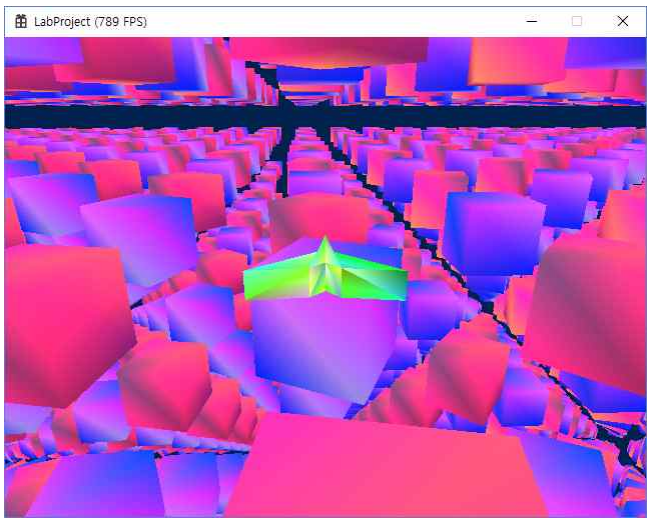
□ 예제 프로그램 13: LabProject12(인스턴싱: Instancing)

예제 프로그램 LabProject11을 기반으로 여러 개의 직육면체들을 인스턴싱으로 렌더링하도록 구현한다. 인스턴싱은 입력 레이아웃과 정점 버퍼를 사용한다.



① 새로운 프로젝트의 생성

먼저 새로운 프로젝트 LabProject12를 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 “LabProject12”를 입력하고 『확인』을 선택한다.

❶ 파일 탐색기에서 프로젝트 “LabProject11” 폴더의 다음 파일을 선택하여 프로젝트 “LabProject12” 폴더에 복사한다.

- GameFramework.h
- GameFramework.cpp
- Mesh.h
- Mesh.cpp
- Camera.h
- Camera.cpp
- Player.h
- Player.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 “LabProject12”에 추가한다. 오른쪽 마우스 버튼으로 『LabProject12』을 선택하고 『추가』, 『기존 항목』를 차례로 선택한다. 그러면 “기존 항목 추가” 대화 상자가 나타난다. 위의 파일들을 마우스로 선택(Ctrl+선택)하여 『추가』를 누르면 선택된 파일들이 프로젝트 “LabProject12”에 추가된다.

② LabProject12.cpp 파일 수정하기

이제 “LabProject12.cpp" 파일의 내용을 “LabProject11.cpp" 파일의 내용으로 바꾸도록 하자. “LabProject11.cpp" 파일의 내용 전체를 “LabProject12.cpp" 파일로 복사한다. 이제 “LabProject11.cpp" 파일에서 “LabProject11”을 “LabProject12”로 모두 바꾼다.
그리고 “LABPROJECT11”을 “LABPROJECT12”로 모두 바꾼다.

③ “Mesh.h” 파일 수정하기

“Mesh.h” 파일을 다음과 같이 수정한다.

❶ “CMesh” 클래스에 인스턴싱을 위한 Render() 함수를 다음과 같이 선언한다.

```
   virtual void Render(ID3D12GraphicsCommandList *pd3dCommandList, UINT nInstances);
   virtual void Render(ID3D12GraphicsCommandList *pd3dCommandList, UINT nInstances,
D3D12_VERTEX_BUFFER_VIEW d3dInstancingBufferView);
```

④ “Mesh.cpp” 파일 수정하기

“Mesh.cpp” 파일을 다음과 같이 수정한다.

❶ “CMesh” 클래스의 Render() 함수를 다음과 같이 수정한다.

```
void CMesh::Render(ID3D12GraphicsCommandList *pd3dCommandList)
{
   pd3dCommandList->IASetVertexBuffers(m_nSlot, 1, &m_d3dVertexBufferView);
   Render(pd3dCommandList, 1);
}
```

❷ “CMesh” 클래스의 인스턴싱을 위한 Render() 함수를 다음과 같이 정의한다.

```
void CMesh::Render(ID3D12GraphicsCommandList *pd3dCommandList, UINT nInstances)
{
   pd3dCommandList->IASetPrimitiveTopology(m_d3dPrimitiveTopology);
   if (m_pd3dIndexBuffer)
   {
      pd3dCommandList->IASetIndexBuffer(&m_d3dIndexBufferView);
      pd3dCommandList->DrawIndexedInstanced(m_nIndices, nInstances, 0, 0, 0);
   }
   else
   {
      pd3dCommandList->DrawInstanced(m_nVertices, nInstances, m_nOffset, 0);
   }
}

void CMesh::Render(ID3D12GraphicsCommandList *pd3dCommandList, UINT nInstances,
D3D12_VERTEX_BUFFER_VIEW d3dInstancingBufferView)
{
//정점 버퍼 뷰와 인스턴싱 버퍼 뷰를 입력-조립 단계에 설정한다.
   D3D12_VERTEX_BUFFER_VIEW pVertexBufferViews[] = { m_d3dVertexBufferView,
d3dInstancingBufferView };
   pd3dCommandList->IASetVertexBuffers(m_nSlot, _countof(pVertexBufferViews),
pVertexBufferViews);
   Render(pd3dCommandList, nInstances);
}
```

⑤ “GameObject.h” 파일 변경하기

❶ “CGameObject” 클래스에 다음 Render() 함수를 선언한다.

```
   virtual void Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera, UINT
nInstances, D3D12_VERTEX_BUFFER_VIEW d3dInstancingBufferView);
```

⑥ “GameObject.cpp” 파일 변경하기

❶ “CGameObject” 클래스의 다음 Render() 함수를 정의한다.

```
//인스턴싱 정점 버퍼 뷰를 사용하여 메쉬를 렌더링한다.
void CGameObject::Render(ID3D12GraphicsCommandList *pd3dCommandList, CCamera *pCamera,
UINT nInstances, D3D12_VERTEX_BUFFER_VIEW d3dInstancingBufferView)
{
   OnPrepareRender();

   if (m_pMesh) m_pMesh->Render(pd3dCommandList, nInstances, d3dInstancingBufferView);
}
```

⑦ “Shader.h” 파일 변경하기

❶ “Shader.h” 파일에 다음 구조체를 추가한다.

```
//인스턴스 정보(게임 객체의 월드 변환 행렬과 객체의 색상)를 위한 구조체이다.
struct VS_VB_INSTANCE
{
   XMFLOAT4X4      m_xmf4x4Transform;
   XMFLOAT4        m_xmcColor;
};
```

❷ “CInstancingShader” 클래스를 다음과 같이 선언한다.

```
class CInstancingShader : public CObjectsShader
{
public:
   CInstancingShader();
   virtual ~CInstancingShader();

   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 CreateShaderVariables(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList);
   virtual void UpdateShaderVariables(ID3D12GraphicsCommandList *pd3dCommandList);
   virtual void ReleaseShaderVariables();

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

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

protected:
//인스턴스 정점 버퍼와 정점 버퍼 뷰이다.
   ID3D12Resource                  *m_pd3dcbGameObjects = NULL;
   VS_VB_INSTANCE                  *m_pcbMappedGameObjects = NULL;

   D3D12_VERTEX_BUFFER_VIEW        m_d3dInstancingBufferView;
};
```

⑧ “Shader.cpp” 파일 변경하기

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

```
void CObjectsShader::BuildObjects(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList)
{
}
```

❷ “CInstancingShader” 클래스의 생성자와 소멸자를 다음과 같이 정의한다.

```
CInstancingShader::CInstancingShader()
{
}

CInstancingShader::~CInstancingShader()
{
}
```

❸ “CInstancingShader” 클래스의 CreateInputLayout() 함수를 다음과 같이 정의한다.

```
D3D12_INPUT_LAYOUT_DESC CInstancingShader::CreateInputLayout()
{
   UINT nInputElementDescs = 7;
   D3D12_INPUT_ELEMENT_DESC *pd3dInputElementDescs = new
D3D12_INPUT_ELEMENT_DESC[nInputElementDescs];

//정점 정보를 위한 입력 원소이다.
   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 };
//인스턴싱 정보를 위한 입력 원소이다.
   pd3dInputElementDescs[2] = { "WORLDMATRIX", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0,
D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA, 1 };
   pd3dInputElementDescs[3] = { "WORLDMATRIX", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16,
D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA, 1 };
   pd3dInputElementDescs[4] = { "WORLDMATRIX", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32,
D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA, 1 };
   pd3dInputElementDescs[5] = { "WORLDMATRIX", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48,
D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA, 1 };
   pd3dInputElementDescs[6] = { "INSTANCECOLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1,
64, D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA, 1 };

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

   return(d3dInputLayoutDesc);
}
```

❹ “CInstancingShader” 클래스의 CreateVertexShader(), CreatePixelShader(), CreateShader() 함수를 다음과 같이 정의한다.

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

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

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

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

❺ “CInstancingShader” 클래스의 CreateShaderVariables()와 ReleaseShaderVariables() 함수를 다음과 같이 정의한다.

```
void CInstancingShader::CreateShaderVariables(ID3D12Device *pd3dDevice,
ID3D12GraphicsCommandList *pd3dCommandList)
{
//인스턴스 정보를 저장할 정점 버퍼를 업로드 힙 유형으로 생성한다.
   m_pd3dcbGameObjects = ::CreateBufferResource(pd3dDevice, pd3dCommandList, NULL,
sizeof(VS_VB_INSTANCE) * m_nObjects, D3D12_HEAP_TYPE_UPLOAD,
D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER, NULL);

//정점 버퍼(업로드 힙)에 대한 포인터를 저장한다.
   m_pd3dcbGameObjects->Map(0, NULL, (void **)&m_pcbMappedGameObjects);

//정점 버퍼에 대한 뷰를 생성한다.
   m_d3dInstancingBufferView.BufferLocation =
m_pd3dcbGameObjects->GetGPUVirtualAddress();
   m_d3dInstancingBufferView.StrideInBytes = sizeof(VS_VB_INSTANCE);
   m_d3dInstancingBufferView.SizeInBytes = sizeof(VS_VB_INSTANCE) * m_nObjects;
}

void CInstancingShader::ReleaseShaderVariables()
{
   if (m_pd3dcbGameObjects) m_pd3dcbGameObjects->Unmap(0, NULL);
   if (m_pd3dcbGameObjects) m_pd3dcbGameObjects->Release();
}
```

❻ “CInstancingShader” 클래스의 UpdateShaderVariables() 함수를 다음과 같이 정의한다.

```
//인스턴싱 정보(객체의 월드 변환 행렬과 색상)를 정점 버퍼에 복사한다.
void CInstancingShader::UpdateShaderVariables(ID3D12GraphicsCommandList
*pd3dCommandList)
{
   for (int j = 0; j < m_nObjects; j++)
   {
      m_pcbMappedGameObjects[j].m_xmcColor = (j % 2) ? XMFLOAT4(0.5f, 0.0f, 0.0f, 0.0f) :
```

```
XMFLOAT4(0.0f, 0.0f, 0.5f, 0.0f);
      XMStoreFloat4x4(&m_pcbMappedGameObjects[j].m_xmf4x4Transform,
XMMatrixTranspose(XMLoadFloat4x4(&m_ppObjects[j]->m_xmf4x4World)));
   }
}
```

❼ “CInstancingShader” 클래스의 BuildObjects() 함수를 다음과 같이 정의한다.

```
void CInstancingShader::BuildObjects(ID3D12Device *pd3dDevice, ID3D12GraphicsCommandList
*pd3dCommandList)
{
   int xObjects = 10, yObjects = 10, zObjects = 10, i = 0;

   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->SetPosition(fxPitch*x, fyPitch*y, fzPitch*z);
            pRotatingObject->SetRotationAxis(XMFLOAT3(0.0f, 1.0f, 0.0f));
            pRotatingObject->SetRotationSpeed(10.0f*(i % 10));
            m_ppObjects[i++] = pRotatingObject;
         }
      }
   }

//인스턴싱을 사용하여 렌더링하기 위하여 하나의 게임 객체만 메쉬를 가진다.
   CCubeMeshDiffused *pCubeMesh = new CCubeMeshDiffused(pd3dDevice, pd3dCommandList,
12.0f, 12.0f, 12.0f);
   m_ppObjects[0]->SetMesh(pCubeMesh);

//인스턴싱을 위한 정점 버퍼와 뷰를 생성한다.
   CreateShaderVariables(pd3dDevice, pd3dCommandList);
}
```

❽ “CInstancingShader” 클래스의 Render() 함수를 다음과 같이 수정한다.

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

```
//모든 게임 객체의 인스턴싱 데이터를 버퍼에 저장한다.
   UpdateShaderVariables(pd3dCommandList);

//하나의 정점 데이터를 사용하여 모든 게임 객체(인스턴스)들을 렌더링한다.
   m_ppObjects[0]->Render(pd3dCommandList, pCamera, m_nObjects,
m_d3dInstancingBufferView);
}
```

⑨ “Scene.h” 파일 수정하기

❶ “CScene” 클래스에서 m_pShaders 멤버 변수의 선언에서 CObjectsShader를 CInstancingShader로 바꾼다.

```
   CObjectsShader *m_pShaders = NULL;

   CInstancingShader *m_pShaders = NULL;
```

⑩ “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 CInstancingShader[m_nShaders];
   m_pShaders[0].CreateShader(pd3dDevice, m_pd3dGraphicsRootSignature);
   m_pShaders[0].BuildObjects(pd3dDevice, pd3dCommandList);
}
```

⑪ “Shaders.hlsl” 파일 수정하기

“Shaders.hlsl” 파일에 다음을 추가한다.

```
//정점 데이터와 인스턴싱 데이터를 위한 구조체이다.
struct VS_INSTANCING_INPUT
{
   float3 position : POSITION;
   float4 color : COLOR;
   float4x4 mtxTransform : WORLDMATRIX;
   float4 instanceColor : INSTANCECOLOR;
};

struct VS_INSTANCING_OUTPUT
{
   float4 position : SV_POSITION;
   float4 color : COLOR;
};
```

```
VS_INSTANCING_OUTPUT VSInstancing(VS_INSTANCING_INPUT input)
{
   VS_INSTANCING_OUTPUT output;

   output.position = mul(mul(mul(float4(input.position, 1.0f), input.mtxTransform),
gmtxView), gmtxProjection);
   output.color = input.color + input.instanceColor;

   return(output);
}

float4 PSInstancing(VS_INSTANCING_OUTPUT input) : SV_TARGET
{
   return(input.color);
}
```