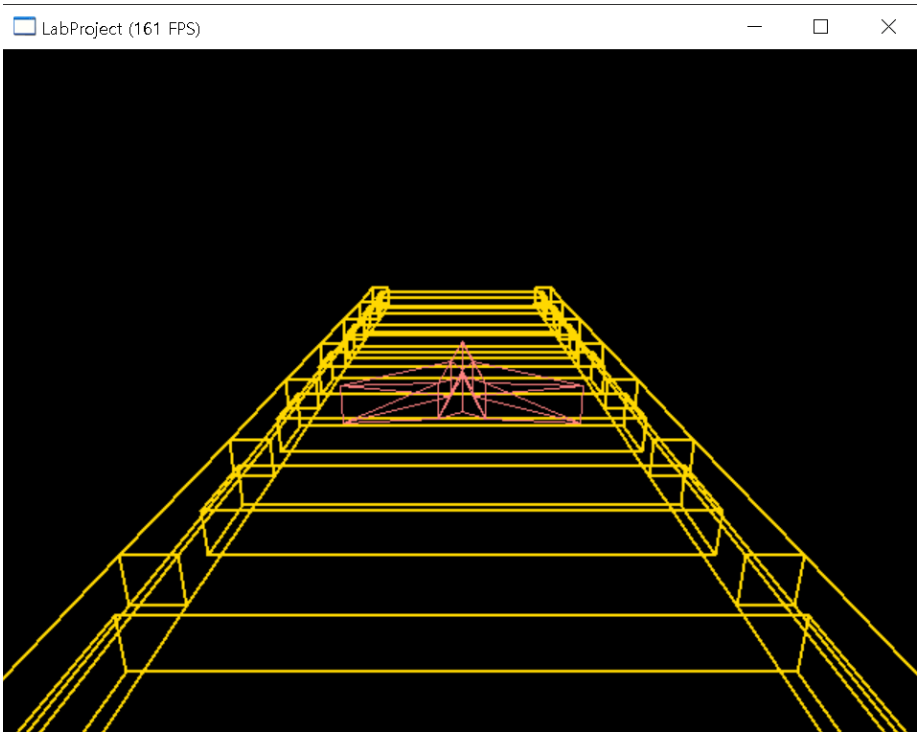
과제 01 보고서

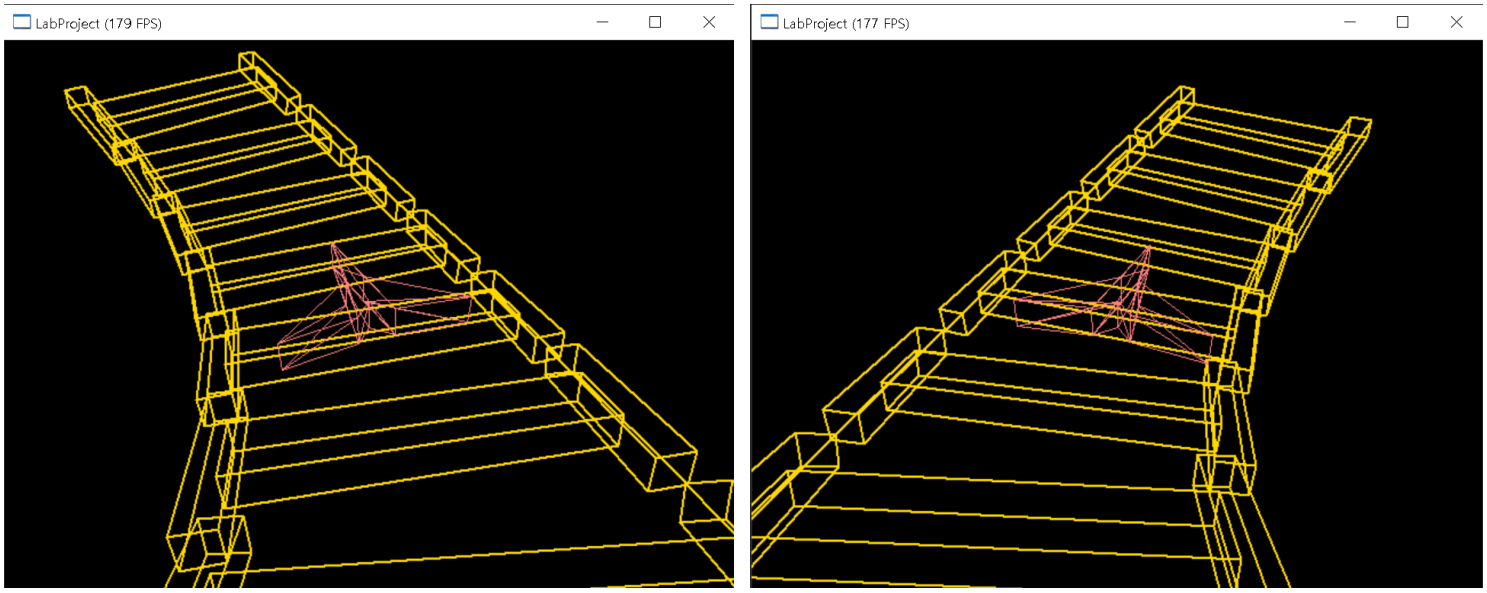
2017180035 장수현

1. 과제에 대한 목표 및 가정
    - 과제 01은 Direct3D를 시작하기 전에 Window API를 이용하여 구현을 해보는 것을 목표로 한다.
    - Mesh를 추가하고 그 mesh를 이용하여 오브젝트를 만드는 것을 목표로 한다.
    - 오브젝트를 무한히 만들 수 있는 것처럼 보이도록 하는 것을 목표로 한다.
    - 오브젝트를 회전하고 원하는 위치에 생성하는 것을 목표로 한다.
    - 기본 Lapproject(Scenes)의 코드는 다 이해하고 있다는 걸 가정으로 한다.

2. 조작법과 실행 결과
    - 조작법
        - .exe 파일을 실행한다. (만약 비주얼 2017에서 실행하고 싶다면 Release - X64로 실행한다.)
        - 방향 전환: 키보드 좌, 우, 상, 하 키보드
        - 총알 발사: 스페이스바
    - 실행 결과



키 입력을 받고 있지 않을 때



키 입력을 받아 이동하는 모습

3. 과제01을 프로그래밍할 때 사용한 자료구조 및 알고리즘

1. Rail Mesh를 추가한다.

➔ 위에서 본 레일 하나의 mesh 모양

```
float fLenWidth = fWidth * 1.0f;
float fLenHeight = fHeight * 0.1f;
float fLenDepth = fDepth * 0.2f;
```

```
std::deque <CRailObject*>     m_pRailObject;
```

- RailObject를 만들기 위해서 나는 deque 자료구조를 사용하였다. CRailObject를 포인터로 가지고 있는 deque를 만들어서 생성과 소멸시 용의 하게 했다.
  ➔ 생성시에는 레일이 deque에 제일 뒤에 생성하고 소멸시에는 제일 앞 레일을 소멸해야 하기 때문에 emplace_back(), front_pop()을 사용하면 생성과 소멸시 좀 더 편리하다.
- 가운데 부분
  레일의 가운데 큐브를 그리기 위하여 width는 그대로 Height, Depth는 0.1, 0.2배씩 한 결과를 이용하며 CPolygon을 동적할당 받아 면들을 만들고 그 면들을 이용하며 큐브를 만든다.
- 왼쪽과 오른쪽 부분

```
float fSideWidth = fWidth * 0.1f;
float fSideHeight = fHeight * 0.1f;
float fSideDepth = fDepth * 0.5f;

float fSideWidthInterval = fWidth + fSideWidth / 2.f;
```

```
pFrontFace = new CPolygon(4);
pFrontFace->SetVertex(0, CVertex(-fSideWidthInterval - fSideWidth, +fSideHeight, -fSideDepth));
pFrontFace->SetVertex(1, CVertex(-fSideWidthInterval + fSideWidth, +fSideHeight, -fSideDepth));
pFrontFace->SetVertex(2, CVertex(-fSideWidthInterval + fSideWidth, -fSideHeight, -fSideDepth));
pFrontFace->SetVertex(3, CVertex(-fSideWidthInterval - fSideWidth, -fSideHeight, -fSideDepth));
SetPolygon(i++, pFrontFace);
```

레일의 왼쪽과 오른쪽 큐브를 그리기 위하여 가운데 부분의 높이와 내 height, width는 동일한 사이즈로 설정을 한다. 또한 depth는 width, height보다 좀 더 길게 하기 위해 0.5배를 한다. 가운데 부분과 동일하게 동적할당을 이용하여 면을 만들고 큐브도 만든다.

왼쪽은 -fSideWidthInterval, 오른쪽은 +fSideWidthInterval로 CVertex를 생성한다.

2. Rail Mesh를 이용하여 Rail Object를 생성한다.
   CGameObject를 상속 하는 CRailObject class를 만든다.

3. CRollerCoasterScene class를 만든다.
   CScene를 상속하는 CRollerCoasterScene class를 만든다.
   - CGameFramework에서 내가 사용할 비행기(롤러코스터 object대신) object를 생성하고 object들을 띄울 CRollerCoasterScene을 생성한다.

```
void CGameFramework::BuildObjects()
{
    CAirplaneMesh *pAirplaneMesh = new CAirplaneMesh(6.0f, 6.0f, 1.0f);
    m_pPlayer = new CAirplanePlayer();
    m_pPlayer->SetPosition(0.0f, 0.0f, 0.0f);
    m_pPlayer->SetMesh(pAirplaneMesh);
    m_pPlayer->SetColor(RGB(255, 130, 130));
    m_pPlayer->SetCameraOffset(XMFLOAT3(0.0f, 5.0f, -15.0f));


    m_nScene = 5; // 키보드 입력을 따로 받기 위해서

    m_pScene = new CRollerCoasterScene();
    m_pScene->BuildObjects();

    m_pScene->m_pPlayer = m_pPlayer;
}
```

프로젝트를 실행했을 경우 보여줄 Scene과 object를 GameFrameWork에서 만들어준다. 이 정보들은 처음에 한번 만들어주고 런타임시에 해주면 프레임을 떨어트리는 요인이 될 수 있으므로 BuildObject()에서 생성해준다.

```
if ((dwDirection != 0) || (cxDelta != 0.0f) || (cyDelta != 0.0f))
{
    if (m_nScene != 5 && (cxDelta || cyDelta)) //
    {
        if (pKeyBuffer[VK_RBUTTON] & 0xF0)
            m_pPlayer->Rotate(cyDelta, 0.0f, -cxDelta);
        else
            m_pPlayer->Rotate(cyDelta, cxDelta, 0.0f);
    }

    if (m_nScene != 5 && dwDirection)
        m_pPlayer->Move(dwDirection, 0.5f);
}
```

나는 다른 씬들을 추가 했을 경우 RollercoasterScene에서만 작동하는 키보드 입력을 따로 받기 위해서 m_nScene을 5로 설정을 하여 GameFrameWork에서 처리하는 키보드 입력은 받기 않도록 하였다. 또한 마우스 입력도 RollerCoasterScene에서는 받지 않도록 하였다.

- 키보드 입력

```
virtual void OnProcessingKeyboardMessage(HWND hWnd, UINT nMessageID, WPARAM wParam, LPARAM lParam);
```

RollerCoasterScene에서만 사용하는 키보드 입력을 위하여 OnProcessingKeyboardMessage()를 추가했다. 자세한 코드설명은 뒤에서 하겠다.

- RollerCoasterScene 생성시 초기화

```
void CRollerCoasterScene::BuildObjects()
{
    pRailCubeMesh = new CRailMesh(4.0f, 4.0f, 4.0f);

    for (int i = 0; i < nRail; ++i) {
        m_pRailObject.emplace_back(nullptr);
        m_pRailObject[i] = new CRailObject;
        m_pRailObject[i]->SetPosition(0.0f, 0.0f, 0.0f + RailInterval * i);
        m_pRailObject[i]->SetMesh(pRailCubeMesh);
        m_pRailObject[i]->SetColor(RGB(255, 216, 0));
    }

    rotationMatrix = Matrix4x4::Identity();

    pBulletMesh = new CCubeMesh(0.5f, 0.5f, 0.5f);

}
```

- 레일을 만들기 위해서는 메쉬가 필요하므로 레일 메쉬를 동적 할당한다.
- nRail만큼(= 5) 레일 object를 만들어준다.
- Rail Object를 사용하기 위해서 동적 할당하고 원하는 위치와 mesh, 색을

설정해준다. (나는 레일을 만들고 싶기 때문에 mesh를 RailCubeMesh로 설정해줬다.)

- RailInterval(= 4.0f)값만큼 위치로 설정해 줬다.
- 레일을 회전 시키기 위한 행렬을 단위 행렬의 형태로 만들어 준다.
- 스페이스바를 눌렀을 때 총알을 만들기 위해서 큐브 mesh를 동적 할당 해준다. (총알은 RollerCoasterScene이 시작되었을 때 바로 보이는 object가 아니므로 우선 mesh만 만들어 둔다.)

4. 키입력을 받아 Rail을 회전한다.

```
void CRollerCoasterScene::OnProcessingKeyboardMessage(HWND hWnd, UINT nMessageID, WPARAM wParam, LPARAM lParam)
{
    static UCHAR pKeyBuffer[256];
    float angle = 2.5f;

    if (GetKeyboardState(pKeyBuffer)) {
        if (pKeyBuffer[VK_UP] & 0xF0) rotationAngle = Vector3::Add(rotationAngle, { angle, 0.f, 0.f });
        if (pKeyBuffer[VK_DOWN] & 0xF0) rotationAngle = Vector3::Add(rotationAngle, { -angle, 0.f, 0.f });
        if (pKeyBuffer[VK_LEFT] & 0xF0) rotationAngle = Vector3::Add(rotationAngle, { 0.f, -angle, 0.f });
        if (pKeyBuffer[VK_RIGHT] & 0xF0) rotationAngle = Vector3::Add(rotationAngle, { 0.f, angle, 0.f });
        if (pKeyBuffer[VK_SPACE] & 0xF0) {
            m_dBulletObject.emplace_back(nullptr);
            m_dBulletObject[BulletNum] = new CBulletObject();
            m_dBulletObject[BulletNum]->SetMesh(pBulletMesh);
            m_dBulletObject[BulletNum]->SetPosition(m_pPlayer->m_xmf3Position.x,
                                                    m_pPlayer->m_xmf3Position.y,
                                                    m_pPlayer->m_xmf3Position.z);
            m_dBulletObject[BulletNum]->SetColor(RGB(255, 0, 0));
            m_dBulletObject[BulletNum]->SetMovingDirection(m_pPlayer->m_xmf3Look);
            BulletNum += 1;
        }
    }
}
```

- 다중 키입력을 받기 위해서 GetKeyboardState()를 사용했다.
- rotationAngle은 레일을 일정 시간마다 생성시에 전 레일과 비교했을 때 어느 축을 기준으로 얼마나 회전했는지 저장하기 위한 XMFLOAT3변수이다.
- UP키를 눌렀을 때에는 x축을 기준으로 양의 방향으로,
  DOWN키를 눌렀을 때는 x축을 기준으로 음의 방향으로
  내가 정해준 각도 만큼인 angle만큼(= 2.5f) 씩 회전 하기 위해서
  rotationAngle의 x에 angle을 Vector3::Add()를 이용하여 더했다.
- RIGHT, LEFT키를 눌렀을 때도 동일한 알고리즘으로 하였다.
  단, x축이 아닌 y축 기준으로 회전하기 위해서 rotationAngle의 y 에 angle값이 들어간다.
- 스페이스바를 눌렀을 경우 첫번째로 BulletObject를 동적 할당 받는다.

  ```
  std::deque<CBulletObject*> m_dBulletObject;
  ```

  m_dBulletObject은 CBulletObject의 포인터를 가지고 있는 deqye로 만들었다.
  메쉬는 BulidObject에서 미리 만들어놨던 BulletMesh로 설정해준다.
  총알의 첫 위치는 플레이어의 위치와 동일한 위치에서 생성된다.
  총알이 앞으로 나가기 위해서 총알의 direction을 현재 플레이어의 look벡터로 설정해준다.

총알을 인덱스 접근하기 위해서 BulletNum의 값을 더해 나간다.

5. 레일의 생성과 소멸

```
void CRollerCoasterScene::Animate(float fElapsedTime)
```

- 레일은 처음 BuildObject에서 만들어주고 내가 원하는 시간마다 추가 생성된다.

```
float timeToMakeRail = 0.10f;
accumulateTime += fElapsedTime;

if (pRailCubeMesh && accumulateTime >= timeToMakeRail) {
    m_pRailObject.emplace_back(nullptr);
    m_pRailObject[nRail] = new CRailObject;

    // 바로 전에 그린 레일의 행렬을 가져와서 지금 그릴 레일 행렬에 저장
    m_pRailObject[nRail]->m_xmf4x4World = m_pRailObject[nRail - 1]->m_xmf4x4World;
    // 키 입력을 받았으면 회전 (사실 없어도 회전은 함)
    m_pRailObject[nRail]->Rotate(rotationAngle.x, rotationAngle.y, rotationAngle.z);
    // 초기화 해주기
    rotationAngle = XMFLOAT3(0.f, 0.f, 0.f);

    m_pRailObject[nRail]->Move(m_pRailObject[nRail]->GetLook(), RailInterval);

    m_pRailObject[nRail]->SetMesh(pRailCubeMesh);
    m_pRailObject[nRail]->SetColor(RGB(255, 216, 0));
    if (nRail < nMaxRail)
        nRail++;
    accumulateTime -= timeToMakeRail;
}
```

- Animate()함수를 돌 때 마다 accumulateTime에 fElapsedTime를 더하다가 accumulateTime가 timeToMakeRail보다 커지는 순간에 레일을 하나 추가한다.
- m_pRailObject의 크기를 하나 추가하기 위해서 emplace_back을 해준다. 이때 nullptr로 해주는 이유는 아직 할당을 받지는 않았지만 앞으로 사용을 해주기 위해서이다. 이 때문에 바로 다음에 동적 할당 받는다.
- 바로 전 레일의 행렬을 가지고 와서 새로 만들 행렬로 설정해주면 전 레일의 위치와 회전 정보를 가지고 올 수 있다.
- 그 다음에 키보드 입력때 설정해 주었던 rotationAngle값을 이용하여 레일을 Rotate해준다. (사실 없어도 회전은 함) 이 말은 키보드 입력을 받지 않았을 경우 rotationAngle여기엔 {0.f, 0.f, 0.f}값이 들어 있기 때문에 Rotate의 의미가 없기 때문이다.
- 다음 키보드 입력을 받을 때에는 rotationAngle가 {0.f, 0.f, 0.f}으로 설정되어 있어야 하기 때문에 rotationAngle = XMFLOAT3(0.f, 0.f, 0.f)로 설정해준다.
- 새로 생기는 레일은 원래 위치보다 조금 더 앞에 있어야 하기 때문에 RailInterval만큼 Move해준다.
- Mesh는 BuildObject에서 만들어준 RailCubeMesh로 설정을 해주고 색도 내가 원하는 색으로 설정해준다.
- accumulateTime =0;이 아니라 accumulateTime -= timeToMakeRail;로 해주는 이유는 0.00000xxxx 값이 조금씩 넘겨져서 if문에 들어오게 될 수 있는데 그냥

0으로 설정을 해버리면 저 작은 값들이 누적되다 보면 조금씩 값이 틀어지게 되므로 timeToMakeRail만큼씩 빼주었다.

```
if (m_pRailObject.size() > nMaxRail) {
    delete m_pRailObject[0];
    m_pRailObject.pop_front();
}
```

nMaxRail 보다 RailObject의 개수가 많아지면 제일 먼저 만들었던 레일을 삭제한다.

6. 플레이어가 움직인다.

```
float Speed = RailInterval * (1 / timeToMakeRail); // 1초에 10개의 레일

// 바로 앞 레일의 look, up, right로 내 player설정
m_pPlayer->m_xmf3Look = m_pRailObject[nRail - OnPlayerFrontRailInterval]->GetLook();
m_pPlayer->m_xmf3Up = m_pRailObject[nRail - OnPlayerFrontRailInterval]->GetUp();
m_pPlayer->m_xmf3Right = m_pRailObject[nRail - OnPlayerFrontRailInterval]->GetRight();

m_pPlayer->Move(DIR_FORWARD, Speed * fElapsedTime);
float timeToMakeRail = 0.10f;
```

- 0.10초 마다 레일을 새로 생성한다. 1초에는 10개의 레일을 만드는 것과 같다. 속도는 거리 / 시간 이므로 이를 이용하여 속도를 구할 수 있다.
- 플레이어의 Look, Up, Right를 바로 앞의 레일의 Look, Up, Right벡터로 설정해준다. 이렇게 되면 플레이어는 앞방향으로만 이동을 하면 된다. 얼만큼 이동하는지는 거리 = 속도 * 시간 이므로 위에서 구한 속도와 fElapsedTime를 곱한 만큼 이동을 해준다.

7. 총알이 앞으로 나가고 삭제된다.

```
for (const auto& bullet : m_dBulletObject) {
    bullet->Move(bullet->m_xmf3MovingDirection, bullet->m_fBulletSpeed * bullet->m_fElapsedTimes);
    if (m_dBulletObject.size() > 0 &&
        m_dBulletObject[0]->GetPosition().z > m_pPlayer->m_xmf3Position.z + fBulletMaxdistance) {
        delete m_dBulletObject[0];
        m_dBulletObject.pop_front();
        BulletNum--;
    }
}
```

- Bullet을 플레이어가 이동하는 원리와 비슷하게 이동을 한다. 방향은 MovingDirection으로 거리는 플레이어가 이동할 때 움직이는 거리와 같은 방법으로 계산해서 이동을 한다.
- 총알이 적어도 하나 있고 플레이어의 z값보다 fBulletMaxdistance 떨어지게 된다면 제일 앞에 만들었던 총알을 삭제한다. 총알은 플레이어보다 빠르기 때문에 가장 먼저 멀어진 총알이 제일 먼저 만들어진 총알이 될거라고 생각했다.