추상화, 일반화, 다중 상속을 통한 비 다형적 동작을 다형적으로 변경

작성자: 이정모

게임 엔진을 만들던 중에 DirectX관련 자원들을 manager class를 통해 한 곳에서 관리해야 할 필요를 느꼈습니다.

그래서 ResourceManager class를 만들게 되었고 외부에서 자원에 필요할 때는 resource manager를 통해서만 얻어올 수 있게 만들었습니다.

DX 관련 자원에는 device, device context, buffer, shader, input layout, shader resource view 등 매우 많은 자료형이 존재합니다.

만약 이 자료형 그대로 사용하게 된다면 자료형마다 각각의 container를 만들어주어야 하는데 이것이 비합리적이라 생각했습니다.

```
std::unordered_map<eVertexBufferType, ID3D11Buffer > m_Buffers;
std::unordered_map<eVertexShaderType, ID3D11VertexShader> m_VertexShaders;
std::unordered_map<eInputLayoutType, ID3D11InputLayout> m_InputLayouts;
std::unordered_map<ePixelShaderType, ID3D11PixelShader> m_PixelShaders;
```

그래서 이들을 하나의 container로 관리하기로 했고 이를 위해 IResource라는 interface class를 만들어서 자료형을 통일 시켰습니다.

```
class IResource
{
public:
  IResource() {}
  virtual ~IResource() = 0 {}
};
```

그런데 또 다른 문제가 발생했습니다. interface로 일관되게 관리는 할 수 있었으나 외부에서 특정 자원을 요청하면 그 자원을 return 해줘야 하는데 이 때 interface에서 T* Get()을 사용해야 해서 특정 type에 종속적이게 되었습니다.

```
template <typename T>
class IResource
{
public:
  IResource() {}
  virtual ~IResource() = 0 {}

  virtual T* Get() = 0 {}
};
```

interface란 하부가 어떤 api나 자료형으로 구성되었는지 모르는 상태에서도 일관되게 접근하기 위해 사용하는 것인데 위처럼 T에 종속적이게 된다면 이는 결국 일관성이 깨지게 되는 것이었습니다.

container도 IResource 하나의 자료형으로만 관리하려고 했으나

결국 container 각각이 IResource<T>를 관리하게 되어버린 셈입니다.

interface에서 T에 종속적인 것을 피하기 위해서 저는 다음 단계로 void*를 사용했습니다.

```
class IResource
{
public:
  IResource() {}
  virtual ~IResource() = 0 {}

  virtual void* Get() = 0 {}
};
```

void*를 사용하게 되면 T를 노출시키지 않을 수 있기 때문에 하나의 interface로 모든 자원을 관리하자는 저의 목적에 부합했습니다.

하지만 저는 여기서 게임 엔진 사용자의 입장을 생각을 해보았습니다.
void*로 type casting을 해서 자원을 return 해준다면,
결국 사용자는 외부에서 포인터 형변환을 위해서 reinterpret_cast<>()를 사용할텐데
강제 포인터 형변환은 너무 위험하다는 생각이 들었습니다.

dynamic_cast<>()는 런타임에 부모, 자식 형변환이 가능한지 검사를 해주고
static_cast<>()는 컴파일 타임에 일반적인 형변환이 가능한지 검사를 해주는데
reinterpret_cast<>()는 검사 여부 없이 강제로 형변환을 진행하기 때문입니다.

이런 이유로 void*도 사용하고 싶지 않게 되었습니다.
그래서 마지막으로 생각한 것이 다중 상속이었습니다.

자원을 생성할 때는 ChildResource<T>로 생성합니다.

```
// iunknown -> concept
template <iunknown T>
class ChildResource : public IResource, public ParentResource<T>
{
public:
  ChildResource(T* ptr);
  ~ChildResource();
};

template<iunknown T>
inline ChildResource<T>::ChildResource(T* ptr)
  : ParentResource<T>(ptr)
{
}

template<iunknown T>
inline ChildResource<T>::~ChildResource()
{
}
```

이 class는 T를 ParentResource<T>에 전달하기 위한 class로 비어있는 틀에 가깝습니다.

생성된 자원들을 IResource로 up casting하여 interface로 관리하다가
특정 자원에 대한 요청이 들어오면 ParentResource로 dynamic_cast<>()를 하여
해당 타입을 반환해주는 방법입니다.

```
template <typename T>
class ParentResource
{
public:
  ParentResource(T* ptr);
  virtual ~ParentResource();

  T* Get();

private:
  T* m_resource;
};

template<typename T>
inline ParentResource<T>::ParentResource(T* ptr)
  :m_resource(ptr)
{
  m_resource->AddRef();
}

template<typename T>
inline ParentResource<T>::~ParentResource()
{
  m_resource->Release();
}

template<typename T>
inline T* ParentResource<T>::Get()
{
  return m_resource;
}
```

ParentResource<T>는 실제 type을 숨겨 놓기 위한 class입니다.
이 class는 interface도 아니기 때문에 T를 노출시켜도 상관이 없습니다.

결국 외부에서 resource manager에게 자원을 요청한다면

```
template<iunknown T>
inline T* ResourceManager::GetResource(unsigned int handle)
{
  auto it = m_handleToResourceTable.find(handle);
```

```
    assert(it != m_handleToResourceTable.end());

    // 다른 부모(ParentResource)로 캐스팅해서 자원에 대한 포인터 반환
    ParentResource<T>* parentResource = dynamic_cast<ParentResource<T>*>(it->second);
    assert(parentResource != nullptr);

    return parentResource->Get();
}
```

핸들에 해당하는 자원을 찾고

그 자원은 IResource로 관리하고 있기 때문에

dynamic_cast<>()를 통해 다른 부모인 ParentResource<T>로 형변환 하여 자원을 return 했습니다.

마무리하자면

자원들의 자료형이 너무나 다양해서 한번에 관리하기 어려웠기 때문에

interface를 사용해서 일관되게 관리하려 했고

interface에서는 특정 타입에 종속적이면 안되기 때문에 Get() 함수의 반환형을 T*에서 void*로 바꿨으나

이 또한 엔진 사용자가 외부에서 reinterpret_cast<>()를 해야 해서 위험하다고 판단했고

다중 상속을 통해 생성은 ChildResource<T>로 관리는 IResource로 하고

외부에서 자원이 필요할 때는 ParentResource<T>로 dynamic_cast<>()하여 자원을 return했습니다.

이상입니다.