Win32 API FrameWork
Portfollio 정리

01. Core
    01. Timer
    02. Camera
    03. Path
    04. Input
    05. Math
    06. Core
    07. Ref
02. Scene
    01. Layer
    02. InGameScene
    03. Scene
    04. SceneManager
03. Object
    01. MoveObj
        01. Player
        02. Minion
        03. Bullet
    02. StaticObj
        01. Stage
    03. Obj
04. Resource
    01. ResourceManager
    02. Texture
05. Collider
    01. Collider
    02. ColliderPixel
    03. ColliderRect
    04. ColliderSphere
    05. ColliderManager
06. Animation
    01. Animation

- 가장 기초적인 게임의 흐름을 알아보기 위해 Win32 API를 써서 제작
- 기본적인 Win32 API를 cpp와 header로 기능별로 나눔
- 싱글톤 패턴을 써서 객체를 하나만 할당하고 그것을 가져다 쓰는 방식으로 제작
    - 그로인해 쓸데없는 메모리 사용방지
    - 다른 클래스들이 데이터를 쉽게 공유할 수 있다.
- 하지만 다른 객체들간의 결합도가 높아질수록 Debuging 과정이 어려워짐

- 프레임워크 내부의 조직도는 아래의 PDF파일을 첨부

- 기본적으로 main.cpp에서는 Core.cpp를 실행시키고 화면을 출력하는 WinMain이 포함
- 주요 기능들을 Core에 설계함으로써 Core를 전체적인 설계의 중심으로 만듬
- Message를 무한루프 시켜서 이벤트를 발생 시키는 GetMessage() 부분을 PeekMessage()로 바꿔서 DeadTime을 활용하여 그 안에 게임로직을 넣는다.

Core

- 기본적으로 Input(), Update(), LateUpdate(), Collision(), Render() 부분으로 나누어서 제작
  각각의 함수는 이름 그대로의 기능을 수행하며, 다른 클래스에서도 이 부분을 동일하게 제작
- Message 무한루프 구간에 위의 기능들을 넣고 다른 클래스에서도 동일하게 작성함으로써 Logic()부분에서 실행시킨다.

- Core부분에서 기본적인 Window프로그램의 설정을 다룬다(창의 크기, 타이틀바, 스타일, 아이콘이미지 등등…)
- Main을 제외한 최상위 클래스로써 이 곳에서 다른 기능들의 Manager클래스를 초기화하고 실행한다.

Timer

- 시간 관련 변수 및 사용자의 키입력이나 Rendering의 변화, 시간이 지남에 따라 변해야 하는 기능들을 위해 Timer클래스 제작

- 기능들에 Timer를 적용시키기 위해 원하는 지점과 종료점에서 Timer를 호출해야하는데 이 지점은 CPU의 클럭 타임이 된다. 그래서 따로 LARGE_INTEGER타입으로 Time변수의 QuadPart 값으로 설정한다.

m_fDeltaTime = (tTime.QuadPart - m_tTime.QuadPart) / (float)m_tSecond.QuadPart;

- 이전 프레임(시작점)과 다음 프레임(종료점)의 차이를 시간단위로 나누면 프레임마다 걸리는 시간이 나오기 때문에 일정한 시간의 흐름을 가지게 된다.

*** QueryFrequencyCount

기기가 동작하는 타이밍이 있는데 클럭에 따라 동작하고 안하고 한다.
이 함수는 메인보드의 고해상도 타이머의 주파를 반환한다.
컴퓨터의 메인보드에는 하나의 고해상도 타이머가 존재. 이 타이머는 정확하고 일정한 주파수 (즉, 1초에 각 타이머의 성능에 맞는 진동수)를 갖기 때문에 측정하고자 하는 코드들의 시작과 끝에서 CPU 클럭수를 얻어 그 차로 수행시간을 얻을 수 있다.

1. 처음에 초기화 할 때 LARGE_INTEGER 타입으로 초기화 한다.
   이 때 타이머가 지원되는 경우 0 이 아니고, 지원 안되는 경우 0 으로 초기화
   지금은 지원 안되기 때문에 0으로 초기화
2. 함수가 호출된 시점의 타이머 값이 설정되어 클럭을 받아온다.
3. C 클럭수 = B 클럭수 - A 클럭수
   수행시간 (second 단위) = C클럭수 / 주파수
   수행시간 (millisecond단위) = C클럭수 / (주파수 / 1000)
4. Update함수에서 다시 클럭을 받아온 다음에 초기화한 클럭를 빼서 시간당 프레임을 측정

Ref

- 메모리를 할당 할 때 개발자의 의도와는 다르게 지워지는 메모리를 방지 하기 위해 Reference Counting 기법을 사용
- Ref클래스는 Core, Main을 제외한 모든 객체의 최상위 클래스가 된다.
- 메모리를 참조한다면 Countring + 1, 해제하면 Counting - 1이 돼서 최종적으로 0이 되면 Delete
- Obj클래스의 삭제를 좀더 쉽게 하기 위해 Enable, Life변수를 추가해 객체를 on/off하고, 죽이는 기능 추가

InGameScene
- 게임 실행 화면 부분에 해당되며 Scene에서 Layer를 설정하고 Object가 생성되고 값이 변하며 지워진다.
- InGameScene을 두어서 상위 클래스에 Scene을 상속받게 하여 실질적으로 InGameScene에서 게임 로직이 실행
- Default Layer로 설정해서 기본적인 화면을 설정

Layer
- 화면의 틀
- 서로 겹치거나 펼쳐 놓을 때 레이어를 설정하면 좀 더 쉽게 조작이 가능
- 기본적인 정보 입력 후 Scene을 설정하도록 설계
    - 굳이 이런 이유는 혹시나 다른 Scene에 설정될 수도 있기 때문에 따로 SetScene기능을 제작함
- ZOrder로 각 레이어마다 층을 정해놓고 출력한다.

SceneManager
- 각각의 Scene을 관리하고 다음 Scene과 현재 Scene을 구별하여 생성 가능

Scene
- 하나의 장면으로 로그인, 로딩, 게임 Scene과 같이 장면을 나누는데 필요한 클래스
- 기본적인 정보 입력 후
- Layer를 설정 후 해당 Scene에서 출력하기 전에 LayerSort를 통해 Zorder의 값을 비교하여 값이 큰 순서대로 정렬한다.
- 맨 마지막에는 UI가 들어갈 예정
- Obj클래스에서 작성한 Object들의 원본 보존을 위해 Scene클래스에서 ProtoType을 만들어서 복사하여 사용
- 또한 객체를 가져올 때 마다 읽어오는 속도를 줄이기 위해 사용
- Map형식의 ProtoType은 Obj를 Value값으로 받아 Template으로 전역변수 등록한다.
-

Object
- 하나의 객체, 즉 캐릭터, 총알, 벽 같은 사물을 생성하는데 필요한 기능이며, Ref클래스를 상속받는다.

Obj
- 모든 객체의 모체가 되는 클래스
- List를 이용하여 추가, 삭제, 찾기 기능 수행
- 기본적인 정보(이름, 위치, 사이즈, pivot, Collider, Physics, 중력값, Animatioin)을 가지고 있다.
- 이때 Texutre를 가지고 있는 Object가 존재할 수도 있기 때문에 Texture설정 가능하도록 설계
- 충돌클래스에서는 Object들을 선별하여 충돌여부를 조사했다면 여기서 충돌체를 등록하는 기능을 가지도록 설계
- 객체끼리 서로 충돌 시 실행되는 함수를 미리 만들기는 불가능 하기 때문에 함수포인터를 사용하여 생성가능 하도록 설계

- 모든 Object들은 ProtoType을 가지고 있게 하고 그것을 Clone()하여 쓰도록 설계하기 위해 추상화클래스로 제작하고 하위 클래스에서 Clone()을 가지고 새로 메모리를 할당하도록 설계
- 각 오브젝트들은 Texture를 대비해 Pivot을 가지고 있도록 설계
    - 이 과정에서 Pivot의 기본값을 (0.5, 0.5)로 설정
- 각각 Input, Update, LateUpdate, Collision, Render를 할 때 Ref의 Enable, Life를 조사하여 On되어있는 경우에만 반복하도록 설정
- InGameScene에서는 ClonObj()를 써서 객체 등록

MoveObj
- 움직이는데 있어서 필요한 기능, x,y값 증가, 속도, 낙하여부, 움직임감지, 각도설정 같은 기능 설계
- 여러 상황에 대처하기 위해 OverLoading하여 여러 함수 생성
- 움직이는 객체들의 상위 클래스가 되어서 필요할 때 OverRiding을 통해 MoveObj의 기능들을 조작 가능하도록 설계

StaticObj
- 움직이지 않는 객체들의 모음

Player
- 기본적으로 Texture를 입히고, 사이즈를 정하며 속도, Pivot, 충돌체, 중력, HP같은 기능들을 설정 가능
- 사용자가 직접 조작하는 객체이며 쏘는 기능과, Player의 Hit, HitStay상태를 구별하여 효과에 따라서 가변적으로 동작하도록 설계
- Input클래스에서 키를 재정의 하여 사용

Minion
- 적 캐릭터 클래스
- Player클래스와 기본적인 기능은 동일
- 그러나 간단한 AI구현을 위해 Update부분에 일정거리 움직이면 방향전환의 추가와, FireLimitTime을 추가해 총알을 발사 후 일정거리 지나면 삭제하도록 설계

Bullet
- 총알의 거리와, 한계거리를 설정하고 Player 또는 Minion에 적중 시 삭제되도록 설계
- 기본적으로 총알 사이즈, 속도, pivot, Texture, 충돌체 추가
- SetColorKey()를 통해 총알 Texture 부분에 일정색을 제외시켜서 이미지 향상
-

Stage
- 화면에 출력될 Stage를 출력하는 클래스
- 사이즈, pivot, Texture 설정 가능
- 카메라 스크롤 구현을 위해 Stage클래스의 위치변수와, 카메라 위치변수를 BitBlt함수를 이용하여 카메라의 일정 구역을 설정하고 캐릭터가 움직임에 따라 출력부분이 설정되도록 설계

Resource
- 게임출력에 필요한 Texture들의 모음

ResourceManager

- Texture를 필요할 때 마다 찾기 위해 Map으로 작성하고 이미지당 Texture 객체 하나씩 만들어 지도록 작성
- Load, Find기능을 작성
- init기능에서 매개변수로 HINSTANCE, HDC를 넣고, 클래스변수로 선언하여 Texture부분에서는 클래스변수로 실행시키기 위해 선언
- 객체를 그리는 과정에서 잔상효과가 있기 때문에 BackBuffer를 작성
    - 이 과정에서 작성코드 위치가 가장 중요한데 아래와 같은 과정으로 코드를 배치한다.
    BackBuffer 출력 -> 배경출력 -> 나머지 출력
- 이때 기존 HDC가 아닌 BackBuffer의 메모리DC에 그려주는데 최상위 클래스인 Core클래스에 직접 Render를 한다.

Texture

- 이미지를 Load하는데 있어서 이미지에 대한 Pixel정보를 저장할 곳이 필요한데, HDC타입 변수를 선언하여 메모리 상의 DC를 만든다.
- Win32에서 제공하는 Bitmap을 조정하기 위한 핸들 HBITMAP타입 변수를 사용하여 작성
- DC들은 자체적으로 그리기 도구를 가지고 있는데, Texture를 Load할 때 이미지가 가지고 있는 DC를 호출하여 덮어쓰고 사용하도록 작성
- Texture부분에서 필요없는 부분을 잘라내기 위해 SetColorKey함수를 이용해 특정 색을 지우고 출력하는 기능 작성
- BITMAP의 정보 변수를 작성하여 길이 및 Texture정보들을 이용한다.
- Win32에서 제공하는 BITMAP타입을 이용해 Texture들을 출력
- 자세한 사항은 FrameWork 내부에 기재

Collider

- 충돌체의 최상위 클래스로써 추상화 클래스이며 Clone()을 상속하는 형식으로 하위클래스에서 사용하도록 설계
- 충돌 타입변수를 선언하여 충돌상태, 아닌상태, 충돌중인상태를 구별하고 관리를 위해 List선언
- 상태에 따른 배열을 작성하기 위해 함수포인터로 함수를 제작
- 멤버함수 전용, 전역함수 전용 함수포인터를 템플릿을 사용하여 작성
- 충돌체 추가, 체크, 삭제 기능을 넣고 이를 위해 Obj변수를 선언하여 Set(), Get()를 작성
- 추가 시 각 충돌체가 Rectangle, Sphere, Pixel형식 세가지 방식으로 작성하기 위해 RectToRect, SphereToSphere, RectToSphere, RectToPixel함수를 제작하여 타입별로 다른동작을 하도록 작성

ColliderPixel

- Pixel충돌을 위해 vector로 픽셀을 저장하기 위한 변수 확보와 가로 세로 길이를 선언
- SetPixelInfo()를 통해 각 Pixel별로 Setting, 아래와 같은 요령을 통해 기능을 구현한다
    1. 픽셀충돌체를작성
    2. 스테이지에 픽셀충돌체를준다.
    3. 픽셀충돌체를스테이지가가지고있으면서다른물체들과충돌처리를할수있도록한다.
    4. 다른 도형들과의 충돌처리와 같은 뼈대를 가지고 있어야 한다.
    5. 값을 셋팅할 때 BITMAP타입으로 바꾼다.
- BITMAP파일을 읽어오기 위해 BITMAPFILEHEADER, BITMAPINFOHEADER 변수를 이용하여 각각 파일의 정보와, BITMAP정보를 읽어온다

    BITMAPFILEHEADER {

```
WORD      bfType;
DWORD     bfSize;
WORD      bfReserved1;
WORD      bfReserved2;
DWORD     bfOffBits; }
```

```
BITMAPINFOHEADER {
 DWORD        biSize;
 LONG         biWidth;
 LONG         biHeight;
 WORD         biPlanes;
 WORD         biBitCount;
 DWORD        biCompression;
 DWORD        biSizeImage;
 LONG         biXPelsPerMeter;
 LONG         biYPelsPerMeter;
 DWORD        biClrUsed;
 DWORD        biClrImportant; }
```

- 아래와 같은 정보를 토대로 높이와 길이를 읽고 선언한 vector를 통해 값만큼 할당한다
- 주의할 점은 저장할 때 상하반전이 일어난 상태로 저장이 되기 때문에 출력을 할 때는 다시 원상태로 돌려야 한다
- 그렇게 하기 위해 픽셀 한줄을 저장하고 맨 아랫줄과 Swap
- Collision기능 부분에서 충돌체 타입에 맞는 경우의 수로 나눠서 Collider클래스의 기능값을 반환 받는다.

ColliderRect
- 위치변수와 월드위치변수를 선언하여 충돌체의 위치가 변할시 월드위치변수에서 움직인 만큼 갱신
- Collision기능 부분에서 충돌체 타입에 맞는 경우의 수로 나눠서 Collider클래스의 기능값을 반환 받는다.
-

ColliderSphere
- 위치변수와 월드위치변수를 선언하여 충돌체의 위치가 변할시 월드위치변수에서 움직인 만큼 갱신
- Collision기능 부분에서 충돌체 타입에 맞는 경우의 수로 나눠서 Collider클래스의 기능값을 반환 받는다.
-

ColliderManager
- 충돌 처리 관리자 클래스
- 모든 게임 Object를 조사하여 충돌처리 가능하도록 작성
- List로 작성
- 부위별 충돌을 위해 Object하나에 여러 충돌처리를 넣어서 종류별로 충돌처리 할 수 있도록 작성
- 충돌처리를 위해 Object추가 기능, 충돌부분을 찾는 함수, 처리하는 함수 추가

충돌처리를 어떻게 해야하는가?
오브젝트가 리스트에 연속적으로 들어갔을때, 처음 키의 벨류값과 나머지, 그 다음키의 벨류와나머지
이런식으로 전체적으로 충돌처리를 할수있다. 버블정렬과 비슷하게 돌아간다.

또 다른식으로 한다면
월드를 그리드(영역)로나눈다. 그렇게 하면 영역안의 오브젝트들만 충돌처리 한다.
이런식은 연산이 줄어들기 때문에 추천, 그러나 기본적인 FrameWork 작성을 위해 첫 번째방식으로 처리

- 충돌부분을 찾는 함수에서 iterator로 반복해서 충돌체를 찾는다. 이때 위에 작성한 작성요령을 위해 iter - 1을 하여 이중반복문으로 찾게 한다.
- 충돌부분을 처리하는 함수에서 매개변수를 Src(해당 충돌체), Dest(상대 충돌체)을 둬서 총 세가지 상태로 구분해서 작성한다
    1. 충돌 시작 상태
    2. 충돌 지속 상태
    3, 충돌 종료 상태

  이때, 충돌되고 있는 다른 충돌체들의 목록을 알수 있도록 작성하면 2,3번 상태를 쉽게 판별 가능하다

- 충돌 시작 상태
  충돌목록을 검사하고 이전에 충돌이 없다면 충돌 시작 상태이기 때문에 서로 시작

- 충돌 지속 상태
  충돌 시작 상태 이외의 상태는 지속되고 있는 상태이기 때문에 이를 위한 코드는 미리 작성하기는 불가능 하기에 함수 포인터로 작성

- 충돌 종료 상태
  충돌이 떨어졌을 때 서로 충돌이 안되므로 충돌목록에서 지운다

- 그리고 충돌여부를 bool타입으로 반환받는다.

Animation
- Animation 정보 및 관리 클래스
- 애니메이션에 필요한 구조체 정보 선언

* ANIMATION_TYPE eType;
  ANIMATION_OPTION eOption;
  // 어떤 텍스쳐를 사용하고 있는지 판별변수
  // 라이브러리 내부에서 겹치는 네임스페이스가 있는듯하다. std:: 써서 표준라이브러리의 백터임을 명시
  std::vector<class CTexture*>vecTexture;

  // 모션 도는데 걸리는시간을 LimitTime에주고 Time에서 증가시킨 후 같다면 다른작업
  float fAnimationTime;
  float fAnimationLimitTime;

  // Frame좌표를 주고 이미지에서 어떤 창의위치에서 동작할 지 결정

```
int iFrameX;
int iFrameY;

// 이미지에 행렬이 x*y 이상일때(x > 1, y > 1) 쓰인다
int iFrameMaxX;
int iFrameMaxY;
int iStartX;
int iStartY;
int iLengthX;
int iLengthY;
float fOptionTime;
float fOptionLimitTime;
```

- 애니메이션 동작을 관리를 위해 Map으로 클립 작성
- 추가 기능을 작성하고 나머지는 Obj클래스에서 객체의 애니메이션을 추가한다.
- 애니메이션 동작 방식으로

```
// 디폴트상태
AO_DEFAULT,
// 계속반복
AO_LOOP,
// 한번돌고idle 상태
AO_ONCE_RETURN,
// 한번돌고오브젝트삭제(이펙트같은것을삭제)
AO_ONCE_DESTROY,
// 일정시간동안돌아가다가idle상태
AO_TIME_RETURN,
// 일정시간동안돌아가다가오브젝트삭제
AO_TIME_DESTROY
```

- 애니메이션 동작 방식에는 아틀라스와 프레임 기법이 있는데 프레임 애니메이션 리소스 부족으로 인해 아틀라스 기법으로 구현했지만 프레임 애니메이션 과정도 구현

*** 아틀라스, 프레임
- 아틀라스 : 프레임처럼 동작하나가 이미지가 아닌 전체적인 동작을 한 이미지에 담아 알고르짐을 통해 애니메이션을 동작시킨다.
- 프레임 : 한 동작마다 이미지가 한 개씩 들어가 있다.

- 애니메이션의 전환을 위해 기본 애니메이션을 DefaultClip(), 변한 애니메이션을 Map에서 꺼내 Changeclip()에 넣는다.
- ImageOffset의 추가로 인해 프레임에 이미지가 다 안들어가는 현상이 발생하기 때문에 애니메이션 길이 / 최대 X프레임, 애니메이션 높이 / 최대 Y프레임으로 FrameSize값을 구한다.
- AddClip과정에서 Texture와 함께 삽입 후 정보를 Insert한다,
- 아틀라스 기법 경우와 프레임 기법 경우로 나눠서 Object의 Texture에 Set한다.
- Texture클래스와 동일하게 SetColorKey 사용하여 불필요한 특정 픽셀 제외

*** 애니메이션 함수를 실행하면서 순간적으로 렉이 걸릴 확률이 높고 그동안 DeltaTime이 몇 개의 프레임을 생략할 수 있기 때문에 while반복문으로 현재 애니메이션 동작이 한 프레임당 애니메이션 동작 횟수보다

높으면 빼주고 다시 프레임에 더해주어서 동작의 위화감을 줄인다.

선택한 애니메이션이 끝까지 재생됬다면 시작지점으로 돌아가고 다음줄 애니메이션을 동작시킨다.
가로 프레임의 애니메이션 처리와 세로줄의 애니메이션 처리를 동일하게 처리한다.