**WinApi**

**<API Handle>**

- 운영체제가 무언가를 식별하기 위한 키 값을 핸들이라고 함.

- 핸들은 보통 32bit 정수형 값을 가지고 있음 => 정수형 비교가 가장 빠르기 때문.

- 핸들번호는 운영체제가 정함.

- 핸들은 절대 중복되는 정보를 가지지 않음.

- Windows는 윈도우 창마다 ID값을 부여함. => **HWND**(윈도우 핸들)

- HWND : 핸들 윈도우의 약자. 윈도우의 핸들번호를 저장해서 사용.

하나의 프로그램에서 여러 윈도우를 띄울 수 있으며,

각 윈도우창의 번호(HWND)로 구분하는 것.

- HINSTANCE : 핸들 인스턴스, 프로그램의 인스턴스 식별자. 프로그램 자체의 실체화된 주소임.

**\* HINSTANCE는 프로그램 자체의 핸들이며, HWND는 프로그램안의 윈도우창의 번호임.**

**<HDC>**

- HDC -> Device Context를 만들어서 ID를 반환함.

\* Device Context? – 화면에 그리기 작업(렌더링)을 수행하는데 필요한 데이터의 집합체.

- 연필, 브러쉬 등 그리기 작업을 수행할 때 필요한 집합체임. 윈도우에 무엇을 그릴때는 항상 DC를 가져와서 그려야 함.

- DC의 목적지는 HWND, DC의 펜은 기본 펜(Black), DC의 브러쉬는 기본 브러쉬(White)임.

\* 커널 오브젝트? - Windows 운영체제에서 리소스(Resource : 프로세스, 쓰레드, 파일)들을

관리하기 위한 데이터를 저장하는 메모리 블록

**<Pixel>**

- 우리가 화면으로 볼 수 있는 것은 모니터라는 출력장치를 통해 볼 수 있는 것.

- 우리가 보고 있는 화면은 한 칸 한 칸의 pixel로 이루어져 있고, 기본 단위는 pixel.

- 한 pixel의 값은 R, G, B 각각 1byte(0~255)씩 메모리에 위치함. 1pixel = 3byte.

**<BeginPaint()>**

**-** HDC를 생성할 수 있는 함수.

- EndPaint()와 짝.

- 메시지 처리방식에서만 사용할 수 있는 전용함수라고 생각해야 함.

- 게임루프에 사용하기에는 부적절.

**<윈도우 이벤트>**

- WM\_PAINT : 윈도우에 무효화 영역(invalidate)이 발생한 경우 실행.

\* 무효화 영역 : 윈도우 창을 “최소화” 하면 발생.

\* WM\_PAINT를 강제로 발생하는 함수가 있음.

\* InvalidateRect([윈도우 핸들], nullptr, [이전 그리기를 지울지 여부]);

- WM\_KEYDOWN

WPARAM – 키다운시, 키 입력된 키 정보를 아스키 코드 값으로(대소문자 구분)넘김.

LPARAM – 마우스 클릭 시, 클라이언트 기준으로 마우스 좌표를 반환.

[ x(2byte), y(2byte) ] , LOWORD() - X좌표, HIWORD() – Y좌표.

- WM\_LBUTTONDOWN

\* 이 외에도 다양한 이벤트가 존재함. => MSDN 참조.

**<(실습)사각형 그리기>**

|  |
| --- |
| case WM\_PAINT: //무효화 영역이 발생한 경우.  {  PAINTSTRUCT ps;  HDC hdc = BeginPaint(hWnd, &ps); // DeviceContext(그리기와 관련.)  //1. 펜, 브러쉬 생성  HPEN hRedPen = CreatePen(PS\_DASHDOT, 1, RGB(255, 0, 0));  HBRUSH hRedBrush = (HBRUSH)GetStockObject(LTGRAY\_BRUSH);  //2. 펜, 브러쉬 설정  // SelectObejct()  // 내가 만든 Pen 혹은 Brush를 HDC에 설정하고,  // 이전에 사용하던 Pen, Brush를 void형 포인터로 반환.  HPEN hDefaultPen = (HPEN)SelectObject(hdc, hRedPen);  HBRUSH hDefaultBrush = (HBRUSH)SelectObject(hdc, hRedBrush);    //3. 사각형 그리기  Rectangle(hdc, 10, 10, 110, 110);    //4. 다시 default로 되돌리기.  SelectObject(hdc, hDefaultPen);  SelectObject(hdc, hDefaultBrush);  //5. 만들었던 펜, 브러쉬 삭제.  DeleteObject(hRedPen);  DeleteObject(hRedBrush);      EndPaint(hWnd, &ps);  }  break; |

**<(실습)사각형 이동하기>**

**<(실습)사각형 그리기2>**

**<게임의 프레임>**

- 일반적으로 게임은 Tick을 사용해 초당 60프레임 이상의 업데이트를 수행하는 프로그램임.

- 사람은 60프레임 이상은 되어야 자연스럽다고 느낀다.

**<GetMessage vs PeekMessage>**

**-** GetMessage

메시지 큐에서 메시지 확인 될 때까지 대기상태.

Msg.message == WM\_QUIT인 경우 false를 반환. = > 프로그램 종료

- PeekMessage

메시지 유무와 관계 없이 반환.

메시지 큐에서 메시지를 확인한 경우 true, 아니라면 false반환.

\* 윈도우의 메시지 처리시간은 전체 프로그램 실행 기간 동안 매우 일부에 불과함.

[GetTickCount() 메소드를 통해 확인해볼 수 있음. => GetTickCount()는 1초에 1000번 호출됨.]

\* 따라서, 메시지를 처리하는 시간 외의 시간을 게임 로직에 사용하기 위해

메시지 처리를GetMessage가아닌, PeekMessage로 검사 처리하여 사용함.

**<\_Inout\_, \_in\_>**

**-** 인자가 어떻게 사용되는지에 대한 **주석.**

- \_Inout\_ => 값을 받기도 하고, 접근하여 결과값을 되돌리기도 함.

- \_In => 값을 받기만 함.

**<AdjustWindowRect([LPRECT], [윈도우 스타일(WS\_OVERAPPEDWINDOW], [메뉴창 크기 계산 여부])>**

- AdjustWindowRect 함수는 첫번째 인자로 넣어준 LPRECT를 통해 RECT의 값을 수정함.

- 윈도우 창, 메뉴바 두께도 계산하여 자동으로 rt의 값을 세팅.

**<해상도에 맞게 윈도우 크기 조절.>**

|  |
| --- |
| int CCore::Init(HWND \_hWnd, POINT \_ptResolution)  {  m\_hWnd = \_hWnd;  m\_ptResolution = \_ptResolution;    //해상도에 맞게 윈도우 크기를 조절.  RECT rt = {0, 0, \_ptResolution.x, \_ptResolution.y};  AdjustWindowRect(&rt, WS\_OVERLAPPEDWINDOW, false);  SetWindowPos(m\_hWnd, nullptr, 100, 100, rt.right - rt.left, rt.bottom - rt.top, 0);  return S\_OK;  } |

**<GetDC([윈도우 핸들])>**

- BeginPaint()를 대체하여 DC를 얻는 함수.

- ReleaseDC([윈도우 핸들], [DC])와 짝.

**<비동기 입력 함수>**

**-** GetAsyncKeyState(VK\_LEFT)

- 해당 키가 이전에 눌렸는지, 안 눌렸는지, 상태 값을 비트값으로 리턴하는 함수임.

- 단순하게 현재 눌렸는지 안 눌렸는지 알려면 제일 상위비트를 체크하면 됨(비트연산).

Ex)

|  |
| --- |
| if (GetAsyncKeyState(VK\_LEFT) & 0x8000) {  //TODO  } |

**<Timer>**

- 화면 렌더링 혹은 update 처리 시, 컴퓨터의 성능에 따라 다른 결과가 나오게 됨.

- 이를 해결하기 위해, 시간 동기화를 사용.(현실 시간과 매칭)

- 예를 들어, 1초에 100px의 거리를 이동한다 할 때,

1. 100FPS 컴퓨터에서는 프레임 당 1px을 이동하면 됨.

2. 200FPS 컴퓨터에서는 프레임 당 0.5px을 이동하면 됨.

- 1 프레임당 걸리는 시간을 구해, Speed에 곱해주면 시간 동기화가 가능함.

**[공식]**

|  |
| --- |
| 이동량 \* **(1/frame)** |

\* (1/frame)? : 한 프레임 수행에 걸리는 시간.

**<GetTickCount vs QueryPerformanceCounter>**

- GetTickCount는 초당 1000번 호출.

- **FPS를 구하기엔 부적합**.

- QueryPerformanceCounter는 100만 단위임.

- QueryPerformanceCounter를 사용하려면 카운터 값 차이가 얼마나 나는지

구해야 함.(QueryPerformanceFrequency() => 빈번도)

**[구현]**

**CTimeMgr.h**

|  |
| --- |
| #pragma once  class CTimeMgr  {  SINGLE(CTimeMgr);  private:  LARGE\_INTEGER m\_llCurCount;  LARGE\_INTEGER m\_llFrequency;  LARGE\_INTEGER m\_llPrevCount;  double m\_dDT; // 프레임 간의 시간값.  double m\_Acc; // 1초 체크를 위한 누적 시간  UINT m\_iCallCount; // 함수 호출 횟수 체크  UINT m\_iFPS; //  //FPS  //1 프레임당 시간(Delta Time)    public:  void Init();  void update();  public:  double GetDT() { return m\_dDT; }  double GetfDT() { return (float)m\_dDT; }  }; |

**CTimeMgr.cpp**

|  |
| --- |
| #include "pch.h"  #include "CTimeMgr.h"  #include "CCore.h"  CTimeMgr::CTimeMgr()  :m\_llCurCount{}  , m\_llPrevCount{}  , m\_llFrequency{}  , m\_dDT(0.)  , m\_iCallCount(0)  , m\_Acc(0)  , m\_iFPS(0)  {  }  CTimeMgr::~CTimeMgr()  {  }  void CTimeMgr::Init()  {  //현재 카운트  QueryPerformanceCounter(&m\_llPrevCount);  //초당 카운트 횟수(1초당 몇번 셀 수 있는지)  QueryPerformanceFrequency(&m\_llFrequency);  }  void CTimeMgr::update()  {  QueryPerformanceCounter(&m\_llCurCount);  //1. 이전 프레임의 카운팅과, 현재 프레임 카운팅 값의 차이를 구하고, 초당 카운트 횟수를 나눈다.  m\_dDT = (double)(m\_llCurCount.QuadPart - m\_llPrevCount.QuadPart) / (double)m\_llFrequency.QuadPart;    //2. 이전 카운트 값을 현재 카운트 값으로 갱신한다.  m\_llPrevCount = m\_llCurCount;  ++m\_iCallCount;  m\_Acc += m\_dDT; // DT누적.(흐른 시간을 의미)  //3. 흐른 시간(누적시간)이 1이 넘으면 1초가 경과.  if (m\_Acc >= 1.)  {  m\_iFPS = m\_iCallCount;  m\_Acc = 0;  m\_iCallCount = 0;  wchar\_t szBuffer[255] = {};  swprintf\_s(szBuffer, L"FPS : %d, DT : %f", m\_iFPS, m\_dDT);  SetWindowText(CCore::GetInst()->GetMainHwnd(), szBuffer);  }  } |

**<Double Buffering(이중버퍼링)>**

- 윈도우는 내부적으로 작업영역만큼의 픽셀을 보유. => Bitmap.

- 1. 이중 버퍼링에는 비트맵과 DC가 필요.

CreateCompatibleBitmap()

CreateCompatibleDC()

- 2. 위에서 생성한 비트맵과 DC를 연결해줘야 함.

HBITMAP SelectObject([dc], [bitmap]);

DeleteObject() => 1px 반환되서 그냥 지워버리면 됨.

- 3. 프로그램이 종료될 때, 만든 비트맵과 DC는 DeleteDC(), DeleteObject()로 지워야 함.

DeleteDC()

DeleteObject()

- 이렇게 그린 비트맵을 BitBlt()함수로 main DC에 그려 줌(**복사**).

BitBlt(m\_hDC, 0, 0, m\_ptResolution.x, m\_ptResolution.y,

m\_memDC, 0, 0, SRCCOPY);

- 반복적인 렌더링 처리는 CPU가 부담하기엔 너무 연산이 많다.

따라서 나중에 렌더링 처리는 GPU로 처리하게 됨.

WinApi는 **렌더링 또한** **CPU를 통해 연산**을 함. =>

FPS가 꽤 많이 떨어지나, BitBlt()의 복사비용은 고정임.

DirectX라이브러리를 사용해서 GPU로 컨트롤하게끔 해야함. (GPU에서 하는 그래픽 가속)

**<키매니저>**

**-** 프로그램은 매번(한 프레임) update함수를 호출하고, 렌더링 과정을 거치게 됨.

=> DeltaTime.

- 프로그램은 순차적으로 실행되는데 만약 움직이는 물체 A와 B가 있을 때,

DeltaTime간 A,B 물체가 ‘동시에’ 움직이지 않고 A물체를 먼저 움직이고 B물체를 움직이게 됨.

- 이를 해결하기 위해 키매니저를 통해 **프레임 동기화**를 해주어야 함.

\* 프레임 동기화? : 동일 프레임 안에서 같은 키에 대해 동일한 이벤트를 가져야 함.

- 또한 키매니저는 키 입력 이벤트를 처리함. => 유한상태기계(FSM)

1. tap => 누름.

2. hold => 누르고 있는 상태.

3. away => 뗌.

**<Scene>**

**-** Scene에서는 화면에 그려지는 **모든 오브젝트들을 관리**함.

- 오브젝트를 종류별로 분류하여 관리한다.

**<Scene Manager>**

**-** 한 게임에서 Scene은 여러 개 존재함. (Login, Game, Map1, Map2 … 등등)

- 따라서 이를 관리하는 매니저 클래스가 필요함.

**<Timer>**

**-** 사용자의컴퓨터 성능에 따라, 작업 시간이 다르게 발생.

- 정확히는 CPU 주파수(클럭)에 따라 1초간 Tick이 매우 자주 반복됨.

- 따라서 프로그램은 각기 다른 컴퓨터 성능에 따라 1초 단위로 시간을 동기화해야 함.

-> DeltaTime

**<BOOL WINAPI QueryPerformanceFrequency(\_\_out LARGE\_INTEGER \*lpFrequency)>**  
- 머신이 QueyPerformanceCounter를 지원하면 TRUE, 그렇지 않으면 FALSE를 반환한다.  
- Output 파라미터인 lpFrequency에는 **초당 count** 가 설정된다.

**<BOOL WINAPI  QueryPerformanceCounter(\_\_out LARGE\_INTEGER \*lpPerformanceCounter)>**

- 성공시 TRUE, 실패시 FALSE를 반환한다.

- Output 파라미터인 lpPerformanceCounter에는 **현재 count가 설정**된다.

**[구현]**

|  |
| --- |
| void CTimeMgr::update()  {  QueryPerformanceCounter(&m\_llCurCount);  //1. 이전 프레임의 카운팅과, 현재 프레임 카운팅 값의 차이를 구하고, 초당 카운트 횟수를 나눈다.  //QuadPart는 union 타입.  //union => 구조체와 비슷.  m\_dDT = (double)(m\_llCurCount.QuadPart - m\_llPrevCount.QuadPart) / (double)m\_llFrequency.QuadPart;    //2. 이전 카운트 값을 현재 카운트 값으로 갱신한다.  m\_llPrevCount = m\_llCurCount;  //디버그에서만 동작.  #ifdef \_DEBUG  if (m\_dDT > (1. / 60.))  m\_dDT = (1. / 60.);  #endif // \_DEBUG{}    } |

**<Path Manager>**

**<Resource Manager>**

**<Collider>**

**<Collision Manager>**

**<Event Manager>**

**<Animator>**

**<Animation>**

**<Unity Build>**

* 하나의 파일에 모든 cpp파일을 합쳐서 빌드하는 것.
* .vcproj 파일을 열어보면, itemgroup부분에 cpp파일이 몰려 있는 것을 볼 수 있음.
* VisualStudio에서 기능을 제공하고는 있지만, 아직 실험적 단계임.

**[비주얼 스튜디오 최신버전]**

속성 – 고급 – Unity Build 사용.

**[주의] 헤더파일에 #pragma once 꼭 달아 주자.**

**[테스트 버전 (Deplicated)]**

1. .vcproj 파일에 아래 프로퍼티 그룹을 추가하면 아래와 같은 창이 나타남.

|  |
| --- |
| <PropertyGroup>  <EnableUnitySupport>true</EnableUnitySupport>  </PropertyGroup> |

2. 속성 – 일반 – 다중 프로세서 컴파일 켜기.

텍스트이(가) 표시된 사진

자동 생성된 설명

참고)

<https://woo-dev.tistory.com/316>

**<Tool>**

**<Tile>**

**<MAKEINTRESOURCE()>**

**<Menu Bar>**

**<Modal vs Modal Less>**

**<stdcall>**

**<GetDlgItemInt()>**

**<dynamic\_cast>**

**<assert()>**

**<UI>**

**<상속에서의 복사 생성자>**

**<UI Focus>**

**<Button Function Pointer>**

**<확장자에 대하여>**

**<파일 입출력>**

**<이중 포인터>**

**<Fwrite()>**

**<멤버함수 포인터>**

**<공통 부모>**

**<전방선언>**

**<Alpha Blending>**

* 알파 블렌딩은 알파채널의 값을 이용하여, 텍스쳐 자체의 투명도를 조절하거나, 겹쳐져있는 텍스쳐 끼리 RGB값을 계산하여 섞인 색상을 계산하여 화면에 출력하기 위한 방법임.
* bmp파일에는 RGB값만 존재하고, alpha값이 없음.
* 따라서, Texture에 Alpha 채널을 추가해야 함.

**<렌더링 최적화>**

* Scene\_Start에서 Object(Tile)가 많아지면 프레임 드랍이 심함.
* 카메라에 보여지는 오브젝트만 render하는 식으로 개선해야 함.
* 렌더링 함수를 별도로 구현하여 개선.
* 이렇게 구현하면, 실시간 삭제는 조금 어려움..

|  |
| --- |
| * void CScene::render\_tile(HDC \_dc) * { * const vector<CObject\*>& vecTile = GetGroupObejct(GROUP\_TYPE::TILE); * Vec2 vCamLook = CCamera::GetInst()->GetLookAt(); * Vec2 vResolution = CCore::GetInst()->GetResoultionVec(); * Vec2 vLeftTop = vCamLook - vResolution / 2.f; * int iTileSize = TILE\_SIZE; * int iLTCol = (int)vLeftTop.x / iTileSize; * int iLTRow = (int)vLeftTop.y / iTileSize; * int iLTIdx = (m\_iTileX \* iLTRow) + iLTCol; * int iClientWidth = (int)vResolution.x / TILE\_SIZE; * int iClientHeight = (int)vResolution.y / TILE\_SIZE; * for (int iCurRow = iLTRow; iCurRow < (iLTRow + iClientHeight); ++iCurRow) * { * for (int iCurCol = iLTCol; iCurCol < (iLTCol + iClientWidth); ++iCurCol) * { * int iIdx = (m\_iTileX \* iCurRow) + iCurCol; * vecTile[iIdx]->render(\_dc); * } * } * } |

**<FSM 유한상태기계>**

* Finite State Machine. AI의 일종.
* 프로그래머의 노가다(?)로 만들어지는 인공지능.
* 유한상태기계는 프로그래머의 노가다(?) 정도에 따라 AI가 다양해짐.
* 디자인 패턴의 상태패턴과 유사함.
* 한계는 명확.(컴퓨터가 움직이는 것 같은 느낌은 지울 수 없음. 어색하단 소리.)

**<Factory Pattern>**

**<Sound>**

**<RigidBody>**

**<최대정지 마찰력, 운동 마찰력>**

**<중력>**

**<점프>**

**<애니메이션 파일 저장>**

**<데이터 직렬화(Serialization)>**