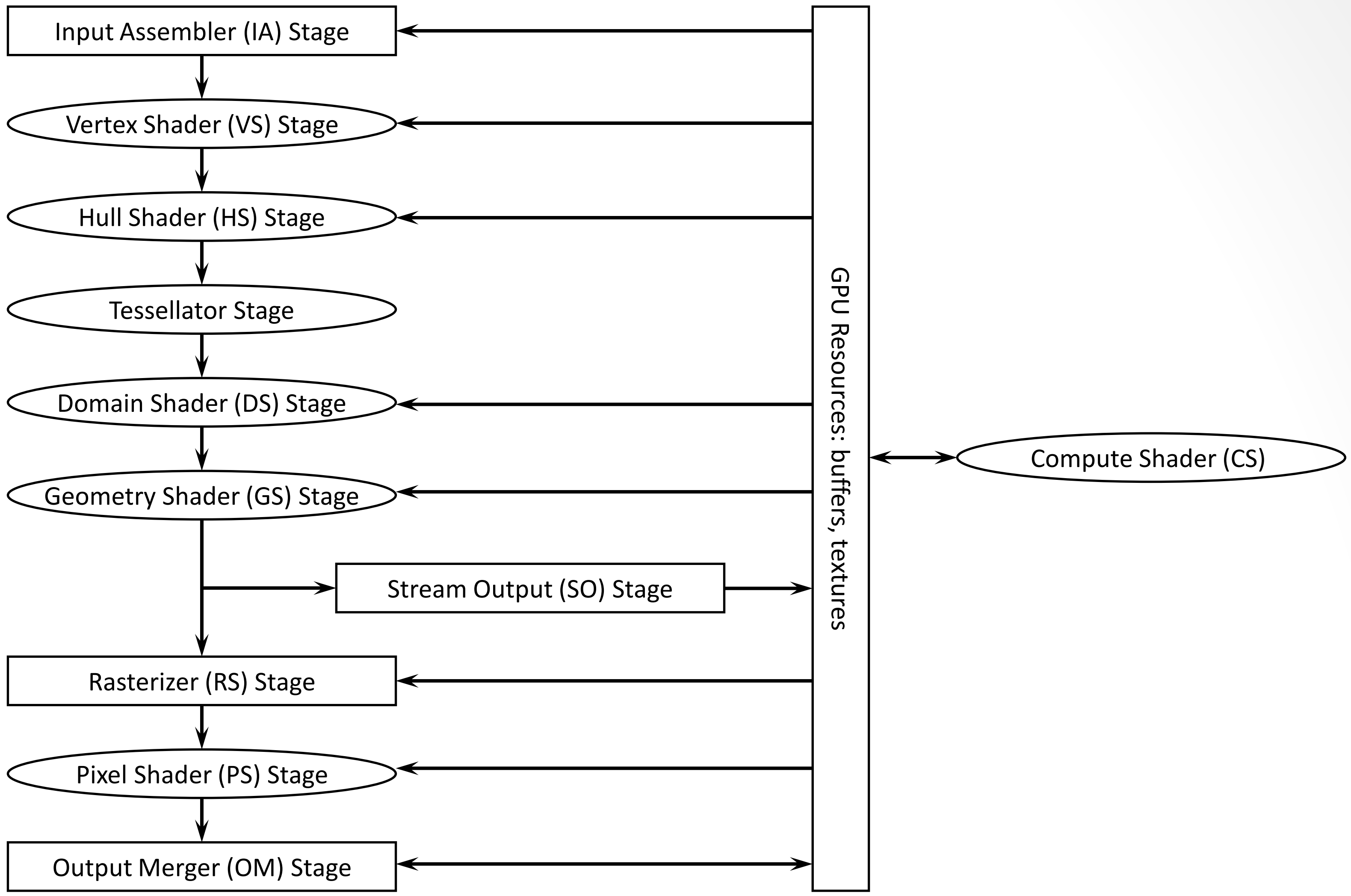
II. Direct3D Foundations
13. The Compute Shader

Game Graphic Programming
Kyung Hee University
Software Convergence
Prof. Daeho Lee

Compute Shader (1)



Compute Shader (2)

- The compute shader is not part of the rendering pipeline but sits off to the side.
- The compute shader can read and write to GPU resources.
- The compute shader can be mixed with graphics rendering, or used alone for GPGPU programming.

Threads & Thread Groups

- A thread group is executed on a single streaming multiprocessor.
  - GTX 1080: 20 SMs × two blocks of 64 cores = 20×2×64 = 2,560 cores
  - RTX 4090: 128 SMs = 128×128 = 16,384 cores (SM can contain up to 8 blocks)
  - A thread group consists of *n* threads.

  - In Direct3D, thread groups are launched via the following method call:
    -
      ```
      void ID3D12GraphicsCommandList::Dispatch(
       UINT ThreadGroupCountX,
       UINT ThreadGroupCountY,
       UINT ThreadGroupCountZ);
      ```
    - This enables you to launch a 3D grid of thread groups.

Kyung Hee University
nize@khu.ac.kr

The Components of CS (1)

- A compute shader consists of the following components:
  - 1. Global variable access via constant buffers.
  - 2. Input and output resources, which are discussed in the next section.
  - 3. The **`[numthreads(X, Y, Z)]`** attribute, which specifies the number of threads in the thread group as a 3D grid of threads.
  - 4. The shader body that has the instructions to execute for each thread.
  - 5. Thread identification system value parameters.
    - Each thread group is assigned an ID by the system: **`SV_GroupID`**.
    - Inside a thread group, each thread is given a unique ID relative to its group: **`SV_GroupThreadID`**.
    - A **`Dispatch`** call dispatches a grid of thread groups. The dispatch thread ID uniquely identifies a thread relative to all the threads generated by a **`Dispatch`** call: **`SV_DispatchThreadID`**.

The Components of CS (2)

```
// VecAdd.hlsl
struct Data
{
   float3 v1;
   float2 v2;
};

StructuredBuffer<Data> gInputA : register(t0);
StructuredBuffer<Data> gInputB : register(t1);
RWStructuredBuffer<Data> gOutput : register(u0);
   // unordered access view

[numthreads(32, 1, 1)]
void CS(int3 dtid : SV_DispatchThreadID)
{
   gOutput[dtid.x].v1 = gInputA[dtid.x].v1 + gInputB[dtid.x].v1;
   gOutput[dtid.x].v2 = gInputA[dtid.x].v2 + gInputB[dtid.x].v2;
}
```

The Components of CS (2)

```
// For textures

Texture2D gInputA;
Texture2D gInputB;
RWTexture2D<float4> gOutput;

[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
  gOutput[dispatchThreadID.xy] =
      gInputA[dispatchThreadID.xy] + gInputB[dispatchThreadID.xy];
}
```

Compile Shader & Compute PSO (1)

D3D12_COMPUTE_PIPELINE_STATE_DESC

```
typedef struct D3D12_COMPUTE_PIPELINE_STATE_DESC {
  ID3D12RootSignature          *pRootSignature;
  D3D12_SHADER_BYTECODE        CS;
  UINT                         NodeMask;
  D3D12_CACHED_PIPELINE_STATE CachedPSO;
  D3D12_PIPELINE_STATE_FLAGS  Flags;
} D3D12_COMPUTE_PIPELINE_STATE_DESC;
```

- This describes a compute pipeline state object.

Compile Shader & Compute PSO (1)

```
void VecAddCSApp::BuildShadersAndInputLayout() {
   mShaders["vecAddCS"]
      = d3dUtil::CompileShader(L"Shaders\\VecAdd.hlsl", nullptr,
         "CS", "cs_5_0");
}

void VecAddCSApp::BuildPSOs() {
   D3D12_COMPUTE_PIPELINE_STATE_DESC computePsoDesc = {};
   computePsoDesc.pRootSignature = mRootSignature.Get();
   computePsoDesc.CS = {
      reinterpret_cast<BYTE*>(mShaders["vecAddCS"]->
         GetBufferPointer()),
      mShaders["vecAddCS"]->GetBufferSize()
   };
   computePsoDesc.Flags = D3D12_PIPELINE_STATE_FLAG_NONE;
   ThrowIfFailed(md3dDevice->
      CreateComputePipelineState(&computePsoDesc,
      IID_PPV_ARGS(&mPSOs["vecAdd"])));
}
```

Data Input & Output Resources (1)

- Texture inputs
  - Inputs are bound to shader resource view (SRVs).
- Texture outputs and unordered access views (UAVs)
  - Outputs are treated special and have the special prefix to their type “**`RW`**,” which stands for read-write.
  - Inputs are bound to unordered access views (UAVs).

Data Input & Output Resources (2)

-
  ```
  void ID3D12Device::CreateUnorderedAccessView(
        ID3D12Resource                    *pResource,
        ID3D12Resource                    *pCounterResource,
        const D3D12_UNORDERED_ACCESS_VIEW_DESC *pDesc,
    [in] D3D12_CPU_DESCRIPTOR_HANDLE   DestDescriptor
  );
  ```

  -
    ```
    typedef struct D3D12_UNORDERED_ACCESS_VIEW_DESC {
      DXGI_FORMAT         Format;
      D3D12_UAV_DIMENSION ViewDimension;
      union {
        D3D12_BUFFER_UAV         Buffer;
        D3D12_TEX1D_UAV          Texture1D;
        D3D12_TEX1D_ARRAY_UAV    Texture1DArray;
        D3D12_TEX2D_UAV          Texture2D;
        D3D12_TEX2D_ARRAY_UAV    Texture2DArray;
        D3D12_TEX2DMS_UAV        Texture2DMS;
        D3D12_TEX2DMS_ARRAY_UAV Texture2DMSArray;
        D3D12_TEX3D_UAV          Texture3D;
      };
    } D3D12_UNORDERED_ACCESS_VIEW_DESC;
    ```

Data Input & Output Resources (3)

-
  ```
  HRESULT ID3D12Device::CreateCommittedResource(
    [in]      const D3D12_HEAP_PROPERTIES *pHeapProperties,
    [in]      D3D12_HEAP_FLAGS             HeapFlags,
    [in]      const D3D12_RESOURCE_DESC    *pDesc,
    [in]      D3D12_RESOURCE_STATES        InitialResourceState,
            // D3D12_RESOURCE_STATE_UNORDERED_ACCESS
    [in, optional]  const D3D12_CLEAR_VALUE
                                          *pOptimizedClearValue,
    [in]            REFIID                riidResource,
    [out, optional] void                  **ppvResource
  );
  ```

Data Input & Output Resources (4)

```
ComPtr<ID3D12Resource> mInputBufferA = nullptr;
ComPtr<ID3D12Resource> mInputUploadBufferA = nullptr;
ComPtr<ID3D12Resource> mInputBufferB = nullptr;
ComPtr<ID3D12Resource> mInputUploadBufferB = nullptr;
ComPtr<ID3D12Resource> mOutputBuffer = nullptr;
ComPtr<ID3D12Resource> mReadBackBuffer = nullptr;
const int NumDataElements = 32;

void VecAddCSApp::BuildBuffers() {
    std::ofstream fout("inputs.txt");
    std::vector<Data> dataA(NumDataElements);
    std::vector<Data> dataB(NumDataElements);
    for(int i = 0; i < NumDataElements; ++i) {
        dataA[i].v1 = XMFLOAT3(i, i, i);       dataA[i].v2 = XMFLOAT2(i, 0);
        dataB[i].v1 = XMFLOAT3(-i, i, 0.0f);  dataB[i].v2 = XMFLOAT2(0, -i);

        fout << "A(" << dataA[i].v1.x << ", " << dataA[i].v1.y << ", "
            << dataA[i].v1.z << ", " << dataA[i].v2.x << ", " << dataA[i].v2.y
            << ")" << "\t";
        fout << "B(" << dataB[i].v1.x << ", " << dataB[i].v1.y << ", "
            << dataB[i].v1.z << ", " << dataB[i].v2.x << ", " << dataB[i].v2.y
            << ")" << std::endl;
    }
```

Data Input & Output Resources (5)

```
UINT64 byteSize = dataA.size()*sizeof(Data);

// Create some buffers to be used as SRVs.
mInputBufferA = d3dUtil::CreateDefaultBuffer(
    md3dDevice.Get(),
    mCommandList.Get(),
    dataA.data(),
    byteSize,
    mInputUploadBufferA);

mInputBufferB = d3dUtil::CreateDefaultBuffer(
    md3dDevice.Get(),
    mCommandList.Get(),
    dataB.data(),
    byteSize,
    mInputUploadBufferB);
```

Data Input & Output Resources (6)

```
    // Create the buffer that will be a UAV.
    ThrowIfFailed(md3dDevice->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(byteSize,
            D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS),
        D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
        nullptr,
        IID_PPV_ARGS(&mOutputBuffer)));

    ThrowIfFailed(md3dDevice->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_READBACK),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
        D3D12_RESOURCE_STATE_COPY_DEST,
        nullptr,
        IID_PPV_ARGS(&mReadBackBuffer)));
}
```

Root Signatures

```
void VecAddCSApp::BuildRootSignature() {
   CD3DX12_ROOT_PARAMETER slotRootParameter[3];

   slotRootParameter[0].InitAsShaderResourceView(0);
   slotRootParameter[1].InitAsShaderResourceView(1);
   slotRootParameter[2].InitAsUnorderedAccessView(0);

   CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3, slotRootParameter,
      0, nullptr, D3D12_ROOT_SIGNATURE_FLAG_NONE);

   ComPtr<ID3DBlob> serializedRootSig = nullptr;
   ComPtr<ID3DBlob> errorBlob = nullptr;
   HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
      D3D_ROOT_SIGNATURE_VERSION_1,
      serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());
    if(errorBlob != nullptr) { /* … */ }    ThrowIfFailed(hr);

    ThrowIfFailed(md3dDevice->CreateRootSignature(
      0, serializedRootSig->GetBufferPointer(),
      serializedRootSig->GetBufferSize(),
      IID_PPV_ARGS(mRootSignature.GetAddressOf())));
}
```

Working CS (1)

```
void VecAddCSApp::DoComputeWork() {
   ThrowIfFailed(mDirectCmdListAlloc->Reset());

   ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(),
      mPSOs["vecAdd"].Get()));

   mCommandList->SetComputeRootSignature(mRootSignature.Get());

   mCommandList->SetComputeRootShaderResourceView(0,
      mInputBufferA->GetGPUVirtualAddress());
   mCommandList->SetComputeRootShaderResourceView(1,
      mInputBufferB->GetGPUVirtualAddress());
   mCommandList->SetComputeRootUnorderedAccessView(2,
      mOutputBuffer->GetGPUVirtualAddress());

   mCommandList->Dispatch(1, 1, 1);   // [numthreads(32, 1, 1)]
// mCommandList->Dispatch(2, 1, 1);   // [numthreads(16, 1, 1)]
```

Kyung Hee University
nize@khu.ac.kr

Working CS (2)

```
   mCommandList->ResourceBarrier(1,
      &CD3DX12_RESOURCE_BARRIER::Transition(mOutputBuffer.Get(),
     D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_COPY_SOURCE));

   mCommandList->CopyResource(mReadBackBuffer.Get(),
   mOutputBuffer.Get());

   mCommandList->ResourceBarrier(1,
     &CD3DX12_RESOURCE_BARRIER::Transition(mOutputBuffer.Get(),
     D3D12_RESOURCE_STATE_COPY_SOURCE, D3D12_RESOURCE_STATE_COMMON));

   ThrowIfFailed(mCommandList->Close());

   ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
  mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

   FlushCommandQueue();
```

Working CS (3)

```
    // Map the data so we can read it on CPU.
    Data* mappedData = nullptr;
    ThrowIfFailed(mReadBackBuffer->Map(0, nullptr,
        reinterpret_cast<void**>(&mappedData)));

    std::ofstream fout("results.txt");
    for(int i = 0; i < NumDataElements; ++i) {
        fout << "(" << mappedData[i].v1.x << ", "
            << mappedData[i].v1.y << ", " << mappedData[i].v1.z
            << ", " << mappedData[i].v2.x << ", " << mappedData[i].v2.y
            << ")" << std::endl;
    }

    mReadBackBuffer->Unmap(0, nullptr);
}
```

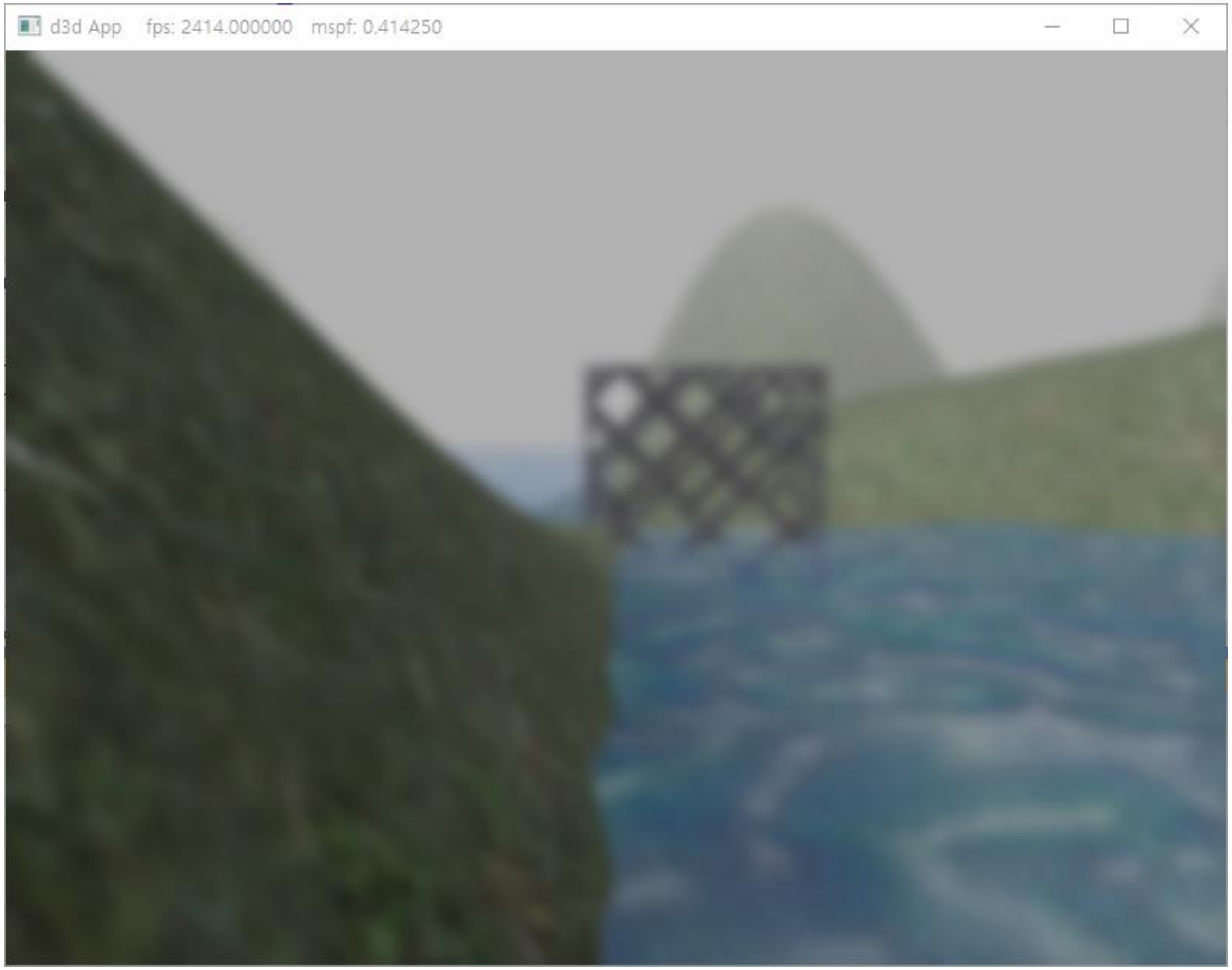
Working CS (4)

```
// inputs.txt
A(0, 0, 0, 0)  B(0, 0, 0, 0, 0)
A(1, 1, 1, 1, 0)  B(-1, 1, 0, 0, -1)
A(2, 2, 2, 2, 0)  B(-2, 2, 0, 0, -2)
A(3, 3, 3, 3, 0)  B(-3, 3, 0, 0, -3)
A(4, 4, 4, 4, 0)  B(-4, 4, 0, 0, -4)
A(5, 5, 5, 5, 0)  B(-5, 5, 0, 0, -5)
A(6, 6, 6, 6, 0)  B(-6, 6, 0, 0, -6)
A(7, 7, 7, 7, 0)  B(-7, 7, 0, 0, -7)
A(8, 8, 8, 8, 0)  B(-8, 8, 0, 0, -8)
A(9, 9, 9, 9, 0)  B(-9, 9, 0, 0, -9)
// …
A(26, 26, 26, 26, 0)   B(-26, 26, 0, 0, -26)
A(27, 27, 27, 27, 0)   B(-27, 27, 0, 0, -27)
A(28, 28, 28, 28, 0)   B(-28, 28, 0, 0, -28)
A(29, 29, 29, 29, 0)   B(-29, 29, 0, 0, -29)
A(30, 30, 30, 30, 0)   B(-30, 30, 0, 0, -30)
A(31, 31, 31, 31, 0)   B(-31, 31, 0, 0, -31)
```

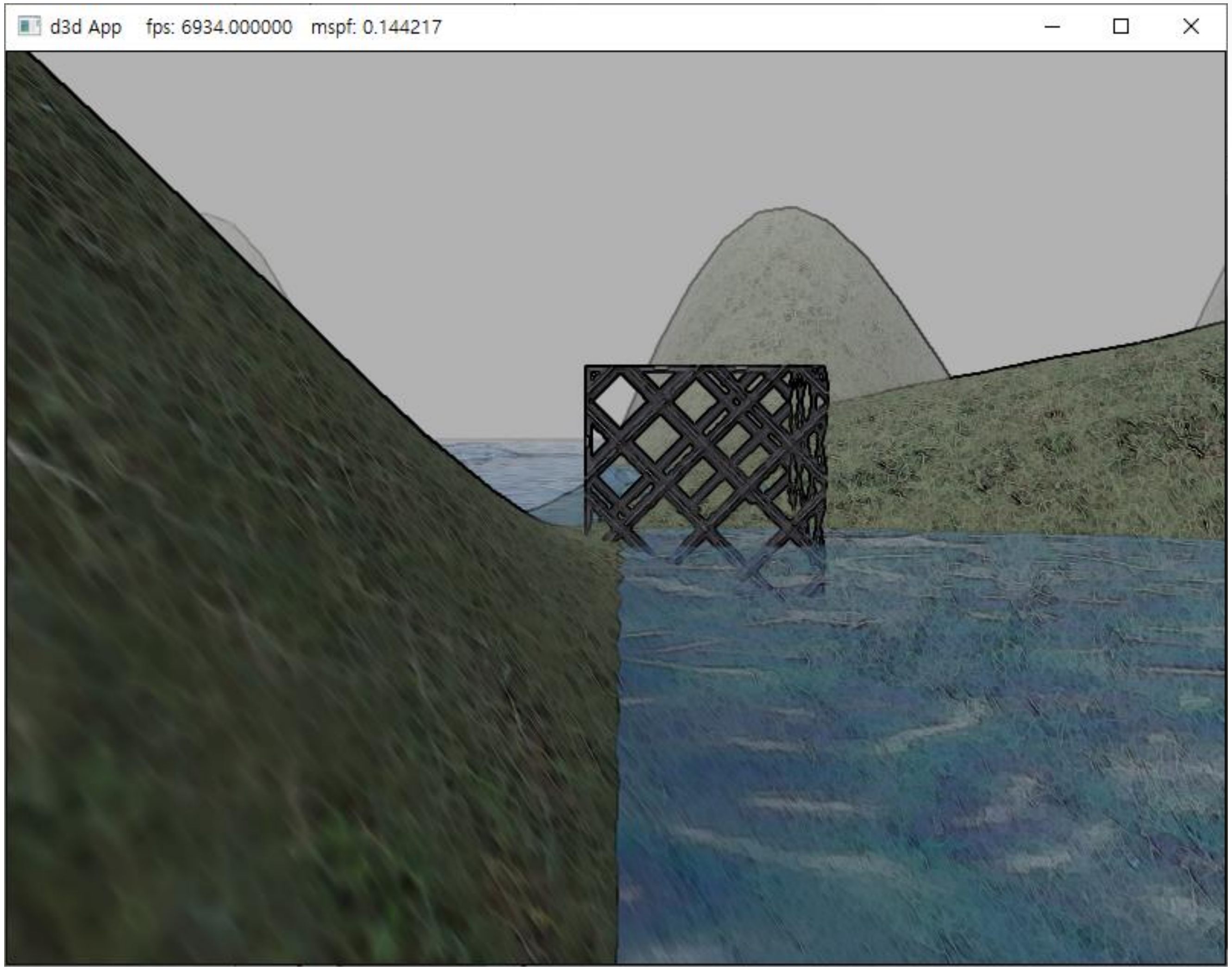
Working CS (5)

```
// results.txt
(0, 0, 0, 0, 0)
(0, 2, 1, 1, -1)
(0, 4, 2, 2, -2)
(0, 6, 3, 3, -3)
(0, 8, 4, 4, -4)
(0, 10, 5, 5, -5)
(0, 12, 6, 6, -6)
(0, 14, 7, 7, -7)
(0, 16, 8, 8, -8)
(0, 18, 9, 9, -9)
// …
(0, 52, 26, 26, -26)
(0, 54, 27, 27, -27)
(0, 56, 28, 28, -28)
(0, 58, 29, 29, -29)
(0, 60, 30, 30, -30)
(0, 62, 31, 31, -31)
```

Blur Example



Sobel Example


d3d App fps: 6934.000000 mspf: 0.144217