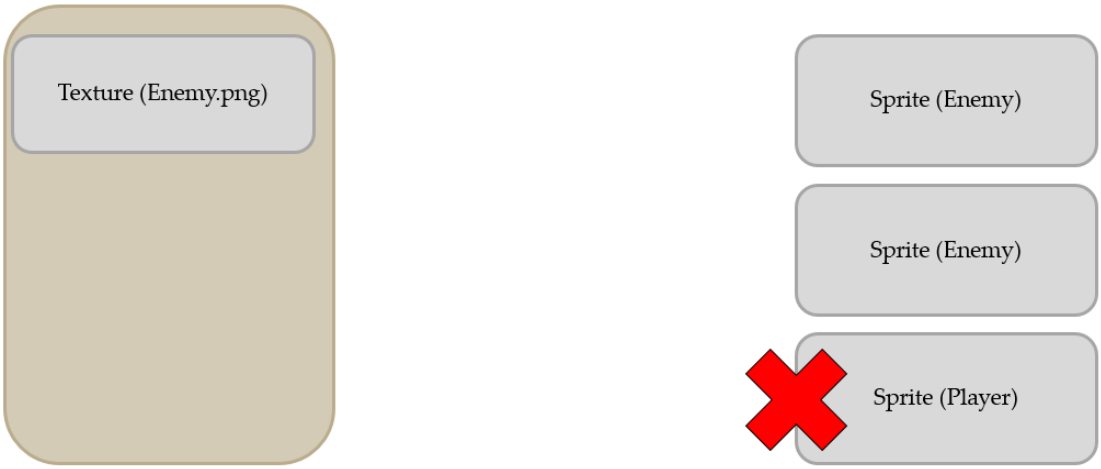


○リソース解放の流れ

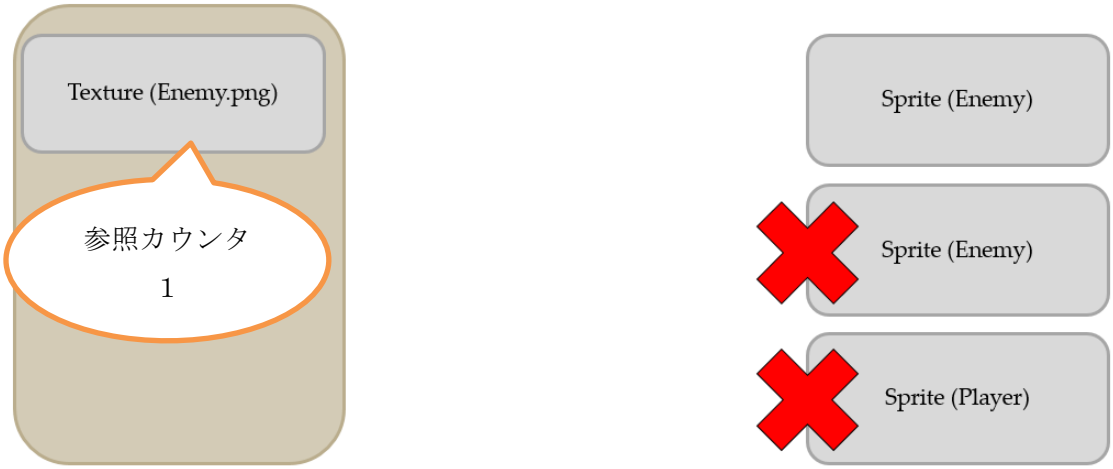
①Player.png を保持するスプライトが削除され、リソースマネージャーの参照カウンタが減る。



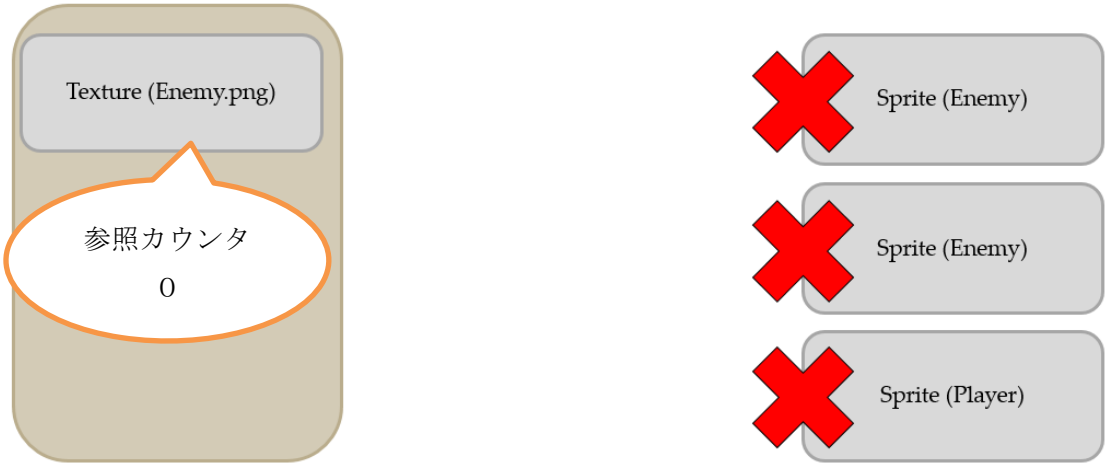
②リソースマネージャーの Player.png の参照カウンタが 0 になったので削除する。



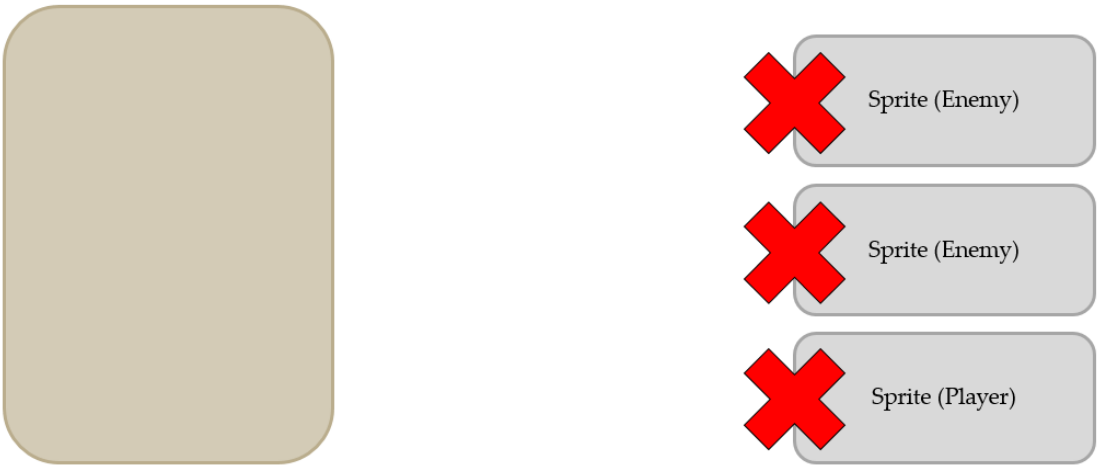
③Enemy.png を保持するスプライトが削除され、リソースマネージャーの参照カウンタが減る。



④Enemy.png を保持するスプライトが削除され、リソースマネージャーの参照カウンタが減る。



⑤リソースマネージャーの Enemy.png の参照カウンタが 0 になったので削除する。



○スマートポインタ

今までスマートポインタをあえて利用せず、new, delete によってメモリの確保と解放を行ってきました。
今回はリソースマネージャーを実装するにあたってスマートポインタを利用します。
スマートポインタとは C++11 から実装されたメモリの解放を自動で削除してくれる機能です。
これにより、メモリの解放忘れや、不適切な対象への delete などがなくなることが期待できます。

スマートポインタは 3 種類存在し、それぞれの特性があります。

- unique_ptr<T>
- shared_ptr<T>
- weak_ptr<T>

○unique_ptr<T>

・一人だけしかメモリを所有できないルールのポインタ。

```
void Hoge()
{
    std::unique_ptr<int> p = std::make_unique<int>();
    *p = 3;
    std::unique_ptr<int> a = p; // ※コンパイルエラー
    std::unique_ptr<int> b = std::move(p); // 所有権を渡す
    *b = 4;  // 所有権があるので実態を操作できる
    *p = 5; // すでに所有権がなのでエラー
}
```

○shared_ptr<T>

・複数の人で同じメモリを共有する。
・所有している人の数を参照カウンタで管理している。
・参照している人がいなくなると破棄される。

```
void Hoge()
{
    // a がメモリを保持したので参照カウンタが1になる。
    std::shared_ptr<int> a = std::make_shared<int>();
    *a = 3;
    { // b がメモリを保持したので参照カウンタが2になる。
        std::shared_ptr<int> b = a;
        *b = 4;
    } // b が破棄され、参照カウンタが1になる。
    // a が破棄され、参照カウンタが0になる。
}   // 参照カウンタが 0 になったのでメモリが破棄される。
```

・相互参照をするとメモリリークしてしまう。

```
struct Foo
{
    std::shared_ptr<Bar> p;
};

struct Bar
{
    std::shared_ptr<Foo> p;
};

void Hoge()
{
    std::shared_ptr<Foo> foo = std::make_shared<Foo>();
    std::shared_ptr<Bar> bar = std::make_shared<Bar>();

    foo->p = bar;
    bar->p = foo;
}
// 循環参照をしているとうまくデストラクタが呼ばれなくてメモリリークする。
```

○weak_ptr<T>

・shared_ptr<T>の弱点を解決する。

・shared_ptr<T>を保持することができるが、参照カウンタが増えない。

```
struct Foo
{
    std::shared_ptr<Bar> p;
};
struct Bar
{
    std::weak_ptr<Foo> p;
};

void Hoge()
{
    std::shared_ptr<Foo> foo = std::make_shared<Foo>();
    std::shared_ptr<Bar> bar = std::make_shared<Bar>();

    foo->p = bar;
    bar->p = std::weak_ptr<Foo>(foo);
}
```

・shared_ptr<T>のリンクが切れていないか（参照カウンタが 0 でないか）確認できる。

```
void Hoge()
{
    std::weak_ptr<int> a;
    {
        std::shared_ptr<int> b = std::make_shared<int>(); // bの参照カウンタは1
        a = b;  // weak_ptrは参照カウンタは増えないので参照カウンタは1
        assert(a.expired() == true);   // bの参照カウンタが1のためリンクが切れていないはず
    }   // bが破棄されるため、参照カウンタは0となる
    assert(a.expired() == false);  // bの参照カウンタが0のためリンクが切れている
}
```

○リソース検索

リソースマネージャーの中に既に読み込み済みのリソースが存在するか素早く検索することもリソース管理に必要な要素の一つです。

std::map<key, value> を利用することで簡単に実装することができます。
std::map<key, value> はキーに対する値の組（ペア）を要素とするコンテナクラスです。

リソースマネージャーではキーを「ファイルパス」、値を「リソース」にすることで、ファイルパスに対応するリソースを簡単に検索するようにします。

○ローディング時間の確認

リソースマネージャーを実装する前に、仮でシーン初期化時に敵を 50 体配置してみましょう。

SceneGame.cpp

```
---省略---

// 初期化
void SceneGame::Initialize()
{
    ---省略---

    // エネミー初期化
    EnemyManager& enemyManager = EnemyManager::Instance();
#if 0
    EnemySlime* slime = new EnemySlime();
    slime->SetPosition(DirectX::XMFLOAT3(0, 0, 5));
    enemyManager.Register(slime);
#else
    // 仮でシーンに敵を大量配置してみる
    for (int i = 0; i < 50; ++i)
    {
        EnemySlime* slime = new EnemySlime();
        slime->SetPosition(DirectX::XMFLOAT3(i * 2.0f, 0, 5));
        enemyManager.Register(slime);
    }
#endif
}
```

実装できたら実行確認をしてみましょう。
ローディング時間がかなり長くなっているはずです。
この問題解決のため、リソースマネージャーを実装していきます。

○リソースマネージャー実装

上記で解説した std::map とスマートポインタを使ってリソースマネージャーを実装しましょう。
ResourceManager.h と ResourceManager.cpp を作成し、下記プログラムコードを記述しましょう。

ResourceManager.h

```
#pragma once

#include <memory>
#include <string>
#include <map>
#include "Graphics/ModelResource.h"

// リソースマネージャー
class ResourceManager
{
private:
    ResourceManager() {}
    ~ResourceManager() {}

public:
    // 唯一のインスタンス取得
    static ResourceManager& Instance()
    {
        static ResourceManager instance;
        return instance;
    }

    // モデルリソース読み込み
    std::shared_ptr<ModelResource> LoadModelResource(const char* filename);

private:
    using ModelMap = std::map<std::string, std::weak_ptr<ModelResource>>;

    ModelMap        models;
};
```

ResourceManager.cpp

```
#include "Graphics/Graphics.h"
#include "ResourceManager.h"

// モデルリソース読み込み
std::shared_ptr<ModelResource> ResourceManager::LoadModelResource(const char* filename)
{
    // モデルを検索
    ModelMap::iterator it = models.find(filename);
    if (it != models.end())
    {
        // リンク（寿命）が切れていないか確認
        if (!it->second.expired())
        {
            // 読み込み済みのモデルリソースを返す
            return it->second.lock();
```

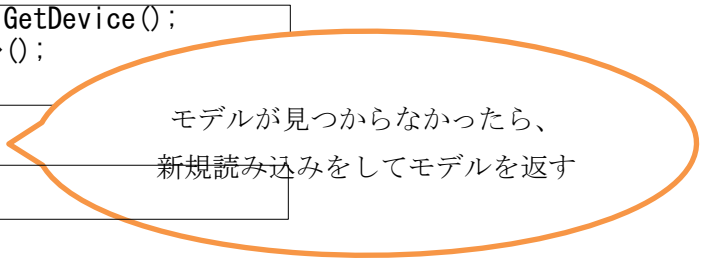
マネージャー内に
指定ファイルパスの
モデルが存在するか検索し、
見つかったらそのモデルを返す

```
        }
    }

    // 新規モデルリソース作成＆読み込み
    ID3D11Device* device = Graphics::Instance().GetDevice();
    auto model = std::make_shared<ModelResource>();
    model->Load(device, filename);

    // マップに登録
    models[filename]   = model;

    return model;
}
```



モデルリソースの読み込み処理をしている個所の変更をしましょう。

Graphics/Model.cpp

```
---省略---
#include "ResourceManager.h"

// コンストラクタ
Model::Model(const char* filename)
{
    // リソース読み込み
    resource = std::make_shared<ModelResource>();
    resource->Load(Graphics::Instance().GetDevice(), filename);
    resource = ResourceManager::Instance().LoadModelResource(filename);

    ---省略---
}
```

実装できたら実行確認をしてみましょう。
ローディング時間が短縮できていれば OK です。

今回はモデルリソースのみ実装しましたが、他のリソースも対応していきましょう。

また、今後の拡張としてオープンワールド系のゲームを作っていく場合はリソースの読み込みを非同期で行うようにする必要があります。
リソースマネージャー自身が管理するスレッドを立て、リソースが読み込みを完了しているかを確認するインターフェースが必要になってきます。

色々考えて拡張していきましょう。

お疲れさまでした。