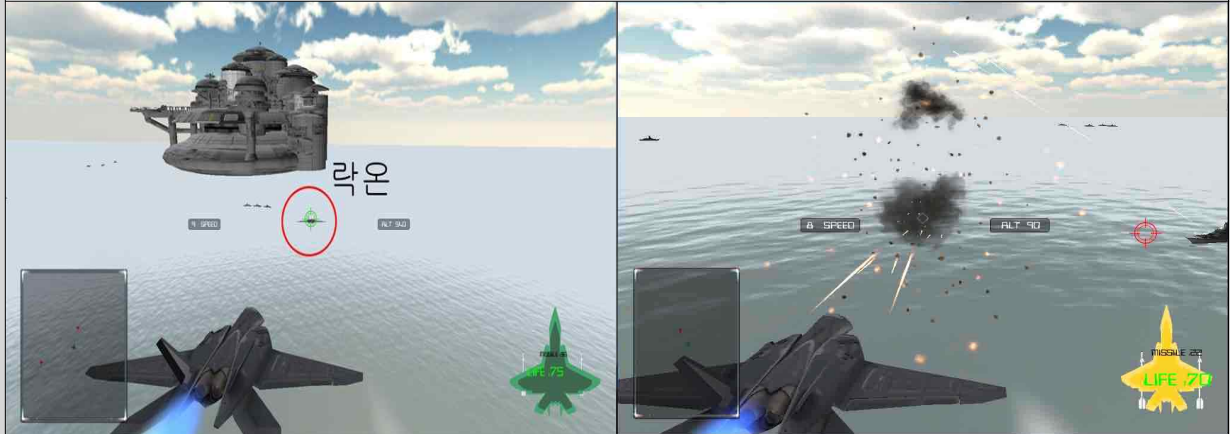


제목	AirTemplate_TeamProject (G-STAR 출품 예정작)
제작 기간	14.5.10 ~ 14.6.23
팀 원 및 담당 업무	<ul style="list-style-type: none"> <li>● 하승우(Main Programmer) -Targeting System, 적 AI System, 해상 유닛 제어 System, 최종 방어 기지 System, Player 제어 System 구축</li> <li>● 염예찬(Main Programmer) -GUI System, 지상 유닛 제어 System, 미사일 System, 레이더 System, Stage 및 Scene System 구축</li> <li>● 김정민(Graphic Designer) -전체 Design Source 제공</li> </ul>
기본 조작법	<ol style="list-style-type: none"> <li>1. 방향키 : W A S D (Q, E 초보자용 좌, 우 방향조절)</li> <li>2. 유도미사일발사 : (LockON시) SpaceBar</li> <li>3. 총알발사 : MouseLeft</li> <li>4. Speed UP &amp; Down: J(up), K(down)</li> </ol>
특징	<ul style="list-style-type: none"> <li>- Unity를 사용한 3D 비행슈팅 게임</li> <li>- 비행슈팅 게임 매니아를 위한 조작법</li> <li>- 지상 / 공중 AI로 다양한 AI 구현</li> <li>- 요새의 핵심 방어기지를 파괴하여 요새를 점령하는 미션 클리어 방식</li> <li>- 미사일(유도)과 기관총의 2가지 공격 방식</li> <li>- C# Script를 이용한 System 구현</li> </ul>
플랫폼	PC / Unity3D
만들게 된 이유	<p>Unity 공부를 하면서 2D 슈팅게임에 관한 예제를 보았다. Unity의 특성상 2D게임을 카메라 시점 변환을 통하여 3D 게임으로 만들면 평소에 재밌게 즐기던 HWAX 나 AceCombat을 만들 수 있을 것 같아서 시작하게 되었다.</p> <p>롤 모델 역시 HWAX 와 AceCombat을 참고하여 만들었다.</p>
게임을 만들면서	<p>이 게임을 제작하면서 가장 걱정된 부분은 그래픽 부분이었다.</p> <p>하지만 Unity3D특징인 Asset Store를 통해 많은 그래픽 Source를 얻을 수 있었고, Asset Store에 없는 Source는 3D_max Source를 직접 축 변환을 시켜 Unity로 가져와 사용 하였다.</p> <p>그리고 두 번째 문제로 3D게임을 처음 제작하는데 있어서 각도 문제가 정말 힘들었다. 덕분에 게임을 제작하면서 Quaternion각도와 오일러각도의 차이도 알 수 있었고, 3D 좌표계에 대한 개념도 알게 되었다.</p>
재미요소	<ol style="list-style-type: none"> <li>1. 3D 비행 슈팅게임의 3차원 적인 움직임</li> </ol> <ul style="list-style-type: none"> <li>- 다소 초보자에겐 어려울 수도 있지만, 실제 자신이 비행을 하는듯한 느낌을 느낄 수 있다.</li> <li>- (Ex. 적이 발사한 미사일 회피)</li> </ul> 

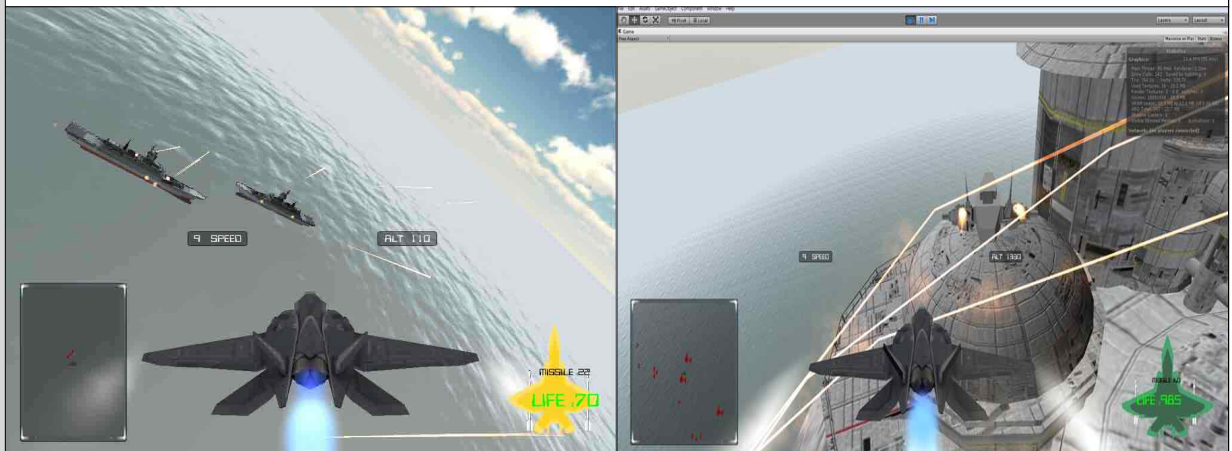
## 2. 2가지 공격 방식(유도탄, 기관총)으로 선택적 공격이 가능

-유도탄은 LockOn 시스템으로 구현을 하여 미사일 적중이 어려운 3D 슈팅게임의 단점을 보완하기 위하여 적과의 일정 거리와 각도를 계산하여 자동적으로 LockOn이 되고 유도탄이 발사 되게 하였다. 또한, 유도탄만으론 공격의 딜레이와 단조로움을 없애기 위하여 LockOn시 적중률이 높아지는 기관총 공격방법도 구현 하였다.



## 3. AI System(적들의 인공지능)

-비행기는 요새 주변을 선회하다가 Player를 쫓아온다거나 요새 방어 Turret의 Player 자동인식 기능 등으로 적의 공격에 대한 Player의 컨트롤을 요함



## 4. 임팩트 효과

-Player의 Hp가 일정 이하로 감소하면 기체에서 연기가 발생하거나, 적 방어기지 타격 시 폭발 후 해당 위치에 연기가 발생하여 더욱 실감 나는 게임을 표현



## 5. 다양한 UI System

-Player 전투기 선택, 레이더, 고도, 속도표시, 미션성공, 게임오버 등 기존 3D비행 슈팅의 UI를 구현



제작 시  
어려웠던 점

### 1. AI System과 각도 System

-AI 같은 경우 어색하지 않고 능동적인 움직임을 표현을 하는데 있어서 어려움이 있었지만, 이 부분은 Eunm과 Switch문을 통하여 동작 컨트롤을 할 수 있었다. 하지만, 동작 컨트롤과 동시에 자연스러운 움직임을 표현하기 위하여 각도 부분을 자동적으로 변화 되게 하는데 있어서 많은 어려움을 겪었다.

### 2. 적 Targeting System

-조작이 어려운 3D슈팅 게임에서 적을 정확히 타격하기 위해선 유도미사일 콘텐츠가 꼭 필요 하였다.

그래서 Targeting 하기 위해서 일정 범위의 적들을 List에 추가시켜 List를 검색하여 Targeting 주려고 하였으나, add와 Search부분에서 오류가 너무 많아 많은 어려움을 겪었다.

그래서 Stage 시작할 때 모든 적들을 List에 넣어 가까운 적을 List에서 찾아 LockOn되도록 하였다.

## 개설점

- 적 전투기의 회전 움직임을 Player와 동일하게 기체를 회전 하여 움직임
- Tutorial 외 다른 Stage구성
- 각종 버그 수정

## 핵심 스크립트 및 코드

### 코드 구조 설명

Script	Method	Method 설명
CsEnumAir	TraceMove()	<pre> void TraceMove() {     // 플레이어와의 전방에 대한 각도 계산     angle = Vector3.Angle(target.transform.position-transform.position, transform.forward);     keyRot = angle/180.0f; // 전방 180도      float amtRot = rotSpeed * Time.smoothDeltaTime; // 회전 속도     Vector3 dir = target.transform.position - transform.position; // 플레이어에 대한 바라보는 방향 계산     float direction = Vector3.Dot (dir,transform.forward);      Vector3 pos = target.position - transform.position; // 플레이어와의 거리 계산     Quaternion ang = Quaternion.LookRotation(pos); // 거리에 대한 쿼터니언 각도 계산      // 거리가 200 이내 이고, 발사 물체가 없고, 락온 상태일 경우     if(dist &lt; 200 &amp;&amp; canfire &amp;&amp; transform.GetComponent&lt;TargetDetector&gt;().TargetLocked)     {         StartCoroutine("EnemyAttact"); //공격 시작     }      // 플레이어가 전방 -&gt; 바로 추적 , 플레이어가 후방 -&gt; 거리가 80이상 벌어졌을 경우 추적     if(direction &gt; 0.5f    direction &lt; 0.5f &amp;&amp; dist&gt;80)     {         // 플레이어와의 각도를 보관하여 부드럽게 추적 가능         transform.rotation = Quaternion.Slerp(transform.rotation, ang, amtRot );     }      // 플레이어가 전방에 위치하고 거리가 20이하이면 수직 상승하여 회피     if(direction &gt; 0.5f &amp;&amp; dist &lt; 20)     {         state = STATE.ShootUp;     }      // 플레이어의 후방 일 경우 재추적 준비     if(direction &lt; 0.5f)     {         state = STATE.ReturnTrace;     }      // 바다와의 거리가 3이하일 경우 수직 상승     if(transform.position.y &lt; 3)     {         state = STATE.ShootUp;     } } </pre> <p>Player와의 거리와 공격 가능 여부를 판단하는 함수</p> <p>*Vector3.Dot는 객체와 객체 사이의 거리 값으로 내적 값을 구한다.          구한 내적의 값이 -1일 경우 객체의 후방이고, +1일 경우 전방이다.</p> <p>-플레이어가 전방 -&gt; 바로 추적 , 플레이어가 후방 -&gt;          거리가 80이상 벌어졌을 경우 추적</p> <p>-플레이어가 전방에 위치하고 거리가 20이하이면 수직 상승하여 회피</p> <p>-플레이어의 후방 일 경우 재 추적 준비</p> <p>-바다와의 거리가 3이하일 경우 수직 상승</p>



	DefaultMove()	<pre> void DefaultMove(){     float movedist ;     bool returnCheck;      //페트롤 지역과 거리체크     float distPetrol = Vector3.Distance (transform.position, Apoint.transform.position);     Vector3 dir = Apoint.transform.position - transform.position;     float direction = Vector3.Dot (dir,transform.forward);     float amtRot = rotSpeed * Time.smoothDeltaTime;    //회전속도      Vector3 pos = Apoint.position - transform.position;     Quaternion ang = Quaternion.LookRotation(pos);     // transform.rotation = Quaternion.Slerp(transform.rotation, ang, amtRot);      if(direction &lt; 0.5f)    //페트롤 지역 선회 조건         returnCheck = true;     else         returnCheck = false;      //페트롤 지역 선회     if(returnCheck)     {         transform.Translate(Vector3.forward * amtMove);         if(distPetrol &gt; 60)         {             transform.rotation = Quaternion.Slerp(transform.rotation, ang, amtRot);         }     }     else if(!returnCheck)     {         transform.Translate(Vector3.forward * amtMove);         if(distPetrol &gt; 60)         {             transform.rotation = Quaternion.Slerp(transform.rotation, ang, amtRot);         }     }      if(target)     {         movedist = Vector3.Distance (transform.position, target.transform.position);         if(movedist &lt; 300)         {             state = STATE.Trace;         }     } } </pre> <p>특정 Patrol 지역을 빈 Object로 설정 하여 빈 Object와의 거리에 대한 Quaternion각도를 이용하여 해당 위치 방향으로 이동하고 Patrol지역을 일정 거리 이상 벗어나게 되어 AI 후방에 Patrol 지역이 위치하게 되면 다시 거리에 대한 Quaternion각도를 계산하고 다시 Patrol지역으로 이동하게 하여 Patrol 지역과의 일정 거리를 유지하면서 정찰한다.</p>
	ReturnMove() ->ReturnTrace()	<pre> void ReturnMove() {     StartCoroutine("ReturnTrace");    // 재추적 함수 실행 }  IEnumerator ReturnTrace(){     transform.Translate(Vector3.forward * amtMove);    // 실행 직후 전진     if(transform.position.y &lt; 3)     {         state = STATE.ShootUp;     }     yield return new WaitForSeconds (1);    // 1초 후 다시 플레이어 추적     state = STATE.Trace; } </pre> <p>Player를 추적하다 Player의 지나쳤을 경우 1초 뒤에 다시 Trace 모드로 변하여 Player를 다시 추적 한다.</p>

		<pre> void ShootUpMove(){     transform.Rotate(-60 * Time.smoothDeltaTime,0,0); // 기체가 하늘방향으로 회전     if(transform.position.y &gt; 5) // 바다와의 거리가 5이상 멀어지면 다시 기본 움직임 상태     {         state = STATE.Default;     } } </pre> <p>Player와의 거리가 너무 가깝거나, Player를 추적 중 바다와 너무 가까울 경우 수직 상승하여 충돌을 방지한다.</p>
CsIdentification	AddPlayerTargets()	<pre> public void AddPlayerTargets() {     GameObject[] Enemy = GameObject.FindGameObjectsWithTag ("Enemy");      foreach(GameObject enemy in Enemy)     {         PlayerTargets.Add(enemy.transform); //모든 적 ArrayList에 추가     } } </pre> <p>Tag가 Enemy인 객체 전부의 transform 값을 enemy ArrayList에 추가시킨다.</p>
TargetDetector	TargetDetector()	<pre> if(Input.GetKeyDown(KeyCode.Backspace) &amp;&amp; TargetLocked) {     transform.GetComponent&lt;TargetDetector&gt;().TargetLocked=null; } public TargetDetector GetComponent&lt;TargetDetector&gt; ()  //락은 상태가 아니면 if(!TargetLocked) {     // 공격자가 Player일 경우     if(!isEnemy)     {         //전체 적 List에서 탐색         foreach(Transform target in CsIdentification.PlayerTargets)         {             //적 List가 비어있지 않으면             if(target)             {                 //현재 List의 객체 Player의 거리 계산                 float dist = Vector3.Distance(transform.position,target.position);                  //다음 List의 객체 와 현재 List의 객체중 Player와 가까운 객체를 타겟팅 함                 if(!TempTarget)TempTarget=target;                 else                 {                     float TempDist = Vector3.Distance(transform.position,TempTarget.position);                     if(TempDist&gt;dist)TempTarget=target;                 }             }         }     } } </pre> <p>Lockon 상태가 안 일 경우 CsIdentification의 Enemy List에 모든 객체 중 가장 Player와 가까운 객체를 target으로 정한다.</p>

CsPlayer

KeyEvent()

```

void KeyEvent ()
{
    //좌우 선회
    if (Input.GetAxis ("Horizontal")>0)
        transform.Rotate (0, 0, -60*Time.smoothDeltaTime);
    else if (Input.GetAxis ("Horizontal")<0)
        transform.Rotate (0, 0, 60 * Time.smoothDeltaTime);

    //상하 회전
    if (Input.GetAxis ("Vertical")>0)
        transform.Rotate (-60 * Time.smoothDeltaTime, 0, 0);
    else if (Input.GetAxis ("Vertical")<0)
        transform.Rotate (60 * Time.smoothDeltaTime, 0, 0);

    //좌우 회전
    if (Input.GetAxis ("LR")<0)
        transform.Rotate (0, -15* Time.smoothDeltaTime, 0);
    else if (Input.GetAxis ("LR")>0)
        transform.Rotate (0, 15 * Time.smoothDeltaTime, 0);

    //미사일 발사
    if (Input.GetKeyDown ("space") && canfire)
    {
        StartCoroutine ("fire");
        AudioSource.PlayClipAtPoint (missile, transform.position);
    }

    //총알 발사
    if (Input.GetButton ("Fire1") && canfireBullet)
    {
        StartCoroutine ("BulletFire");
        AudioSource.PlayClipAtPoint (gun, transform.position);
    }

    //총알 락온 모드
    if (Input.GetButtonDown ("Fire2") && target)
    {
        cam.GetComponent<SmoothFollow>().SetBullet (target);
        LockBullTarget=true;
    }
    if (Input.GetButtonUp ("Fire2"))
    {
        LockBullTarget=false;
        cam.GetComponent<SmoothFollow>().SetDefault ();
    }

    //스피드 업 & 다운
    if (Input.GetAxis ("Turbo") !=0)
    {
        if (speed<=MaxSpeed&&Input.GetAxis ("Turbo")<0)
        {
            speed += 1*Time.smoothDeltaTime;
            needLevelOff= false;
        }
        if (speed>=MinSpeed&&Input.GetAxis ("Turbo")>0)
        {
            speed -= 1*Time.smoothDeltaTime;
            transform.Rotate (new Vector3 (Random.Range (-noise.x, noise.x),
                                           Random.Range (-noise.y, noise.y),
                                           Random.Range (-noise.z, noise.z)));
        }
    }

    //속도 평균 속도로 되돌림
    if (Input.GetButtonUp ("Turbo"))
    {
        needLevelOff= true;
    }
}

```

Z축 방향을 기준으로 Rotation 하면 기체 자체가 제자리에서 좌, 우로 회전하게 되고,

Y축 방향을 기준으로 Rotation 하면 기체의 진행 방향이 좌, 우로 바뀌고

X축 방향을 기준으로 Rotation 하면 기체의 진행 방향이 상, 하로 바뀌게 된다.

실제 비행에서는 X, Y 축만 사용하여 기체를 회전 시킨 뒤 비행하는 것이 정상 이지만, 초보자 또는 게임의 난이도 조절을 위하여 Q, E를 이용하여 기체의 회 전 없이 진행 방향을 좌, 우로 조정 가능하게 하였다.

그리고 기체의 스피드를 느리게 비행 할 경우 기체가 좌우로 흔들리기 때문에 그 효과를 표현하기 위하여 스피드를 Down 시킬 시 기체의 각도가 랜덤하게 바뀌어 흔들리게 하였다.

FireDamage()  
-positionMove ()

```
void FireDamage() {
    // HP 80~ 50
    if (hp < 80 && hp > 50 && damageCheckNum == 0)
    {
        damageCount = 1;
    }
    // HP 50 ~ 30
    else if (hp < 50 && hp > 30 && damageCheckNum == 1)
    {
        damageCount = 2;
    }
    // HP 30 0이하
    else if (hp < 30 && damageCheckNum == 2)
    {
        damageCount = 3;
    }
}
```

각 기체의 HP 구간마다 damageCount 라는 상태 변수를 선언 하여, 기체의 데미지 상태를 체크 할 수 있도록 하였다.

```
if (damageCount < 4) {
    switch (damageCount) {
        case 1: // HP 80~ 50
            fireDamageMove[damageCount - 1] =
                Instantiate(FireDamage0,
                    fireDamagePoint[damageCount - 1].transform.position,
                    Quaternion.identity) as Transform;
            damageCheckNum = 1;
            damageCount = 0;
            break;
        case 2: // HP 50 ~ 30
            fireDamageMove[damageCount - 1] =
                Instantiate(FireDamage1,
                    fireDamagePoint[damageCount - 1].transform.position,
                    Quaternion.identity) as Transform;
            damageCheckNum = 2;
            damageCount = 0;
            break;
        case 3: // HP 30 0이하
            fireDamageMove[damageCount - 1] =
                Instantiate(FireDamage2,
                    fireDamagePoint[damageCount - 1].transform.position,
                    Quaternion.identity) as Transform;
            damageCheckNum = 3;
            damageCount = 0;
            break;
    } //switch
}
void positionMove ()
{
    if (damageCheckNum > 0)
    {
        for (int i = 0; i < damageCheckNum; i++)
        {
            fireDamageMove[i].transform.position = fireDamagePoint[i].transform.position;
        }
    }
}
```

damageCount를 이용하여 기체의 데미지 상태를 파악하고 각 상태 별로 fireDamageMove 배열과 fireDamagePoint 배열의 Index를 control 할 수 있게 하였다. (fireDamageMove -> 기체의 Position과 동일)  
그리고 fireDamageMove 배열에 fireDamagePoint의 Position값을 동일 index에 넣어 주어 particle이 생성 되는 위치를 동일하게 만들어 주었다.