

닷지게임 만들기

이번 장에서는 이전의 4개의 장에서 배웠던 내용을 바탕으로 간단한 닷지(Dodge)게임을 만들어보고자 합니다. 물론 게임을 만들 때에는 몇 가지가 더 필요합니다, 해당 내용들 역시 이 장에서 내용을 다루게 될 테니 걱정하지 않으셔도 됩니다. 게임을 어떻게 제작하는지는 전적으로 독자 분들의 자유입니다. 또한 여러분에게 도움이 될 수 있도록 저희가 직접 게임을 제작해서 예제로 추가해 두었습니다. 그러니 제작하다가 막히는 부분이 있으시면 저희가 만든 예제 소스를 참고해 주시기 바랍니다.

물론 저희가 제작한 코드는 다소 비효율적인 부분도 있고, 고작 몇 백 줄 짜리의 간단한 게임일 뿐입니다. 하지만 저희는 여러분이 게임 프로그래밍에 처음 입문하고, 이번에 제작하는 것이 여러분의 첫 게임이라고 생각하며 제작했었습니다. 따라서 여러분이 더 넣고 싶은 기능이 있으시다면 마음껏 추가해 보시기 바랍니다. 또한 게임의 리소스가 필요하실 수도 있으신데, 오디오 파일을 제외한 이미지 파일은 저희가 직접 제작한 리소스들이기 때문에 마음대로 사용하셔도 됩니다.

게임 기획: 닷찌!

이 게임은 화살표 키로 캐릭터를 조종해서 사방에서 날라오는 적을 피하는 게임입니다. 만약 적들과 부딪치게 된다면 게임이 종료됩니다. 죽지 않고 버틴 시간이 화면에 표시되고, 최대한 오랫동안 버티는 것이 이 게임의 목표입니다. 얼마나 오래 버티느냐에 따라서 결과 화면이 달라집니다.

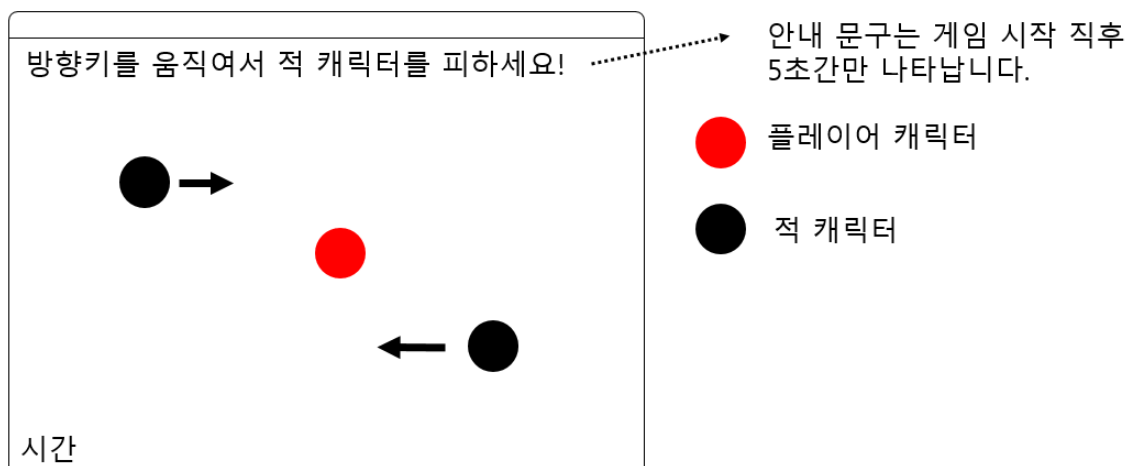
게임 화면

메인 화면(클라이언트 영역 가로 800, 세로 600)



메인 화면에서는 단순히 게임의 제목과 이미지들을 표시합니다. 또한 (1)의 “스페이스 바를 누르면 시작합니다.” 라는 내용은 1초마다 깜빡이며 사라지고 나타나고를 반복합니다. 스페이스 바를 누르면 곧바로 게임 화면으로 넘어가고 게임을 시작하게 됩니다.

게임 화면(크기는 메인화면과 같음)



게임 화면에서는 적 캐릭터와 플레이어 캐릭터가 나타나는데, 플레이어 캐릭터는 화살표 키로 사용자가 조작할 수 있고, 적 캐릭터는 지속적으로 사방에서 무작위로 생성됩니다. 안내 문구는 게임 시작 후 5초간만 나타납니다.

게임 중 ESC를 누르면 게임이 일시정지 됩니다. 이 때 다시 ESC를 누르면 일시정지가 해제됩니다. 플레이어가 적과 부딪치면 게임이 끝나게 되고, 결과 화면이 나타납니다. 결과 화면은 플레이어가 얼마나 오랫동안 죽지 않았냐에 따라 달라지게 됩니다. 결과 화면에서 Enter키를 누르면 메인 화면으로 되돌아가게 됩니다.

어떻게 만들지?

이후의 내용들은 여러분이 게임을 만들면서 도움이 될 내용들에 대해 적어 보았습니다. 만약 게임을 개발하다가 막히는 부분이 생겼을 때 참고하셔도 좋고, 먼저 보신 뒤 게임을 만드셔도 좋습니다.

FPS 제어

4장에서 봤던 대로, GetMessage()함수 대신 PeekMessage()함수를 이용하면 실시간 처리를 할 수 있었습니다. 그런데, 실시간 처리를 하면서 생겼던 문제점은, 너무나도 빨랐단 것이죠. 1초에 수십만 번씩 처리를 하게 된다면 CPU를 풀로 사용하여 다른 작업에 문제가 생기고 비효율적입니다. 예를 들어 1초동안 100px을 움직이는 장면을 100번에 나누어 보여준다면 사람들은 어색함을 느끼지 않을 것입니다, 그런데 굳이 이 장면을 10000번에 나누어서 보여주어야 할까요? 사람들은 실제로 이런 것들을 구별조차 하지 못합니다. 이 나누어서 보여주는 한 장면을 프레임(Frame)이라고 하며, 1초당 나타나는 프레임의 단위를 FPS(Frame per Second)라고 합니다. 대부분의 게임들은 60fps를 사용합니다, 즉 1초에 60번씩 화면에 그리게 되는 것이지요, 그러면 우리가 한번 FPS를 제어해서 60fps를 만들어 봅시다.

방법은 간단합니다, 마지막으로 처리를 했을 때의 시간을 저장한 뒤, 현재 시간을 얻어온 다음, 둘을 비교하여 일정 시간이 지났을 경우에 처리를 하면 됩니다. 현재 시간을 얻어오는 함수는 timeGetTime() 함수를 사용할 것입니다, 이 함수는 winmm.lib을 링크해야 사용할 수 있습니다. 이 함수는 컴퓨터를 키고 윈도우가 부팅되고 난 뒤에 지난 시간을 반환해 줍니다. 아래의 예제 코드는 이 함수를 활용해서 FPS를 제어합니다.

```
변수입니다.
DWORD lastTime = timeGetTime();

//
// 메시지 루프를 돈다.
MSG msg = {0};
while(msg.message != WM_QUIT)
{
```

```

// 처리할 메시지가 있으면 처리하고
if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
// 그렇지 않다면 FPS를 제어해서 갱신과 그리기를 한다
else
{
    // 현재 시간을 얻어온 다음
    DWORD currTime = timeGetTime();

    // 마지막으로 갱신한 시간과의 차이(delta)를 계산한다.
    DWORD delta = currTime - lastTime;

    // 차이가 16ms이상이면 갱신합니다.
    if(delta >= 16)
    {
        // (중요)마지막 갱신 시간을 저장해야 합니다.
        lastTime = currTime;

        OnUpdate(delta / 1000.0f);
        OnDraw();
    }
    // 그렇지 않을 경우 남은 시간은 Sleep()함수로 멈춰 줍니다.
    else
        Sleep(16 - delta);
}
}

```

예제 1. FPS제어 - WinMain 함수의 메시지 루프 부분

이제 OnUpdate()함수에서 인자로 전해지는 delta값에 대해 알아보시다, 결과적으로 마지막 갱신 시간과의 시차(delta)가 초 단위의 float형 값으로 전달되게 됩니다. 이를 OnUpdate() 함수 내부에서 아래의 예처럼 곱셈을 사용하여 활용하면, 원하는 속력만큼 위치를 변화시킬 수 있습니다.

```

float x = 400, y = 300;

void OnUpdate(float delta) {
    float speed = 500.0f;

    if(GetAsyncKeyState(VK_LEFT) & 0x8000)
        x -= speed * delta;
    if(GetAsyncKeyState(VK_RIGHT) & 0x8000)
        x += speed * delta;
}

```

```

    if(GetAsyncKeyState(VK_UP) & 0x8000)
        y -= speed * delta;
    if(GetAsyncKeyState(VK_DOWN) & 0x8000)
        y += speed * delta;
}

```


예제 2. FPS제어 - OnUpdate() 함수

위 예제에서, X, Y값은 초당 최대 500이상 바뀔 수 없습니다.


리스트 사용하기

게임에서는 수많은 데이터들을 관리해야 합니다. 여러 개의 데이터를 관리할 때는 배열을 사용합니다, 그런데 만약 그 배열의 크기를 알 수 없다면? 그래도 동적 할당을 사용하면 됩니다, 하지만 만약 그 크기가 자유자재로 늘어나고, 값이 중간에 계속해서 추가되었다가 사라진다면 어떻게 될까요? 물론 직접 구현할 수 있겠지만 매우 귀찮고 힘든 일일 것입니다. 이에 C++은 STL(Standard Template Library)를 제공하여 이런 귀찮은 일들을 쉽게 해결할 수 있도록 해줍니다. 앞서 말한 것처럼 데이터를 담는 것을 컨테이너(Container)라고 합니다. 그 중에서 데이터를 일반적인 배열처럼 일렬로 순서가 있도록 보관하는 컨테이너를 리스트(List)라고 합니다. 리스트도 종류가 배열 리스트와 연결 리스트로 나뉘고 둘 다 장단점이 존재합니다, 그러나 여기서는 배열 리스트인 `std::vector` 클래스에 대해서만 알아보려고 합니다. 이 클래스는 앞서 말한 문제점들을 모두 해결해 줄 수 있는 템플릿 클래스이며, 헤더 파일 `<vector>`에 포함되어 있습니다. 그림으로 `vector`를 표현하자면 아래와 같습니다.

index	1	2	3	4	5	6	
value	"Hello"	"ABC"	"asdf"	"헤헬"	"무닝그"	"미누그"	



begin()



end()

그림 1. vector의 구조

Vector는 네임스페이스 `std`안에 들어있는 템플릿 클래스입니다. 따라서 어떤 타입이든 저장할 수 있습니다.

```

#include <vector>
std::vector<int> v1; // int타입 벡터
class A { ... };
std::vector<A> v2; // A타입 벡터

```

예제 3. vector 객체 선언하기

push_back 메서드를 통해서 값을 추가할 수 있습니다. 추가한 값을 원소(Element)라고 합니다.

```
v1.push_back(5); // 5 추가
v1.push_back(15); // 15 추가
A a = ... ;
v2.push_back(a);
```

예제 4. vector 에 값 추가하기

벡터가 가지고 있는 원소의 개수는 size()함수로 알아낼 수 있습니다. 또한 벡터의 i번째 원소를 얻어내고 싶다면, 중괄호[]를 사용하여 배열처럼 얻어낼 수 있습니다.

```
for(int i = 0; i < v1.size(); ++i)
    printf(" v[%d] = %d\n" , i, v1[i]);
```

예제 5. vector 의 원소 순회

STL에는 다양한 종류의 컨테이너들이 존재합니다. 하지만 그 컨테이너의 특성으로 인해서, 모두가 []연산자를 이용해서 간단하게 값을 얻어올 수는 없었습니다. 따라서 STL의 개발자들은 모든 컨테이너들에 같은 인터페이스를 적용하여 똑같은 방식으로 사용할 수 있도록 했습니다. 그것이 낳은 결과가 바로 반복자(Iterator)입니다. 반복자는 컨테이너의 특정 원소를 가리키는 객체입니다. 또한 그 객체와 연결되어 있는 이전 혹은 이후의 원소들에 대한 정보 역시 갖고 있습니다. 따라서 컨테이너의 중간의 값을 삭제하거나 추가할 때, 혹은 특정 원소의 값을 얻어오고 싶을 때 바로 반복자를 사용합니다.

STL의 모든 컨테이너는, 기본적으로 begin()과 end()메서드를 갖고 있습니다. 이 두 메서드는 각각 컨테이너의 맨 처음 원소와, 맨 마지막 원소의 다음 원소를 가리키고 있습니다. 즉 end()메서드가 반환하는 반복자는 컨테이너의 완전한 끝을 의미합니다. 따라서 맨 마지막에 값을 추가하거나, 특정 위치를 가리키는 반복자를 반환해야 하는데, 특정 위치에 원소가 없을 경우, 그리고 반복자를 이용해서 모든 값을 탐색하는데 더 이상 탐색할 값이 없을 경우 이 end() 메서드의 반복자가 반환되게 됩니다. 이러한 특성을 이용하면, 아래처럼 반복자를 이용해서 vector의 모든 원소를 순회할 수 있습니다. 아래 예제의 결과는 예제 5와 같습니다.

```
typedef std::vector<int>::iterator Iterator;
iterator it = v1.begin(); // 첫 반복자를 반환합니다.

// 원소의 끝인지 비교합니다.
while(it != v1.end()) {
    int &a = *it; // *연산자로 원소의 참조자를 얻을 수 있습니다.
    a += 5; // 따라서 이렇게 값을 변경할 수도 있습니다.
```

```

    printf(" v[%d] = %d\n" , i, a);
    it ++; // 다음 번 원소를 가리키게 합니다 .
}

```

예제 6. 반복자를 이용한 vector 원소 순회

반복자의 이동은 단순히 ++이나 --연산자 뿐만 아니라 +=, -=, +, - 등의 연산자도 사용 가능합니다. 그러나 여러분의 컴파일러가 이를 지원해주어야 합니다. 현재 제가 사용중인 Visual Studio 2012의 STL의 경우에는 앞서 말씀 드린 모든 연산자가 사용 가능합니다.

반복자를 이용하면 값을 추가하거나 제거하는 것도 가능합니다. 값의 추가는 insert() 메서드를 사용하고, 값의 제거는 erase() 메서드를 사용합니다.

```

typedef vector<int>::iterator Iterator;

vector<int> v;
int values[] = {
    1, 2, 3, 4, 5,
    6, 7, 8, 9, 10,
};

// 벡터의 처음에 배열의 모든 값을 추가합니다 .
v.insert(v.begin(), values, values + 10);

// 벡터의 특정 위치에 값 추가
v.insert(v.begin() + 5, 55);
v.insert(v.end(), 111);
v.insert(v.end(), 11122);

v.pop_back();    // 마지막 값 제거하기
v.erase(v.begin() + 3); // 4번째 값 제거

// 특정 범위의 값 제거 (3번째 ~ 7번째 )
v.erase(v.begin() + 2, v.begin() + 7);

Iterator it = v.begin();
int i = 0;
while(it != v.end())
{
    int a = *it;
    printf("[%d] = %d \n", ++i, a);

    /*

```

순회 중 값을 제거할 때에는 반환되는 반복자를 저장해 두어야 합니다.

왜냐하면 값을 제거할 경우 그 반복자는 잘못된 원소를 가리키기 때문에

제거한 값의 다음 원소를 가리키는 반복자를 저장해야 합니다.

따라서 제거할 경우에는 `it++` 도 하지 않아야 합니다.

```

*/

// 홀수이면 제거합니다.
if(a % 2)
    it = v.erase(it);
else
    ++it;
}

```

예제 7. vector 의 원소 추가/삭제

그냥 vector를 깔끔하게 비워버리고 싶다면, `clear()` 메서드를 사용하시면 됩니다.

```
v1.clear();
```

예제 8. vector 의 모든 원소 제거

충돌 처리

게임 내에서, 적과 플레이어가 충돌할 경우 게임이 종료됩니다. 그런데 적과 플레이어가 부딪쳤다는 것을 어떻게 알 수 있을까요? 그것은 두 객체의 위치와 크기를 통해서 비교하면 손쉽게 알 수 있습니다. 대표적으로 AABB 충돌처리 알고리즘이 사용됩니다. AABB란 Axis-Aligned Bounding Box의 약자로, 축에 정렬된 경계 상자입니다. AABB는 물체가 회전하면 경계 상자가 바뀌게 됩니다. 물체가 회전해도 경계가 그대로인 축에 정렬되지 않은 경계 상자는 OBB(Oriented Bounding Box)라고 합니다.

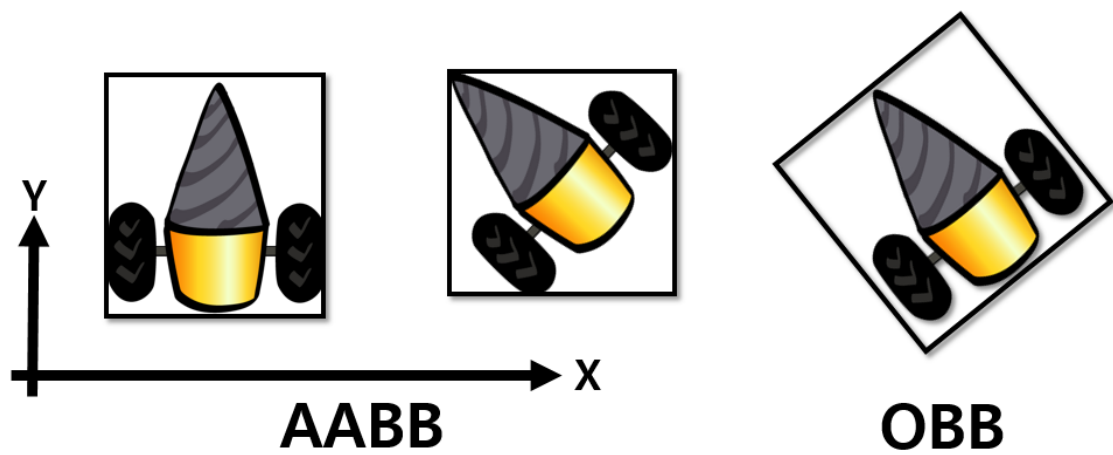


그림 2. AABB와 OBB

Direct2D에서는 이러한 AABB와 똑같은 구조를 가진 D2D1_RECT_F 구조체를 제공합니다. 이 구조체는 left, top, right, bottom 4가지 float타입 멤버를 갖고 있습니다. 각각의 멤버는 스크린 좌표계에서 기하학적으로 아래와 같습니다.

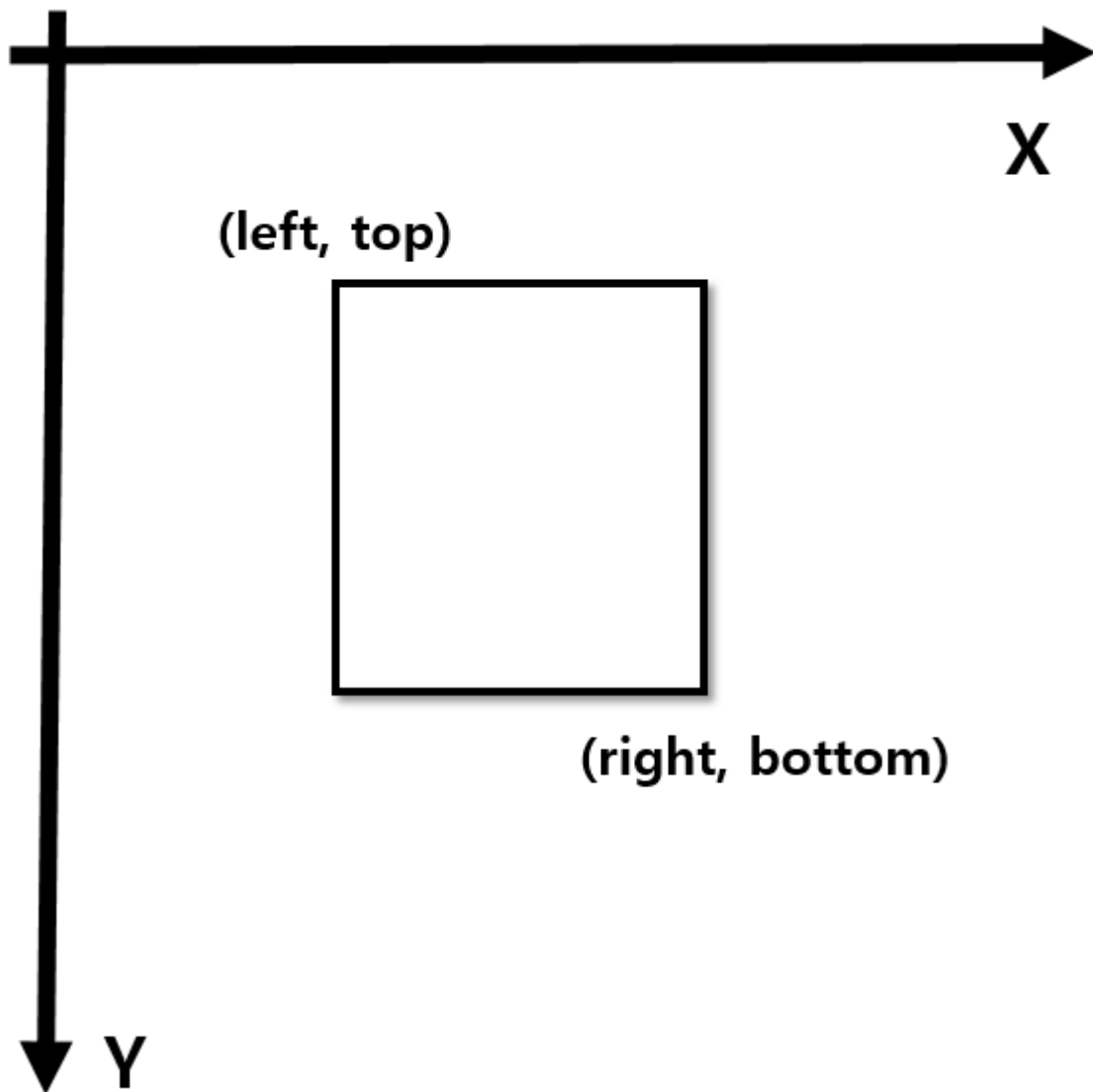


그림 3. D2D1_RECT_F 구조체가 나타내는 AABB

이러한 두 AABB가 충돌했는지 알아내는 함수는 아래와 같습니다.

```
inline bool IsIntersect(  
    const D2D1_RECT_F& a,  
    const D2D1_RECT_F& b )  
{  
    return a.right > b.left &&  
           b.right > a.left &&  
           a.bottom > b.top &&  
           b.bottom > a.top;  
}
```

예제 9. AABB의 충돌 여부를 알아내는 함수

만약 두 AABB가 충돌했다면, 위 함수의 조건을 만족해야 합니다. 하나는, 서로의 우측 면이 상대방의 왼쪽 면보다 우측에 존재해야 합니다. 다른 하나는 서로의 아랫면이 상대방의 윗면보다 아래에 있어야 합니다. 이 함수에 대해서 이해하셨다면, 다음에는 한번 경계상자 A가 경계상자 B를 포함(Contain)하는지 알아내는 함수를 만들어 보시기 바랍니다. 충돌 알고리즘은 세세하게 따지고 들어가면 굉장히 복잡합니다. AABB충돌처리는 굉장히 간단하고 빠르기 때문에 대부분의 게임들이 사용합니다. 좀 더 복잡한 알고리즘으로는 SweptAABB, SAT, GJK등이 존재합니다.

게임 설계

게임 설계란 것은 간단하게 말하면 게임을 어떻게 짤 지를 기획하는 것입니다. 이전에 했던 화면 구성과 같은 것도 게임 설계의 일종입니다. 하지만 여기에서는 어떻게 프로그래밍을 할 지에 대해서 기획해 보도록 하겠습니다. 일단은 게임 화면이나 메인 화면이나에 따른 작업 목록(Task List)를 만들어 보았습니다. 또한 프로그램을 시작할 때 해야 할 목록도 만들어 보았습니다.

프로그램을 시작할 때

1. 윈도우 창을 생성한다.
2. Direct2D 객체들을 생성한다(드로잉용).
3. IrrKlang 객체를 생성한다(사운드 재생용).
4. 비트맵들을 로딩한다.
5. 글꼴을 생성한다.
6. 몇몇 사운드를 미리 로딩한다.

메인 화면에서

갱신해야 할 부분

1. Space바가 눌렸는지 확인하고 눌렀으면 게임을 시작.
2. "스페이스바를 누르면 시작합니다." 문자가 1초마다 깜빡이게 해야 함.

그려야 할 부분

1. 메인 화면 로고 비트맵
2. "스페이스바를 누르면 시작합니다." 문자열

게임 화면에서

갱신해야 할 부분

-
1. 게임이 끝났으면 Enter가 눌렸는지 확인하고 메인 화면으로 이동
 2. ESC키가 눌렸는지 확인하고 일시정지 상태를 활성화/비활성화 한다.
일시정지는 약 0.5초마다 사용할 수 있게 한다. 왜냐하면 잠깐 눌렀음에도 실시간 처리를 하다면 여러 번 입력 받을 수 있기 때문이다.
그리고 일시정지 상태라면 이후의 작업들은 수행해서는 안 된다.
 3. 일정 시간마다 적을 무작위 위치에서 생성해서 적 목록에 추가한다.
 4. 화살표 키 입력을 확인하고 그에 따라 플레이어를 이동시킨다.
 5. 모든 적들에 대해서(적 목록에 대한 순회)
 - A. 적들을 갱신한다.
 - B. 적들이 화면 밖으로 벗어났다면 제거한다.
 - C. 플레이어랑 부딪쳤으면 게임을 종료한다.

그려야 할 부분

1. 배경 비트맵을 그린다.
2. 적들의 캐릭터를 모두 그린다.
3. 플레이어의 캐릭터를 그린다.
4. 일시정지 상태이면 일시정지 화면을 그린다.
5. 그렇지 않고 게임이 종료된 상태이면 결과 화면을 표시한다.

이렇게 작업 목록이 완성되면, 작업에 필요한 변수들을 알 수 있다. 예를 들면 게임 화면에서 적 목록은 Enemy라는 적 캐릭터에 대한 정보를 나타내는 클래스를 만든 뒤, `vector<Enemy>` 타입 객체를 만들면 될 것입니다. 또한 일시정지나 게임 종료와 같은 상태 값은 열거형이나 bool형 변수를 사용하면 해결할 수 있을 것입니다. 또한 게임이 메인 화면인지 게임 화면인지도 열거형 값으로 제어할 수 있을 것입니다.