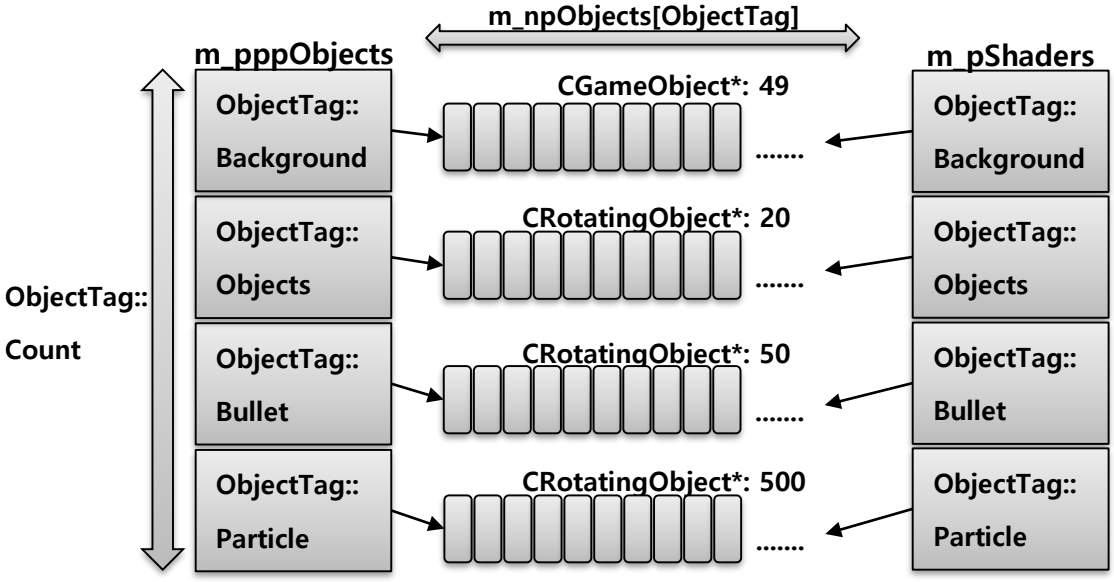
[과제 해결방법]

➢ [프로그램 변경사항]

1. 윈도우 API 기반으로 작성된 과제 01 프로그램을 DirectX12 기반으로 수정하였다.

- 가정
  ① 과제 02 는 따라하기 14 번을 기반으로 진행한다.
  ② 과제 01 의 기능을 그대로 옮겨온다. (3 인칭을 제외한 카메라 모드 제한)
  ③ 플레이어 캐릭터를 제외한 오브젝트를 그릴 때 오브젝트의 종류별로 다른 인스턴싱셰이더 클래스를 사용한다.
  ④ 셰이더 클래스는 오브젝트들을 설정된 상태대로 움직이고 화면에 그리는 것 외의 동작을 하지 않는다.
  ⑤ 모든 물리 처리 및 오브젝트의 상태를 변경하는 행위는 씬에서 모두 처리된다.

- 사용한 자료구조
  STL 컨테이너를 사용하지 않고, 플레이어 캐릭터를 제외한 여러 오브젝트를 하나의 CGameObject 삼중포인터를 이용한 2 차원 포인터배열을 사용하여 관리한다.



- 구현

① 카메라 모드를 3 인칭으로 제한하기 위해 플레이어를 생성할 때 카메라의 기본 모드를 3 인칭으로 설정한 후 stdafx.h 에 #define CAMERA_MODE_ACTIVE 0 를 선언하고 윈도우메시지 VK_F1~VK_F3 부분을 #if CAMERA_MODE_ACTIVE 와 #endif 로 감싸서 실행되지 않도록 비활성화 하였다.

소스코드: GameFramework.cpp – CGameFramework::OnProcessingKeyboardMessage() – 207 line

② 씬에서 typedef enum {} ObjectTag;를 사용하여 오브젝트들의 종류를 각각 Background, Objects, Bullet, Particle 로 나누고 enum 의 마지막에 Count 를 둬서 int 타입의 m_npObjects 배열길이를 ObjectTag::Count 로 설정하고 CGameObject*** m_pppObjects 와 CInstancingShader* m_pShaders 에 일차적으로 ObjectTag::Count 길이의 배열을 동적할당하고 m_pppObjects 배열의 각 태그 위치에 m_npObjects 배열의 태그 위치에 저장한 오브젝트의 개수 값만큼 배열을 동적할당하여 오브젝트의 종류별 배열을 만든 후에 각 배열에 Background 태그일 경우 WallMesh 를 가진 CGameObject 를, Objects, Bullet, Particle 태그일 경우 서로 다른 크기의 CubeMesh 를 가진 CRotatingObject 를 생성하고 m_pShaders 배열의 각 태그에 맞는 CInstancingShader 에 BuildObject 함수를 통하여 CGameObject 배열과 m_npObjects 에 저장된 개수 값을 Shader 에 넘겨서 CInstancingShacer 가 가지고 있는 m_ppObjects 와 m_nObjects 에 값을 저장하고 이를 이용해 오브젝트들을 화면에 그리게 할 수 있도록 하였다.

소스코드: Scene.cpp - CScene::BuildObjects() - 81 line

③ Background 태그의 벽 오브젝트들은 과제 01 과 같은 x, y 너비가 40, z 너비가 10 이며 색을 가진 Objects 태그의 큐브들의 크기는 4, Bullet 태그의 총알은 2, Particle 태그의 파티클은 1 로 설정하였고 각각 49 개, 20 개, 50 개, 500 개씩 m_pppObjects 의 해당하는 태그 위치에 동적

할당 되어있고 Mesh 의 정점들은 모두 RANDOM_COLOR 메크로를 사용하여 얻은 임의의 색을 가지고 있다.

소스코드: Scene.cpp - CScene::BuildObjects() - 81 line

Mesh.cpp – CCubeMeshDiffused:: CCubeMeshDiffused() – 93 line,

CWallMeshDiffused:: CWallMeshDiffused() – 366 line

④ 화면에 오브젝트가 그려질 때 오브젝트의 색은 화면이 출력되기 전에 Mesh 에서 정해진 정점의 색에 CInstancingShader:: UpdateShaderVariables()에서 비디오메모리에 전달된 색을 더한 후 클램핑한 색으로 정해지므로 CGameObject 에 typedef enum {} ObjectType 와 멤버변수 m_Tag 를 추가하여 CScene 의 BuildObject 에서 오브젝트들을 생성할 때 생성된 오브젝트가 어떤 타입인지 알 수 있게 하고 CInstancingShader::UpdateShaderVariables()에서 오브젝트의 m_Tag 값에 따라 더해지는 색을 다르게 하여 오브젝트가 원하는 색을 가지도록 하였다.

소스코드: GameObject.h – 클래스 선언문 - 10 line

Shaders.cpp - CInstancingShader::UpdateShaderVariables() – 352 line

⑤ 오브젝트들의 충돌처리를 위해 과제 01 과 같은 방식으로 BoundingOrientedBox 를 멤버변수에 추가하였고 Mesh 에서 바인딩박스를 읽어와서 초기화 해주었으며 CGameObject:: Animate()에 바인딩박스를 업데이트 하는 부분을 구현하였다. 추가로 m_bActive 멤버 변수를 추가하여 오브젝트의 활성화 여부를 나타내고 호출되었을 때 내부 변수의 값이 바뀌는 함수의 앞부분에 m_bActive 가 false 일 경우 return 하게하는 부분을 추가하여 오브젝트가 비활성화 상태일 때는 움직이지도 그려지지도 않게 하였고 m_bUpdatedWorldMtx 멤버 변수를 추가하여 SetPosition()과 Rotate()에서 월드 행렬이 수정되는 경우에 값을 true 로 변경하여 Animate()에서 m_bUpdatedWorldMtx 가 true 일 경우에만

바인딩박스를 업데이트하게해서 벽 오브젝트같이 잘 움직이지 않는 오브젝트들이 프레임마다 쓸데없이 행렬 곱을 하는 것을 방지하였다.

소스코드: GameObject.h – 클래스 선언문 - 10 line

GameObject.cpp – CGameObject::Animate() – 92 line

2. 충돌처리과정에서 움직이는 객체들이 과도하게 겹칠 경우 부들부들 떠는 현상을 해결하였다.

- 가정
  ① 오브젝트들이 과도하게 겹쳐질 경우 충돌 처리로 인한 방향 벡터의 반전이 수시로 일어나 충돌중인 두 오브젝트가 부들부들 떠는 현상이 발생한다.
  ② 과제 01 에서 이미 구현된, 충돌했을 경우 바뀐 방향 벡터의 방향으로 오브젝트를 밀어주는 기능을 강화하면 해결될 것이다
  ③ 충돌 시마다 방향 벡터가 계속 반전되는 것이 원인이므로 방향 벡터를 반전하는 대신 충돌한 오브젝트들이 서로 반대 방향으로 이동할 수 있도록 방향 벡터를 고정한다.
  ④ 2 번과 3 번 해결 방안을 둘 다 적용한다.

- 구현
  ① 두 오브젝트들이 충돌하였을 경우 먼저 두 오브젝트를 각각 A 와 B 라고 했을 때, 충돌 후 A 는 B 의 중심점에서 A 의 중심점을 향하는 벡터를 가지게 하고 B 는 A 의 중심점에서 B 의 중심점을 향하는 벡터를 가지게 하여 오브젝트의 타입에 따라서 새로 얻은 벡터의 값을 조정한 후 정규화하는 방식으로 방향 벡터를 조정해주었다.
  Ex) x 축을 따라 움직이는 붉은색 큐브의 경우 충돌하여 얻은 벡터의 y, z 값을 0 으로 초기화한 후 정규화하여 방향 벡터를 얻게 하였다. 이런 식으로 벡터를 조정해 줬을 경우 오브젝트가 충돌했을 때 프레임마다 무분별하게 벡터의 방향이 바뀌어 부들거리는 것을 막을 수 있다.

소스코드: Scene.cpp - CScene::PhysicsProcessing() - 701 line

② 과제 01 에서는 두 오브젝트의 충돌 후 방향 벡터를 다시 설정해주고 나면 오브젝트를 설정된 방향으로 Move()를 사용하여 오브젝트에 설정된 속도를 사용해 밀어주었는데 과제 02 에서는 때 곱해주는 속도 값을 오브젝트에 설정된 값이 아닌 고정 값으로 바꾸어주었다.

소스코드: Scene.cpp - CScene::PhysicsProcessing() – 762 line

3. 벽과 부딪히는 빨간색, 초록색 큐브 오브젝트가 벽을 뚫는 현상을 해결하였다.

- 가정
  - ① 오브젝트가 벽에 살짝 겹쳤을 경우 부들거린다.
  - ② 오브젝트가 부들거리는 중 다른 오브젝트와 충돌하면 위치 보정을 위해 오브젝트를 밀어주는 기능 때문에 오브젝트가 벽에 많이 겹치게 된다.
  - ③ 오브젝트가 벽에 많이 겹쳤을 경우 복도 밖으로 벗어난 것으로 판단하여 재생성 하기때문에 오브젝트가 갑자기 사라진다.
  - ④ 충돌 처리로 인한 방향 벡터의 반전이 수시로 일어나 충돌중인 오브젝트가 부들부들 떠는 현상이 발생한다.
  - ⑤ 오브젝트가 벽과 충돌했을 경우 단순히 방향 벡터를 역전시키는게 아니라 벽의 법선 벡터 방향으로 오브젝트의 방향 벡터를 고정하면 해결될 것이다.

- 구현
  - ① 오브젝트들이 벽에 충돌했을 경우에는 보통 반사 벡터를 구해서 오브젝트의 방향 벡터를 바꿔주지만 빨간색, 초록색 큐브 오브젝트의 경우 단순히 축을 따라 움직이기 때문에 반사 벡터를 굳이 구할 필요가 없다고 판단하였다. 따라서 충돌 후 오브젝트의 방향 벡터를 바꾸는

방식을 기존의 벡터의 방향만 바꿔주는 형식에서 충돌한 벽의 법선 벡터를 그대로 적용하는 방식으로 변경하였다.

소스코드: Scene.cpp - CScene::PhysicsProcessing() – 505 line

② 빨간색, 초록색 오브젝트의 충돌 후 방향 벡터를 변경하는 방식을 충돌한 면의 법선 벡터를 그대로 적용하는 방식으로 적용했기 때문에 오브젝트가 다른 면과 충돌하면 문제가 발생하므로 빨간색, 초록색 오브젝트의 생성 위치를 충돌해야하는 면 외의 다른 면과 충돌하지 않도록 벽면에서 일정거리 떨어지게 하였다.

소스코드: Scene.cpp - CScene:: ResetObjects() - 871 line

4. 따라하기 16 의 프러스텀컬링을 연습 삼아 적용하였고 따라하기 14 의 CInstancingShader 의 Render() 함수 오류를 수정하였다.

- 가정

① 모든 게임 객체들이 물리 처리를 위해 월드 공간상의 바인딩박스를 가지고 있기 때문에 별다른 처리 없이 게임 객체와 카메라 프러스텀을 충돌 처리 할 수 있다.

② 인스턴싱을 사용중인 오브젝트들을 Render 할 때 CInstancingShader:: Render()내부를 보면 CObjectsShader::Render()를 사용하는데, 이 함수에는 오브젝트들을 전부 그리는 부분이 존재하고 오브젝트의 Render() 내부에는 Mesh 가 있을 경우 Mesh 의 Render 를, Shader 가 있을 경우 Shader 의 Render 를 한번 더 수행한다. 이때 CInstancingShader 에 연결된 오브젝트들은 0 번 위치의 오브젝트만 Mesh 를 가지고 있는데 앞서 설명한 함수 구조 때문에 0 번 위치의 오브젝트는 Mesh 정보가 입력조립기에 두 번 들어가게 되고 이 때문에 프로그램을 실행하면 허공에 가만히 떠서 아무것도 안하는 의미불명의 오브젝트가 생성된다.

- 구현

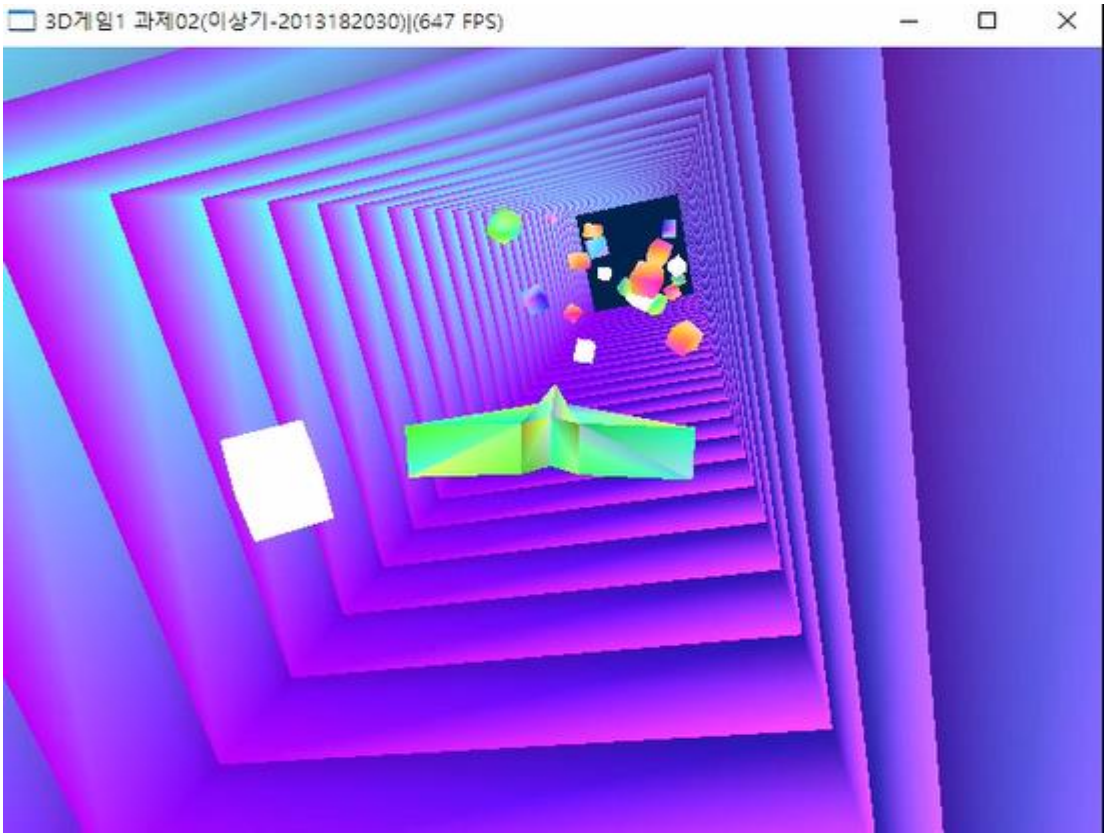
① 프러스텀컬링은 따라하기 16 에서 월드 공간에 존재하는 절두체와 충돌 체크를 하기 위해 모델좌표계의 바인딩박스를 월드변환행렬로 곱해주는 부분을 제거하고 충돌 체크를 수행하는 바인딩박스를 물리 처리를 위해 사용하는 m_OOBB 로 교체하였다.

소스코드: GameObject.cpp - CGameObject::IsVisible() – 158 line

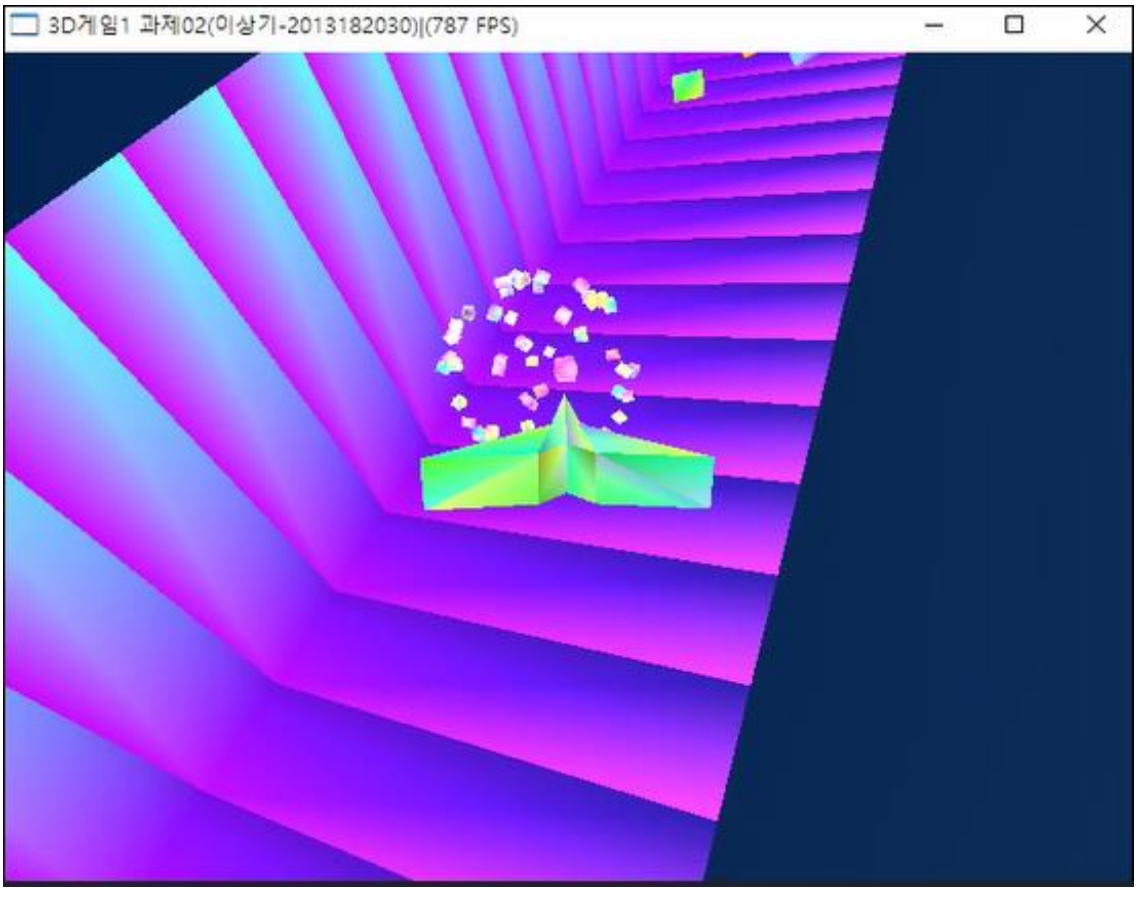
② 허공에 생성되는 의미불명의 오브젝트를 제거하기 위해 CObjectShader:: Render()에서 해당 for 문을 삭제하였다. 인스턴싱을 사용하는 이상 이 부분은 필요가 없고, 만약 CObjectShader 를 사용하는 오브젝트가 있을 때 CObjectShader 에 사용중인 오브젝트가 연결되어 있다면 Render()함수 내에서 순환 참조가 일어나서 스택오버플로우를 일으키며 프로그램이 죽기 때문에 굳이 예외처리를 하기 위해 멤버 변수를 추가하지 않아도 해결할 수 있는 방법인 삭제를 선택하였다.

소스코드: Shader.cpp - CObjectShader:: Render() – 274 line

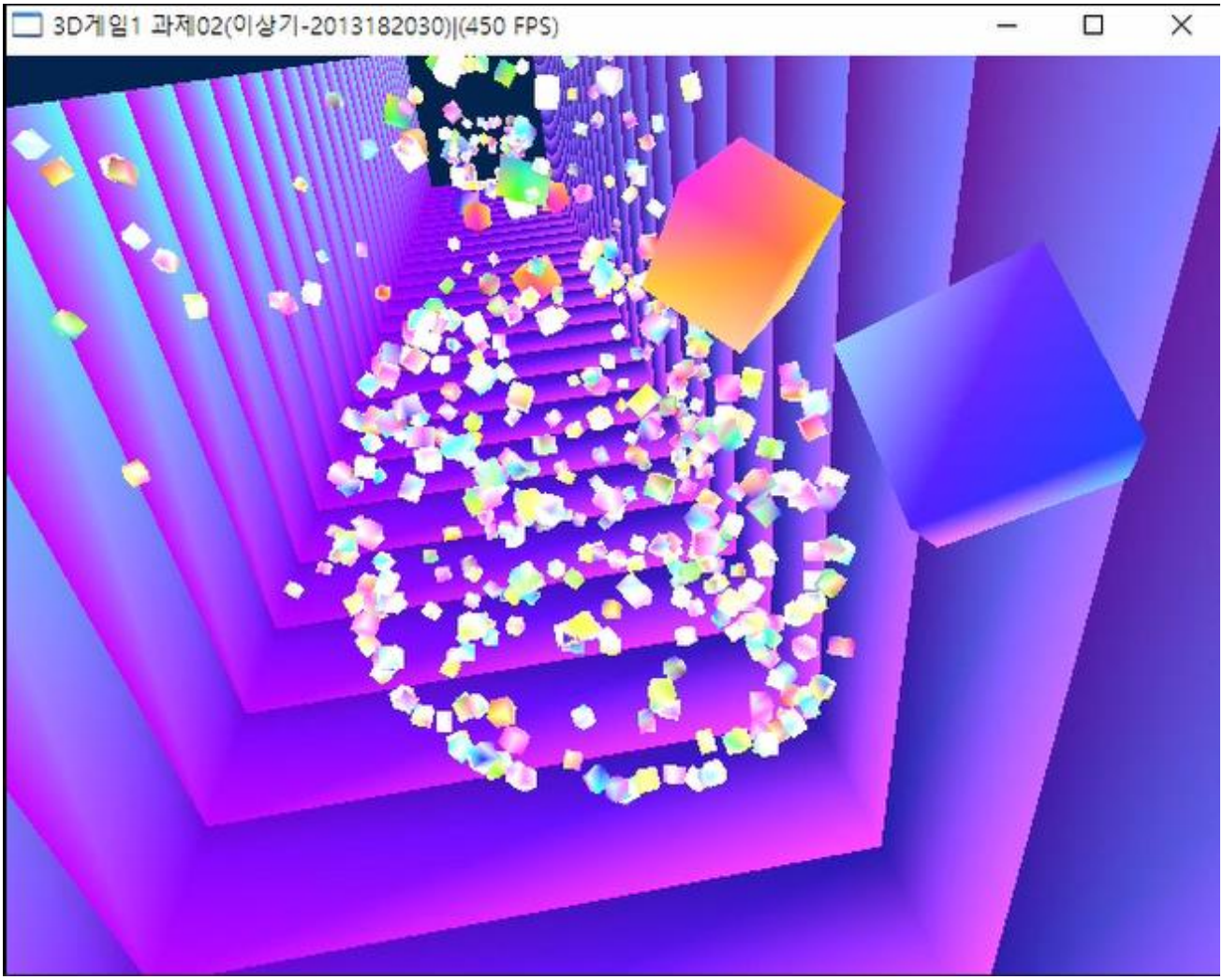
➢ [실행결과]



플레이 화면



오브젝트 파괴



게임 오버

[조작법]

키	설명
W	카메라가 바라보는 방향으로 **(앞으로) 이동**한다.
A	카메라가 바라보는 방향에서 **왼쪽으로 이동**한다.
S	카메라가 바라보는 방향의 반대로 **(뒤로) 이동**한다.
D	카메라가 바라보는 방향에서 **오른쪽으로 이동**한다.
R	카메라가 바라보는 방향의 **위쪽으로 상승**한다.
F	카메라가 바라보는 방향의 **아래쪽으로 하강**한다.
마우스 왼쪽 버튼 + 좌우 상하 이동	**누른 상태**로 마우스를 움직이면 **카메라가 바라보는 방향이 마우스를 따라 움직인다.**
마우스 오른쪽 버튼 + 좌우 이동	**누른 상태**로 마우스를 움직이면 **카메라가 바라보는 방향이 z 축으로 회전한다.**
마우스 오른쪽 버튼 + 상하 이동	**누른 상태**로 마우스를 움직이면 **카메라가 바라보는 방향이 마우스를 따라 움직인다.**
SPACE	플레이어가 **탄환을 발사**한다.
ESC	**프로그램 종료**