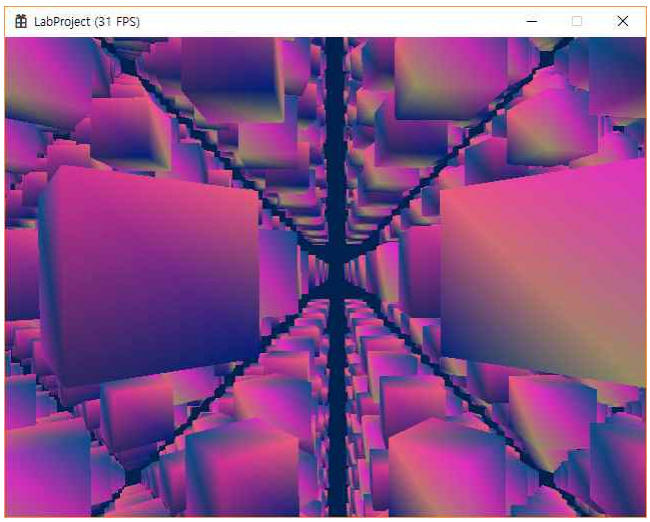
## □ 예제 프로그램 11: LabProject10(회전하는 직육면체 그리기)

예제 프로그램 LabProject09를 기반으로 하여 회전하는 직육면체들(9261 개)을 다음 그림과 같이 그려보도록 하자.




① 새로운 프로젝트의 생성

먼저 새로운 프로젝트 LabProject10를 생성한다. “LabProjects" 솔루션을 열고 솔루션 탐색기에서 마우스 오른쪽 버튼으로 『솔루션 LabProjects』를 선택하고 메뉴에서 『추가』, 『새 프로젝트』를 차례로 선택한다. 그러면 『새 프로젝트 대화상자』가 나타난다. 그러면 프로젝트 이름 “LabProject10”를 입력하고 『확인』을 선택한다.

❶ 파일 탐색기에서 프로젝트 “LabProject09” 폴더의 다음 파일을 선택하여 프로젝트 “LabProject10” 폴더에 복사한다.

- GameFramework.h
- GameFramework.cpp
- Mesh.h
- Mesh.cpp
- Camera.h
- Camera.cpp
- Object.h
- Object.cpp
- Scene.h
- Scene.cpp
- Shader.h

- Shader.cpp
- stdafx.h
- Timer.h
- Timer.cpp
- Shaders.hlsl

❷ 위에서 복사한 파일을 Visual Studio 솔루션 탐색기에서 프로젝트 “LabProject10”에 추가한다. 오른쪽 마우스 버튼으로 『LabProject10』을 선택하고 『추가』, 『기존 항목』를 차례로 선택한다. 그러면 “기존 항목 추가” 대화 상자가 나타난다. 다음 파일들을 마우스로 선택(Ctrl+선택)하여 『추가』를 누르면 선택된 파일들이 프로젝트 “LabProject10”에 추가된다.

② LabProject10.cpp 파일 수정하기

이제 “LabProject10.cpp" 파일의 내용을 “LabProject09.cpp" 파일의 내용으로 바꾸도록 하자. “LabProject09.cpp" 파일의 내용 전체를 “LabProject10.cpp" 파일로 복사한다. 이제 “LabProject10.cpp" 파일에서 “LabProject09”을 “LabProject10”로 모두 바꾼다.
그리고 “LABPROJECT09”을 “LABPROJECT10”으로 모두 바꾼다.

③ “GameObject.h” 파일 수정하기

❶ “CGameObject” 클래스에 상수 버퍼의 생성과 갱신을 위한 내용을 다음과 같이 추가한다.

```
public:
//상수 버퍼를 생성한다.
   virtual void CreateShaderVariables(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList);
//상수 버퍼의 내용을 갱신한다.
   virtual void UpdateShaderVariables(ID3D12GraphicsCommandList *pd3dCommandList);
   virtual void ReleaseShaderVariables();
```

❷ “CGameObject” 클래스에 게임 객체의 이동과 회전을 처리하기 위한 내용을 다음과 같이 추가한다.

```
//게임 객체의 월드 변환 행렬에서 위치 벡터와 방향(x-축, y-축, z-축) 벡터를 반환한다.
   XMFLOAT3 GetPosition();
   XMFLOAT3 GetLook();
   XMFLOAT3 GetUp();
   XMFLOAT3 GetRight();

//게임 객체의 위치를 설정한다.
   void SetPosition(float x, float y, float z);
   void SetPosition(XMFLOAT3 xmf3Position);

//게임 객체를 로컬 x-축, y-축, z-축 방향으로 이동한다.
```

```
    void MoveStrafe(float fDistance = 1.0f);
    void MoveUp(float fDistance = 1.0f);
    void MoveForward(float fDistance = 1.0f);

//게임 객체를 회전(x-축, y-축, z-축)한다.
    void Rotate(float fPitch = 10.0f, float fYaw = 10.0f, float fRoll = 10.0f);
```

④ “GameObject.cpp” 파일 수정하기

❶ “CGameObject” 클래스의 CreateShaderVariables() 함수와 ReleaseShaderVariables() 함수를 다음과 같이 정의한다.

```
void CGameObject::CreateShaderVariables(ID3D12Device *pd3dDevice,
ID3D12GraphicsCommandList *pd3dCommandList)
{
}

void CGameObject::ReleaseShaderVariables()
{
}
```

❷ “CGameObject” 클래스의 UpdateShaderVariables() 함수를 다음과 같이 정의한다.

```
void CGameObject::UpdateShaderVariables(ID3D12GraphicsCommandList *pd3dCommandList)
{
    XMFLOAT4X4 xmf4x4World;
    XMStoreFloat4x4(&xmf4x4World, XMMatrixTranspose(XMLoadFloat4x4(&m_xmf4x4World)));
//객체의 월드 변환 행렬을 루트 상수(32-비트 값)를 통하여 셰이더 변수(상수 버퍼)로 복사한다.
    pd3dCommandList->SetGraphicsRoot32BitConstants(0, 16, &xmf4x4World, 0);
}
```

❸ “CGameObject” 클래스의 Render() 함수를 다음과 같이 수정한다.

```
void CGameObject::Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera)
{
    OnPrepareRender();

//객체의 정보를 셰이더 변수(상수 버퍼)로 복사한다.
    UpdateShaderVariables(pd3dCommandList);

    if (m_pShader) m_pShader->Render(pd3dCommandList, pCamera);

    if (m_pMesh) m_pMesh->Render(pd3dCommandList);
}
```

❹ “CGameObject” 클래스의 게임 객체의 이동과 회전을 처리하기 위한 함수를 다음과 같이 정의한다. 지금까지는 게임 객체의 위치를 (0, 0, 0)으로 설정하여 게임 객체를 하나만 생성하였다. 여러 개의 게임 객체를 생성하려면 게임 객체의 위치를 설정하기 위한 함수가 필요하다.

```
void CGameObject::SetPosition(float x, float y, float z)
{
   m_xmf4x4World._41 = x;
   m_xmf4x4World._42 = y;
   m_xmf4x4World._43 = z;
}

void CGameObject::SetPosition(XMFLOAT3 xmf3Position)
{
   SetPosition(xmf3Position.x, xmf3Position.y, xmf3Position.z);
}

XMFLOAT3 CGameObject::GetPosition()
{
   return(XMFLOAT3(m_xmf4x4World._41, m_xmf4x4World._42, m_xmf4x4World._43));
}

//게임 객체의 로컬 z-축 벡터를 반환한다.
XMFLOAT3 CGameObject::GetLook()
{
   return(Vector3::Normalize(XMFLOAT3(m_xmf4x4World._31, m_xmf4x4World._32,
m_xmf4x4World._33)));
}

//게임 객체의 로컬 y-축 벡터를 반환한다.
XMFLOAT3 CGameObject::GetUp()
{
   return(Vector3::Normalize(XMFLOAT3(m_xmf4x4World._21, m_xmf4x4World._22,
m_xmf4x4World._23)));
}

//게임 객체의 로컬 x-축 벡터를 반환한다.
XMFLOAT3 CGameObject::GetRight()
{
   return(Vector3::Normalize(XMFLOAT3(m_xmf4x4World._11, m_xmf4x4World._12,
m_xmf4x4World._13)));
}

//게임 객체를 로컬 x-축 방향으로 이동한다.
void CGameObject::MoveStrafe(float fDistance)
{
   XMFLOAT3 xmf3Position = GetPosition();
   XMFLOAT3 xmf3Right = GetRight();
   xmf3Position = Vector3::Add(xmf3Position, xmf3Right, fDistance);
   CGameObject::SetPosition(xmf3Position);
}

//게임 객체를 로컬 y-축 방향으로 이동한다.
void CGameObject::MoveUp(float fDistance)
{
   XMFLOAT3 xmf3Position = GetPosition();
   XMFLOAT3 xmf3Up = GetUp();
   xmf3Position = Vector3::Add(xmf3Position, xmf3Up, fDistance);
```

```
    CGameObject::SetPosition(xmf3Position);
}

//게임 객체를 로컬 z-축 방향으로 이동한다.
void CGameObject::MoveForward(float fDistance)
{
    XMFLOAT3 xmf3Position = GetPosition();
    XMFLOAT3 xmf3Look = GetLook();
    xmf3Position = Vector3::Add(xmf3Position, xmf3Look, fDistance);
    CGameObject::SetPosition(xmf3Position);
}

//게임 객체를 주어진 각도로 회전한다.
void CGameObject::Rotate(float fPitch, float fYaw, float fRoll)
{
    XMMATRIX mtxRotate = XMMatrixRotationRollPitchYaw(XMConvertToRadians(fPitch),
XMConvertToRadians(fYaw), XMConvertToRadians(fRoll));
    m_xmf4x4World = Matrix4x4::Multiply(mtxRotate, m_xmf4x4World);
}
```

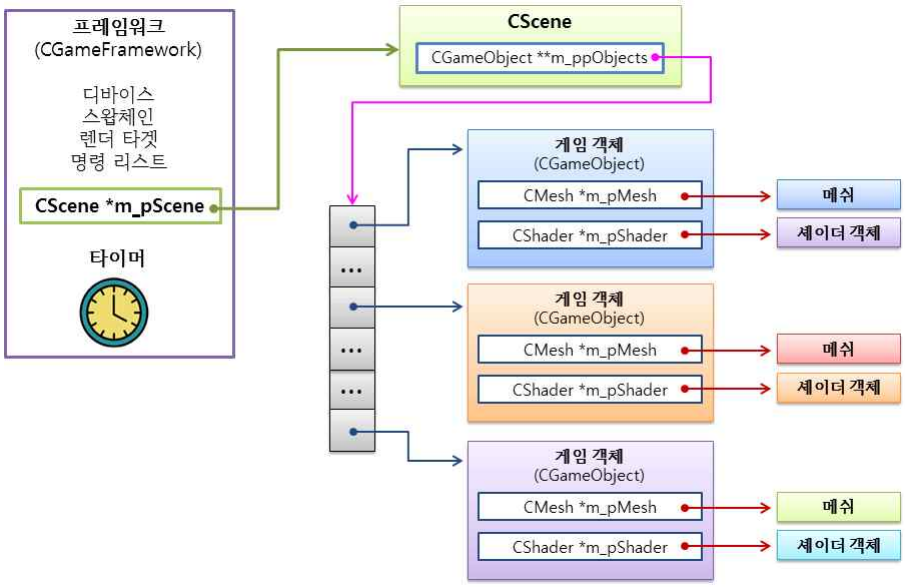
⑤ “Shader.h” 파일 수정하기

❶ “Shader.h” 파일과 “Shader.cpp” 파일에서 “CDiffusedShader” 클래스 이름을 “CPlayerShader”로 모두 변경한다.

❷ “Shader.h” 파일의 앞부분에 다음을 추가한다.

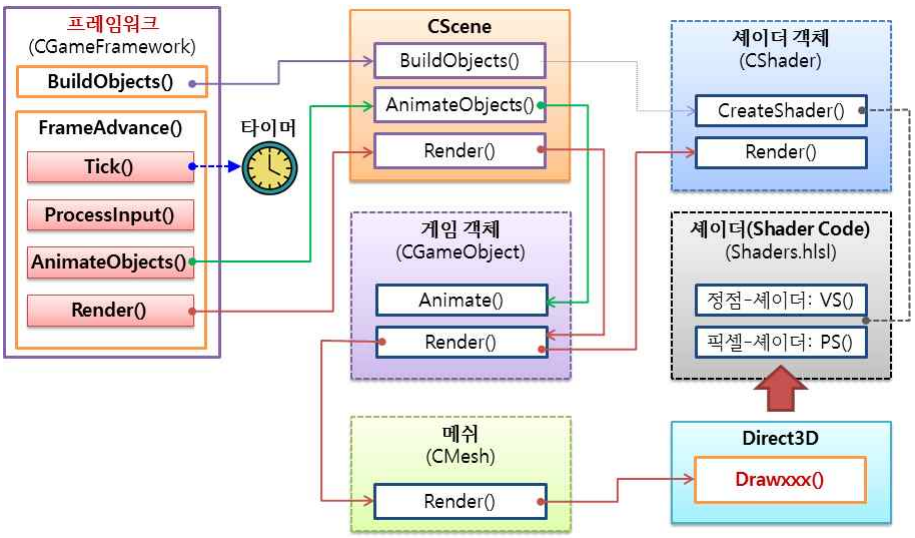
```
#include "Object.h"
```

❸ “CObjectsShader” 클래스를 다음과 같이 선언한다.




게임 객체가 메쉬와 쉐이더 객체를 포함하는 구조

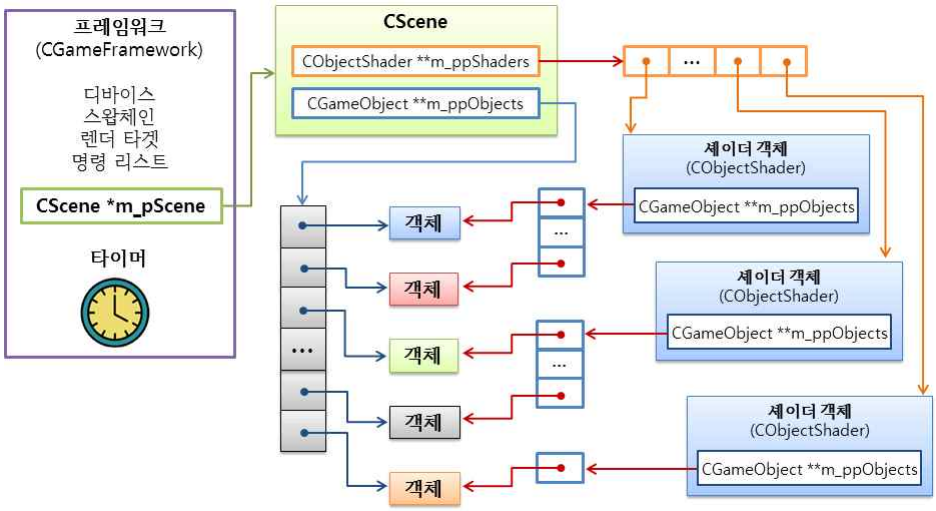
따라하기 10은 위의 그림과 같이 게임 객체(CGameObject)가 셰이더 객체(CShader)와 메쉬(CMesh)에 대한 포인터를 가지고 있는 구조이다. 그리고 씬 객체는 게임 객체들의 배열을 가지고 있으며 씬을 렌더링하기 위해서 게임 객체들을 순서대로 렌더링한다. 각 게임 객체를 렌더링하기 위해서 매번 셰이더 객체를 렌더링(셰이더 정보 - 정점 레이아웃, 정점 셰이더, 픽셀 셰이더를 설정)하고 메쉬 정보를 설정해야 한다. 이러한 구조에서는 각 객체를 렌더링하기 위하여 Direct3D 디바이스의 상태를 변경해야 한다. 씬을 구성하는 객체가 아주 많은 경우 상태 변경을 위한 시간이 늘어날 수 있으므로 프로그램의 프레임 레이트가 떨어질 수 있다.
다음 그림은 게임 객체가 메쉬와 셰이더 객체를 포함하는 구조의 함수 호출을 보이고 있다.



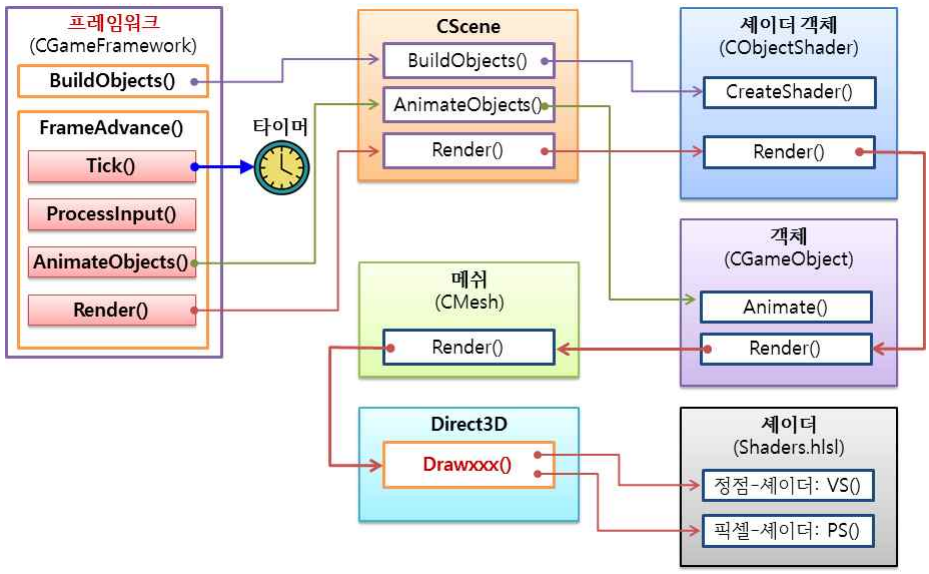

게임 객체가 메쉬와 셰이더 객체를 포함하는 구조의 함수 호출

따라하기 11은 다음 그림과 같이 셰이더 객체가 게임 객체들의 배열을 가지고 있는 구조를 가지고 있다

(셰이더 객체가 메쉬에 대한 포인터를 가질 수도 있다). 그리고 씬 객체는 셰이더 객체들의 배열을 가지고 있으며 씬을 렌더링하기 위해서 셰이더 객체들을 순서대로 렌더링한다. 각 셰이더 객체를 렌더링하기 위해서 셰이더 정보를 한번만 설정하고 게임 객체들의 배열에 포함된 모든 게임 객체들을 순서대로 렌더링하면 된다. 이러한 구조에서는 씬을 렌더링하기 위하여 셰이더 정보를 객체의 개수만큼 설정하지 않아도 된다. 결과적으로 같은 셰이더를 사용하여 렌더링하는 게임 객체들을 그룹화하기 때문에 Direct3D 디바이스의 상태 변화가 더 적게 될 수 있다. 이것으로 프로그램의 프레임 레이트가 향상될 수 있다.




셰이더 객체가 메쉬와 게임 객체를 포함하는 구조




셰이더 객체가 게임 객체를 포함하는 구조의 함수 호출

```
//"CObjectsShader" 클래스는 게임 객체들을 포함하는 셰이더 객체이다.
class CObjectsShader : public CShader
{
```

```
public:
    CObjectsShader();
    virtual ~CObjectsShader();

   virtual void BuildObjects(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList);
   virtual void AnimateObjects(float fTimeElapsed);
   virtual void ReleaseObjects();

   virtual D3D12_INPUT_LAYOUT_DESC CreateInputLayout();
   virtual D3D12_SHADER_BYTECODE CreateVertexShader(ID3DBlob **ppd3dShaderBlob);
   virtual D3D12_SHADER_BYTECODE CreatePixelShader(ID3DBlob **ppd3dShaderBlob);

   virtual void CreateShader(ID3D12Device *pd3dDevice, ID3D12RootSignature
*pd3dGraphicsRootSignature);

   virtual void ReleaseUploadBuffers();

   virtual void Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera);

protected:
   CGameObject **m_ppObjects = NULL;
   int m_nObjects = 0;
};
```

⑥ “Shader.cpp” 파일 수정하기

❶ “CObjectsShader” 클래스의 생성자와 소멸자를 다음과 같이 정의한다.

```
CObjectsShader::CObjectsShader()
{
}

CObjectsShader::~CObjectsShader()
{
}
```

❷ “CObjectsShader” 클래스의 CreateInputLayout() 함수를 다음과 같이 정의한다.

```
D3D12_INPUT_LAYOUT_DESC CObjectsShader::CreateInputLayout()
{
   UINT nInputElementDescs = 2;
   D3D12_INPUT_ELEMENT_DESC *pd3dInputElementDescs = new
D3D12_INPUT_ELEMENT_DESC[nInputElementDescs];

   pd3dInputElementDescs[0] ={ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 };
   pd3dInputElementDescs[1] ={ "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 };

   D3D12_INPUT_LAYOUT_DESC d3dInputLayoutDesc;
   d3dInputLayoutDesc.pInputElementDescs = pd3dInputElementDescs;
   d3dInputLayoutDesc.NumElements = nInputElementDescs;
```

```
   return(d3dInputLayoutDesc);
}
```

❸ “CObjectsShader” 클래스의 CreateVertexShader() 함수와 CreatePixelShader() 함수를 다음과 같이 정의한다.

```
D3D12_SHADER_BYTECODE CObjectsShader::CreateVertexShader(ID3DBlob **ppd3dShaderBlob)
{
   return(CShader::CompileShaderFromFile(L"Shaders.hlsl", "VSDiffused", "vs_5_1",
ppd3dShaderBlob));
}

D3D12_SHADER_BYTECODE CObjectsShader::CreatePixelShader(ID3DBlob **ppd3dShaderBlob)
{
   return(CShader::CompileShaderFromFile(L"Shaders.hlsl", "PSDiffused", "ps_5_1",
ppd3dShaderBlob));
}
```

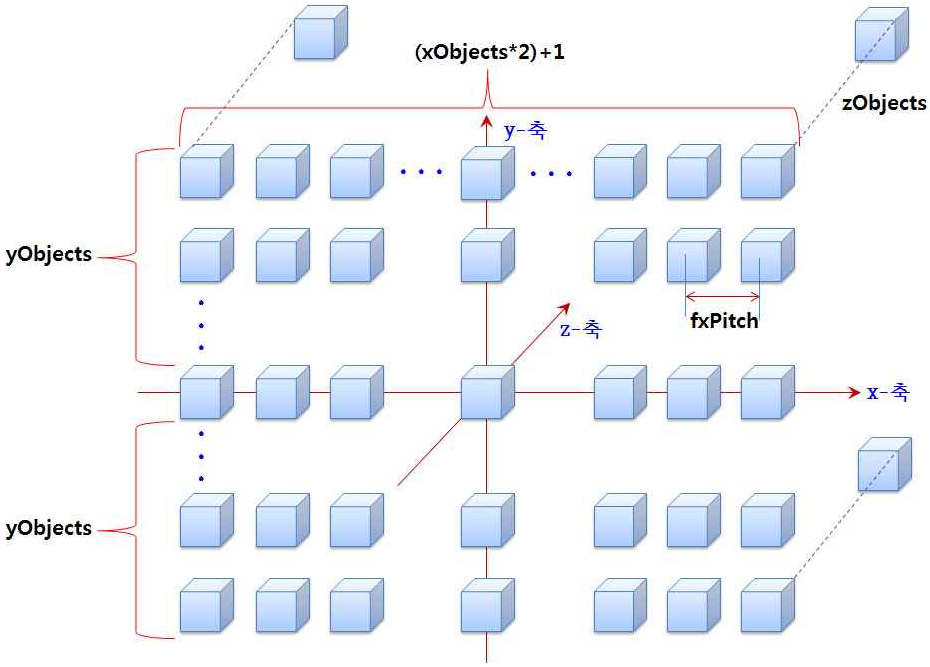
❹ “CObjectsShader” 클래스의 CreateShader() 함수를 다음과 같이 정의한다.

```
void CObjectsShader::CreateShader(ID3D12Device *pd3dDevice, ID3D12RootSignature
*pd3dGraphicsRootSignature)
{
   m_nPipelineStates = 1;
   m_ppd3dPipelineStates = new ID3D12PipelineState*[m_nPipelineStates];

   CShader::CreateShader(pd3dDevice, pd3dGraphicsRootSignature);
}
```

❺ “CObjectsShader” 클래스의 BuildObjects() 함수를 다음과 같이 정의한다.

여러 개의 정육면체 모양의 게임 객체를 생성하여 다음 그림과 같은 형태로 배치할 것이다.




정육면체 객체들의 배치

```
void CObjectsShader::BuildObjects(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList)
{
//가로x세로x높이가 12x12x12인 정육면체 메쉬를 생성한다.
   CCubeMeshDiffused *pCubeMesh = new CCubeMeshDiffused(pd3dDevice, pd3dCommandList,
12.0f, 12.0f, 12.0f);

/*x-축, y-축, z-축 양의 방향의 객체 개수이다. 각 값을 1씩 늘리거나 줄이면서 실행할 때 프레임 레이트가 어떻게
변하는 가를 살펴보기 바란다.*/
   int xObjects = 10, yObjects = 10, zObjects = 10, i = 0;

//x-축, y-축, z-축으로 21개씩 총 21 x 21 x 21 = 9261개의 정육면체를 생성하고 배치한다.
   m_nObjects = (xObjects * 2 + 1) * (yObjects * 2 + 1) * (zObjects * 2 + 1);

   m_ppObjects = new CGameObject*[m_nObjects];

   float fxPitch = 12.0f * 2.5f;
   float fyPitch = 12.0f * 2.5f;
   float fzPitch = 12.0f * 2.5f;

   CRotatingObject *pRotatingObject = NULL;
   for (int x = -xObjects; x <= xObjects; x++)
   {
      for (int y = -yObjects; y <= yObjects; y++)
      {
         for (int z = -zObjects; z <= zObjects; z++)
         {
            pRotatingObject = new CRotatingObject();
            pRotatingObject->SetMesh(pCubeMesh);
//각 정육면체 객체의 위치를 설정한다.
            pRotatingObject->SetPosition(fxPitch*x, fyPitch*y, fzPitch*z);
```

```
            pRotatingObject->SetRotationAxis(XMFLOAT3(0.0f, 1.0f, 0.0f));
            pRotatingObject->SetRotationSpeed(10.0f*(i % 10)+3.0f);
            m_ppObjects[i++] = pRotatingObject;
         }
      }
   }

   CreateShaderVariables(pd3dDevice, pd3dCommandList);
}
```

❻ “CObjectsShader” 클래스의 ReleaseObjects() 함수를 다음과 같이 정의한다.

```
void CObjectsShader::ReleaseObjects()
{
   if (m_ppObjects)
   {
      for (int j = 0; j < m_nObjects; j++)
      {
         if (m_ppObjects[j]) delete m_ppObjects[j];
      }
      delete[] m_ppObjects;
   }
}
```

❼ “CObjectsShader” 클래스의 AnimateObjects() 함수를 다음과 같이 정의한다.

```
void CObjectsShader::AnimateObjects(float fTimeElapsed)
{
   for (int j = 0; j < m_nObjects; j++)
   {
      m_ppObjects[j]->Animate(fTimeElapsed);
   }
}
```

❽ “CObjectsShader” 클래스의 ReleaseUploadBuffers() 함수를 다음과 같이 정의한다.

```
void CObjectsShader::ReleaseUploadBuffers()
{
   if (m_ppObjects)
   {
      for (int j = 0; j < m_nObjects; j++) m_ppObjects[j]->ReleaseUploadBuffers();
   }
}
```

❾ “CObjectsShader” 클래스의 Render() 함수를 다음과 같이 정의한다.

```
void CObjectsShader::Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera)
{
   CShader::Render(pd3dCommandList, pCamera);
```

```
    for (int j = 0; j < m_nObjects; j++)
    {
        if (m_ppObjects[j])
        {
            m_ppObjects[j]->Render(pd3dCommandList, pCamera);
        }
    }
}
```

⑦ “Scene.h” 파일 수정하기

❶ “Scene.h” 파일의 앞부분에 다음을 추가한다.

```
#include "Shader.h“
```

❷ “CScene” 클래스에서 다음 멤버변수를 삭제한다.

```
protected:
    CGameObject **m_ppObjects = NULL;
    int m_nObjects = 0;
```

❸ “CScene” 클래스에 다음 멤버변수를 선언한다.

```
protected:
//배치(Batch) 처리를 하기 위하여 씬을 셰이더들의 리스트로 표현한다.
    CObjectsShader *m_pShaders = NULL;
    int m_nShaders = 0;
```

⑧ “Scene.cpp” 파일 수정하기

❶ “CScene” 클래스의 BuildObjects() 함수를 다음과 같이 수정한다.

```
void CScene::BuildObjects(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList)
{
    m_pd3dGraphicsRootSignature = CreateGraphicsRootSignature(pd3dDevice);

    m_nShaders = 1;
    m_pShaders = new CObjectsShader[m_nShaders];
    m_pShaders[0].CreateShader(pd3dDevice, m_pd3dGraphicsRootSignature);
    m_pShaders[0].BuildObjects(pd3dDevice, pd3dCommandList);
}
```

❷ “CScene” 클래스의 ReleaseObjects(), ReleaseUploadBuffers(), AnimateObjects() 함수를 다음과 같이 수정한다.

```
void CScene::ReleaseObjects()
{
```

```
    if (m_pd3dGraphicsRootSignature) m_pd3dGraphicsRootSignature->Release();

    for (int i = 0; i < m_nShaders; i++)
    {
        m_pShaders[i].ReleaseShaderVariables();
        m_pShaders[i].ReleaseObjects();
    }
    if (m_pShaders) delete[] m_pShaders;
}

void CScene::ReleaseUploadBuffers()
{
    for (int i = 0; i < m_nShaders; i++) m_pShaders[i].ReleaseUploadBuffers();
}

void CScene::AnimateObjects(float fTimeElapsed)
{
    for (int i = 0; i < m_nShaders; i++)
    {
        m_pShaders[i].AnimateObjects(fTimeElapsed);
    }
}
```

❷ “CScene” 클래스의 Render() 함수를 다음과 같이 수정한다.

```
void CScene::Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera)
{
    pCamera->SetViewportsAndScissorRects(pd3dCommandList);
    pd3dCommandList->SetGraphicsRootSignature(m_pd3dGraphicsRootSignature);
    pCamera->UpdateShaderVariables(pd3dCommandList);

    for (int i = 0; i < m_nShaders; i++)
    {
        m_pShaders[i].Render(pd3dCommandList, pCamera);
    }
}
```

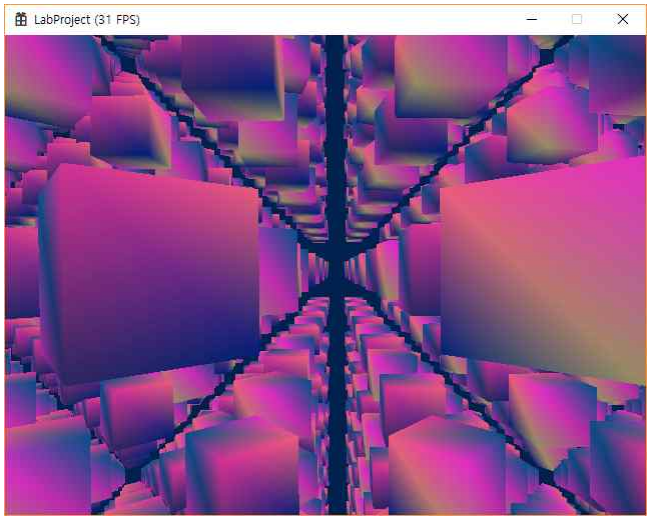
⑨ “GameFramework.cpp” 파일 변경하기

❶ “CGameFramework” 클래스의 BuildObjects() 함수에서 카메라의 위치를 다음과 같이 변경한다.

```
    m_pCamera->GenerateViewMatrix(XMFLOAT3(0.0f, 0.0f, -50.0f), XMFLOAT3(0.0f, 0.0f,
0.0f), XMFLOAT3(0.0f, 1.0f, 0.0f));
```

⑩ 프로젝트 빌드하여 실행하기

❶ 프로젝트를 빌드하여 실행

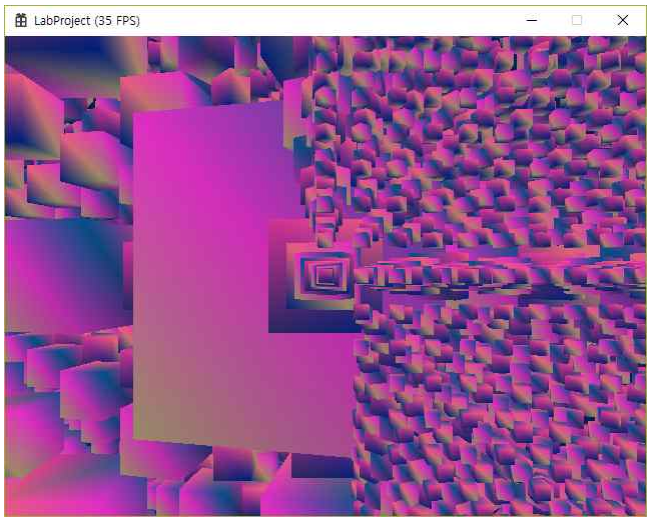



정상적인 렌더링 결과

❷ 깊이-스텐실 상태를 변경하기

“CShader” 클래스의 CreateDepthStencilState() 멤버 함수에서 깊이-스텐실 상태의 깊이-검사(Depth Test)를 하지 않도록 다음과 같이 변경하여 빌드하고 실행하여 깊이-검사를 하여 렌더링이 되는 결과와 비교해 본다.

```
//깊이-검사를 하지 않으므로 여러 개의 객체들이 겹쳐지는 것처럼 그려진다.
   d3dDepthStencilDesc.DepthEnable = FALSE;
```




깊이-검사를 하지 않은 렌더링

프로젝트를 빌드하여 실행하면 위의 그림과 같이 객체들이 겹쳐지거나 비정상적으로 렌더링된다. 이것은 객체들을 생성하여 m_ppObjects[] 배열에 저장할 때의 다음과 같이 순서가 z-축의 좌표가 작은 것부터 저장되기 때문이다.

```
   for (int x = -xObjects; x <= xObjects; x++)
   {
```

```
        for (int y = -yObjects; y <= yObjects; y++)
        {
            for (int z = -zObjects; z <= zObjects; z++)
            {
                ...
                m_ppObjects[i++] = pRotatingObject;
            }
        }
    }
```

그리고 렌더링을 다음과 같이 하면 z-축의 좌표가 작은 것부터 렌더링이 될 것이다.

```
    for (int j = 0; j < m_nObjects; j++)
    {
        if (m_ppObjects[j]) m_ppObjects[j]->Render(pd3dCommandList, pCamera);
    }
```

카메라에서 보았을 때 그려지는 게임 객체들 사이에 가려지는 관계를 따지지 않고 모두 렌더링하고 있기 때문에 모든 게임 객체들이 렌더링되는 순서에 따라 렌더 타겟에 그려진다. 깊이-검사를 사용하지 않고 정상적으로 렌더링을 하려면 "CScene" 클래스의 BuildObjects() 함수에서 게임 객체들을 생성하는 순서를 다음과 같이 수정하면 될 것이다. 이 경우 게임 객체들이 m_ppObjects[] 배열에 저장될 때의 z-축의 좌표가 큰 것부터 저장되고 저장된 순서대로 렌더링을 하면 멀리 있는 게임 객체부터 렌더링이 될 것이다.

```
    for (int i = 0, z = +zObjects; z >= -zObjects; z--)
    {
        for (int y = -yObjects; y <= yObjects; y++)
        {
            for (int x = -xObjects; x <= xObjects; x++)
            {
                ...
```