Multithreaded Rendering

+Dynamic Instancing

목차

시작하며...

이 ppt는 Multithreaded Rendering과 덤으로 Dynamic Instancing에 대해서 다룹니다.

이 프로젝트는 UE4의 코드와 다음 동영상에서 영감을 많이 받았으며 동영상은 한글 자막도 제공하므로 한번 보시는 걸 추천합니다.

https://youtu.be/qx1c190aGhs

Multithreaded Rendering 왜 해야 할까?

현시대에서 코어가 하나만 달린 CPU는 찾아보기 힘들게 되었습니다. CPU의 발전 방향이 코어의 개수를 늘리는 것으로 바뀐 이후로 고성능의 프로그램의 경우 스레드를 사용하는 것이 필수가 되었습니다.

이는 렌더링도 예외가 아닙니다. '그런데 렌더링은 GPU의 성능에 영향을 받는 것이 아닌가?' 라고 생각하실 수 있을 것 같은데요.

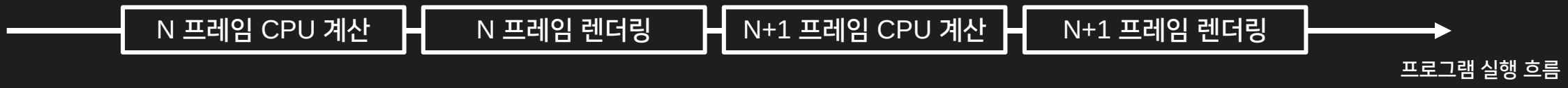
여기서는 스레드를 사용하지 않은 경우 왜 사용률이 떨어지는지를 살펴보겠습니다.

Multithreaded Rendering 왜 해야 할까?

Direct3D11 이전의 그래픽 API는 단일 스레드에서 실행되는 것에 중점을 두고 설계되었습니다.

암시적인 동기화 지점이 존재해서 API를 호출하고 난 다음 GPU의 처리가 완료되었을 때 이후의 코드가 실행됩니다.

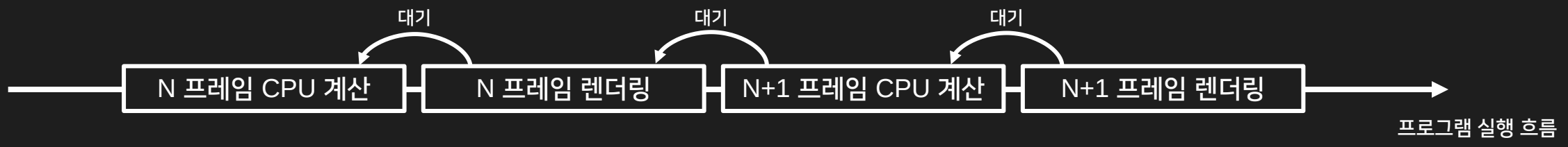
다음은 단일 스레드에서 동작하도록 작성된 프로그램의 실행 흐름을 간략하게 나타낸 것입니다.



Multithreaded Rendering 왜 해야 할까?

문제는 이렇게 순서를 맞춰서 진행하는 방식이 CPU와 GPU를 최대로 사용하지 못한다는 점입니다.

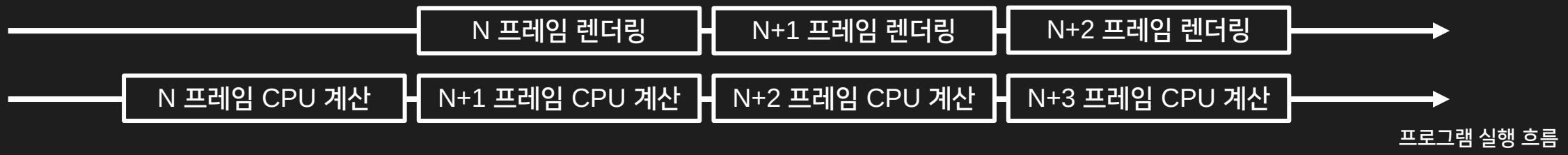
CPU 계산 중에 GPU는 제출된 명령이 없기 때문에 아무런 일도 하지 않고 CPU의 처리를 기다리게 되고 렌더링 중에는 동기화 지점으로 인해서 CPU는 GPU의 처리가 끝날 때 까지 기다리게 됩니다.



Multithreaded Rendering 왜 해야 할까?

따라서 이렇게 CPU와 GPU가 비효율적으로 사용되지 않도록 스레드를 분리하여 렌더링과 CPU 계산이 동시에 이뤄지도록 해야 합니다.(※)

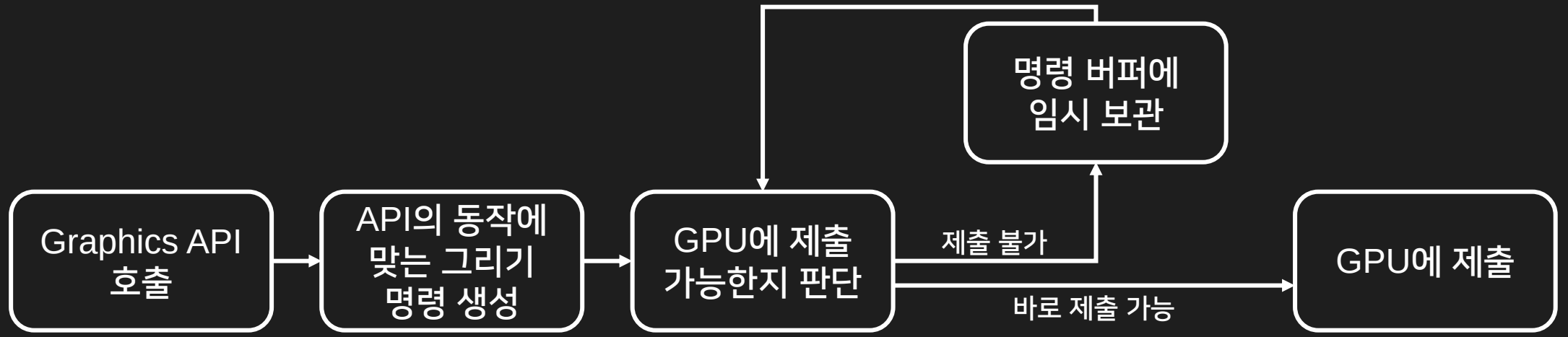
그런데 이렇게 스레드 하나를 추가하는 정도로는 멀티 스레드라고 부르기에는 무리가 있어 보이는데 아직 여러 스레드를 사용하여 성능을 개선할 수 있는 부분이 있습니다.



※ 물론 Direct3D12, Vulkan과 같은 최신 Graphics API의 경우에는 비동기가 기본이 되었기에 상황이 다릅니다.

Multithreaded Rendering 왜 해야 할까?

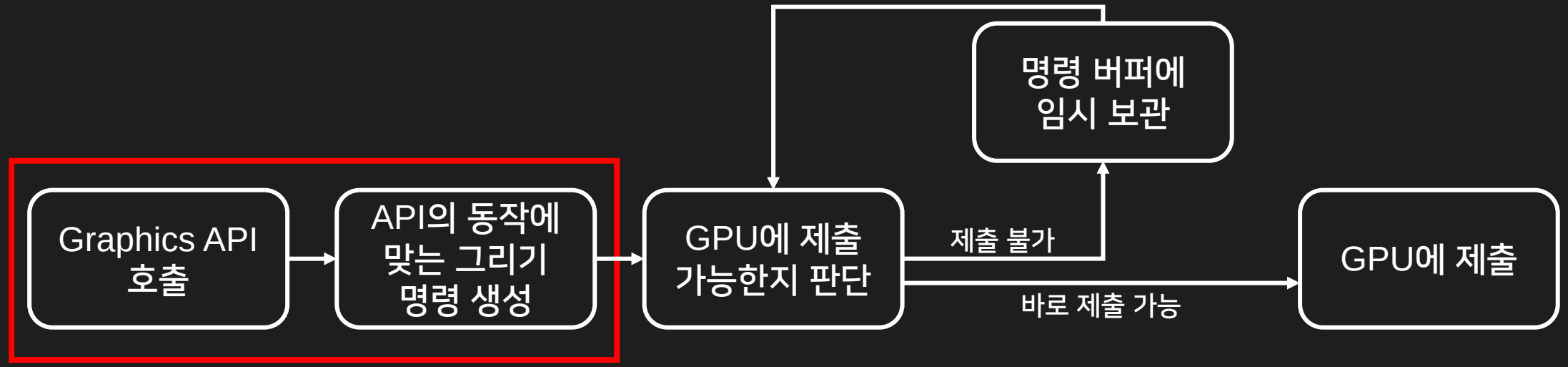
GPU가 무언가를 그리도록 요청하기 위해서 Graphics API를 호출하면 다음과 같은 순서로 그리기 명령(Rendering command)이 생성되어 GPU로 제출됩니다.



Multithreaded Rendering 왜 해야 할까?

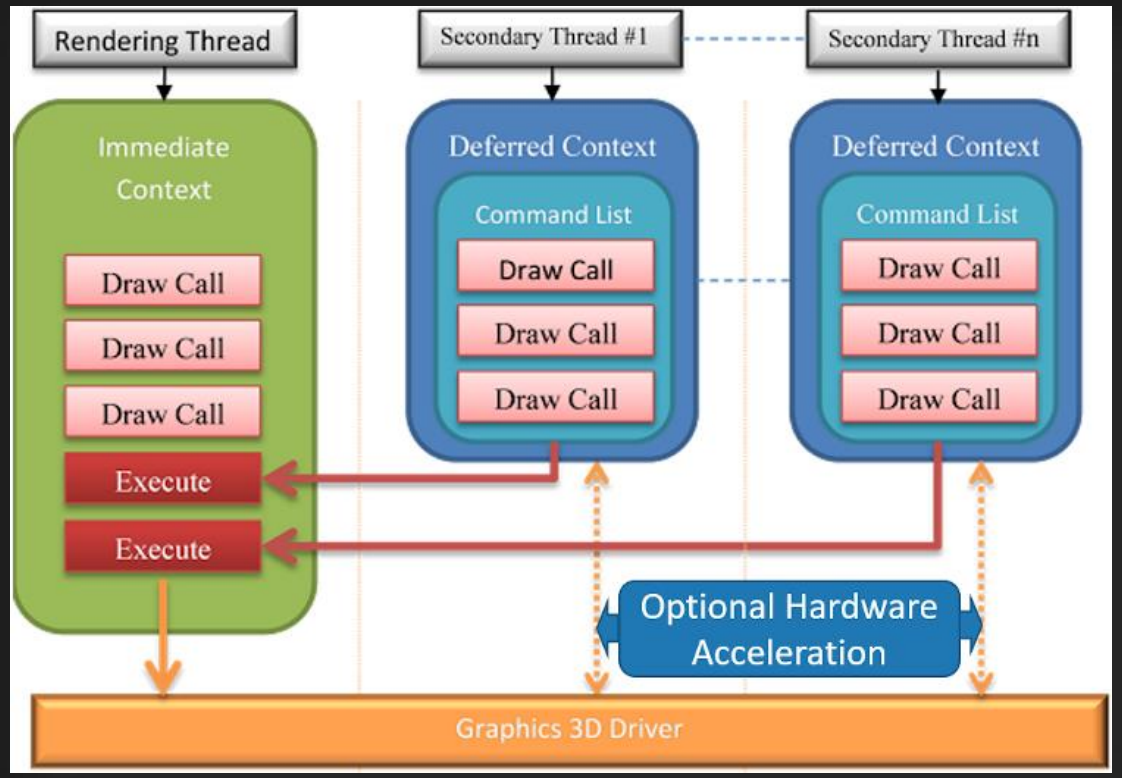
그리기 명령을 생성하고 이를 제출하는 과정은 CPU를 통해서 처리됩니다.

그리고 이 과정이 CPU에서 처리된다면 이 부분을 스레드를 통해서 개선 할 수 있습니다.



Multithreaded Rendering 왜 해야 할까?

Multithreaded Rendering은 그리기 명령을 생성하는 부분을 여러 스레드를 통해서 빠르게 생성하고 이를 GPU에 제출하는 것입니다.



출처 : https://software.intel.com/content/www/us/en/develop/articles/understanding-directx-multithreaded-rendering-performance-by-experiments.html

Multithreaded Rendering 왜 해야 할까?

정리하자면 Multithreaded Rendering은 CPU와 GPU의 사용률을 최대로 하여 높은 성능을 얻으려는 방법입니다.

하지만 스레드를 사용한다면 그에 따른 대가가 따릅니다.

이제 멀티 스레드에서 렌더링을 하기 위해 처리해야 하는 이슈에 대해서 알아보도록 하겠습니다.

Data Race

데이터 경합은 멀티 스레드 프로그래밍에서 피할 수 없는 이슈입니다.

Multithreaded Rendering시 스레드 간의 경합에 더해 GPU와의 경합도 발생합니다.

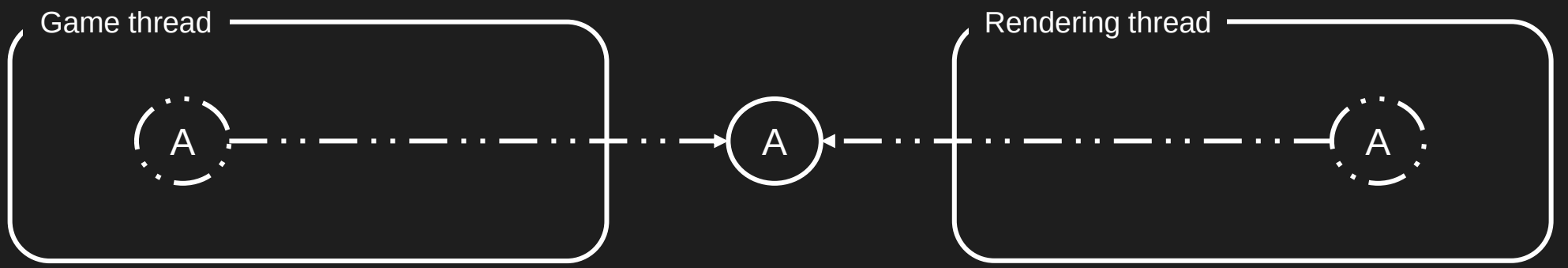
따라서 이런 데이터 경합이 발생하지 않도록 전략을 세워야 합니다.

우선 렌더링 상황에서 일어날 수 있는 데이터 경합의 예시를 몇가지 생각해보겠습니다.

Data Race

첫번째는 스레드간 데이터 경합입니다. 다음과 같이 게임 로직과 렌더링이 별도의 스레드에서 이뤄지고 있는 상황을 생각해보도록 하겠습니다.

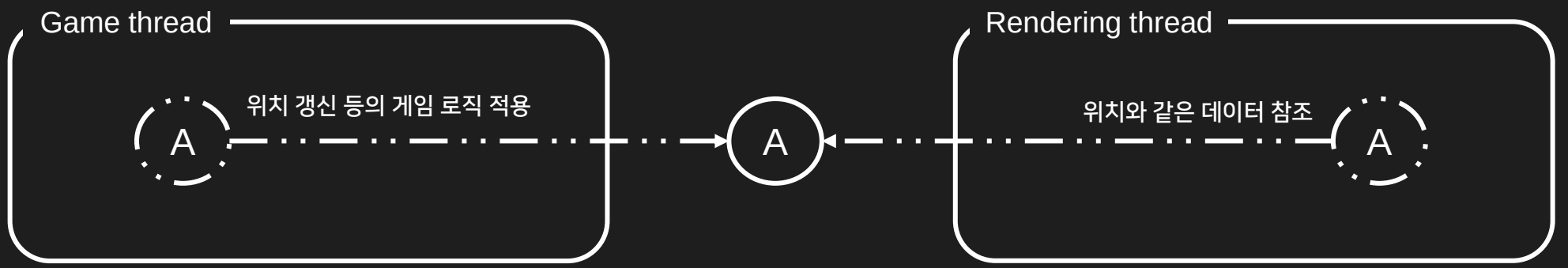
그리고 여기에는 화면에 그려질 수 있는 게임 오브젝트 A가 있습니다.



Data Race

게임 스레드는 A에 대한 게임 로직을 수행하고 렌더링 스레드에서는 A를 화면에 그립니다.

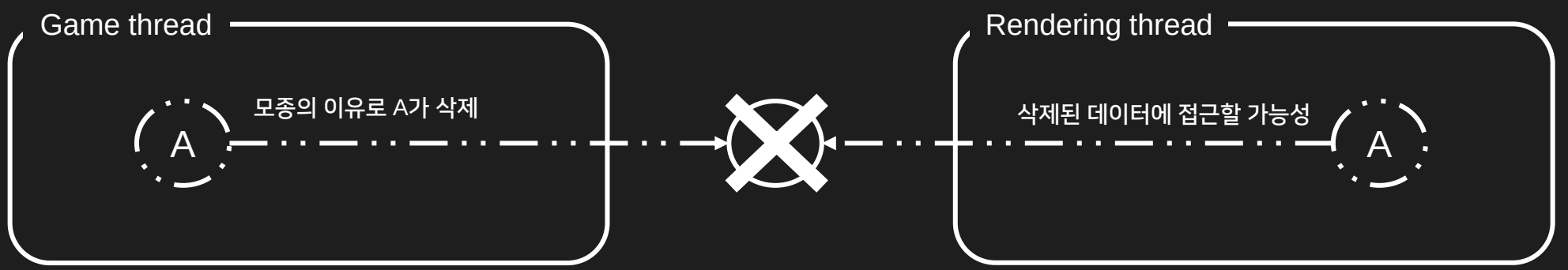
즉 A는 두 개의 스레드가 공유하고 있는 자원입니다.



Data Race

만약 게임 로직에 따라서 A라는 오브젝트가 삭제 되는데 렌더링 스레드가 A를 참조하고 있는 상황이라면 게임 스레드가 A를 바로 삭제하는 것은 문제가 됩니다.

따라서 렌더링 스레드가 A를 참조하지 않을 때 까지 A의 삭제는 유보돼야 합니다.



Data Race

두번째는 GPU와의 경합입니다. GPU에서 A를 그리기 위한 셰이더 코드가 실행되고 있는 경우를 생각해보겠습니다.

GPU는 그래픽 카드 메모리로 전송된 물체의 위치나 재질을 참조하여 물체를 어디에 어떻게 그려야 할지를 결정합니다.

Data Race

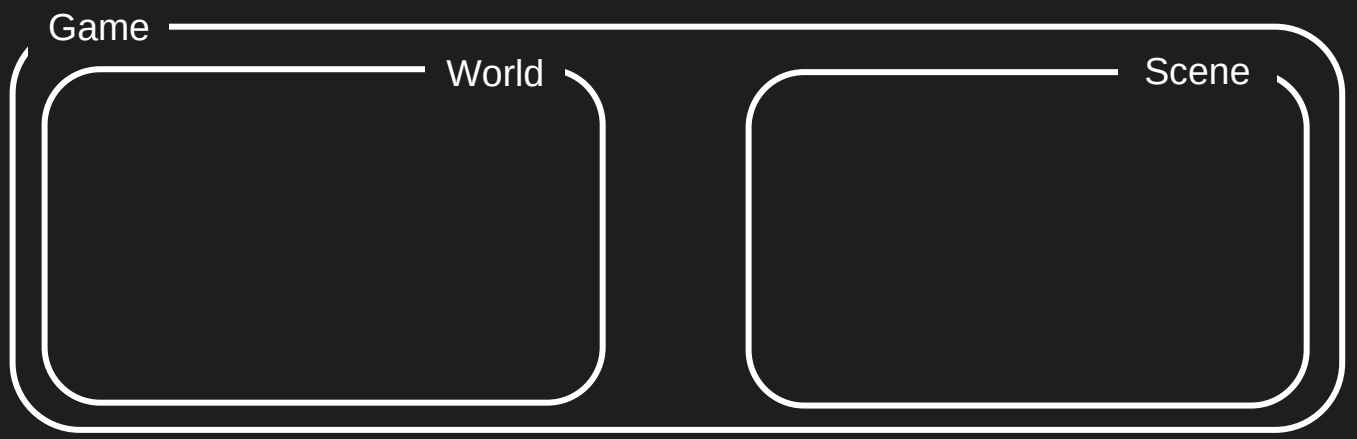
이런 상황에서 CPU가 A의 상태를 업데이트하고 이 결과를 그래픽 카드로 메모리로 전송하면 A를 그리고 있는 도중에 참조하고 있던 데이터의 값이 변경될 수 있습니다.

Data Race

경합 해결 전략

스레드 간의 데이터 경합 그리고 CPU와 GPU 간의 데이터 경합을 해결하기 위한 전략은 게임의 세상을 2가지로 나누는 것입니다.

게임의 세상을 게임 스레드를 위한 World와 렌더링 스레드를 위한 Scene으로 나눕니다.

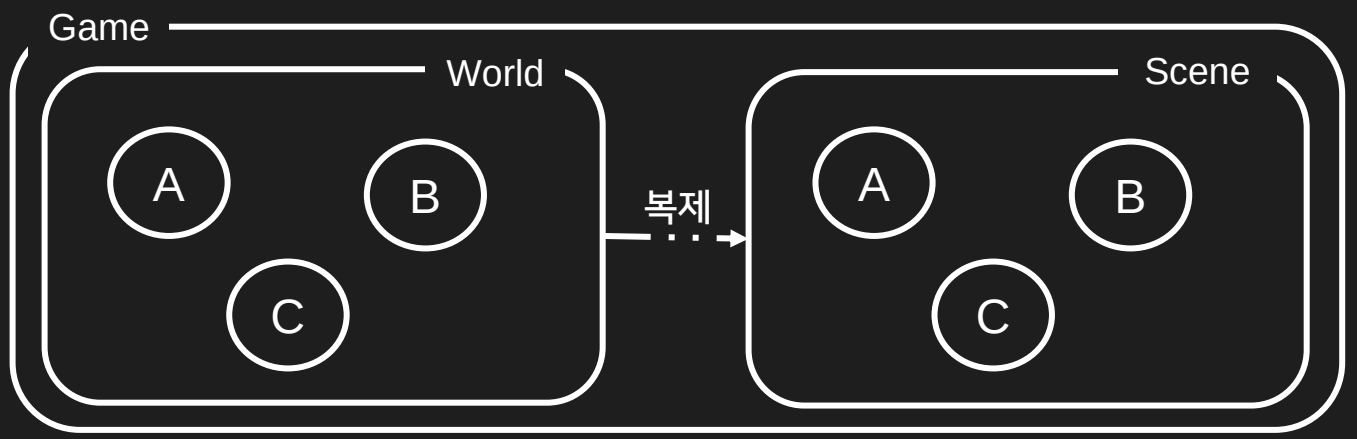


Data Race

경합 해결 전략

Scene은 World의 복제 본인데 렌더링에 관련된 데이터에만 복사해 온 렌더링을 위한 세상입니다.

그리고 게임 스레드는 World만을 수정하고 렌더링 스레드는 Scene만을 수정하도록 엄격하게 제한합니다.



Data Race

경합 해결 전략

이는 프로그래머가 신경을 써야 할 부분이기 때문에 실수를 줄이기 위해서 다음과 같은 도구가 사용될 수 있습니다.

다음은 특정 스레드에서만 수행되야 하는 코드가 해당 스레드에서 실행되는지 검사하여 이런 제약을 준수할 수 있도록 하는 예시입니다.

```cpp
void Scene::AddPrimitiveSceneInfo( PrimitiveSceneInfo* primitiveSceneInfo )
{
    assert( IsInRenderThread() );
    assert( primitiveSceneInfo );

    m_primitiveToUpdate.push_back( static_cast<uint32>( m_primitives.size( ) ) );
    primitiveSceneInfo->m_primitiveId = static_cast<uint32>( m_primitives.size( ) );

    m_primitives.push_back( primitiveSceneInfo );

    primitiveSceneInfo->AddToScene( );
}
```

```cpp
bool IsInRenderThread( )
{
    return GetInterface<ITaskScheduler>( )->GetThisThreadType( ) == ThreadType::RenderThread;
}
```

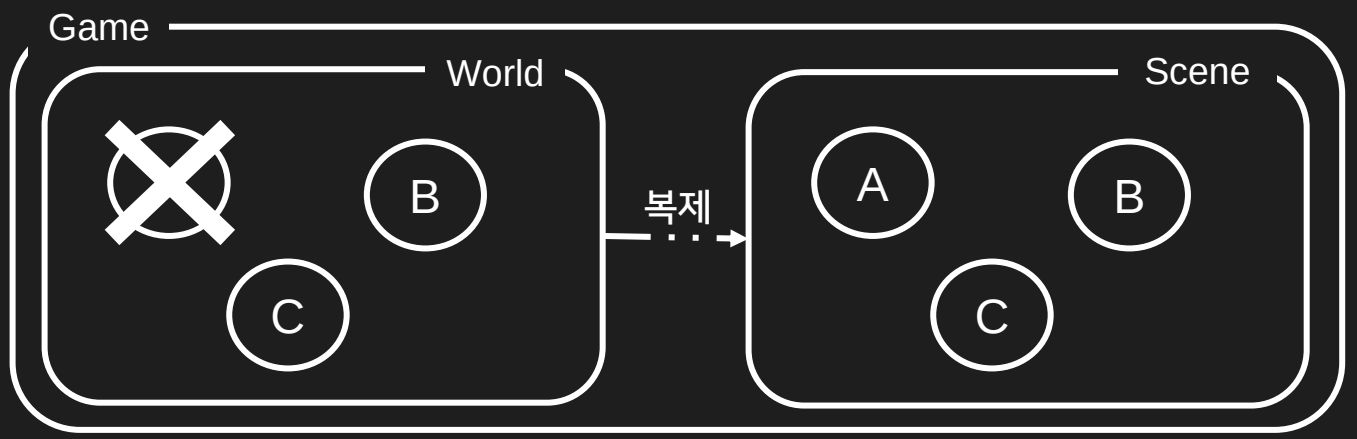
```cpp
size_t TaskScheduler::GetThisThreadType( ) const
{
    std::thread::id thisThreadId = std::this_thread::get_id( );

    for ( size_t i = 0; i < m_workerCount; ++i )
    {
        if ( m_workerid[i] == thisThreadId )
        {
            return i;
        }
    }

    return m_workerCount;
}
```

Data Race

경합 해결 전략

다음과 같이 게임 스레드가 월드의 A를 삭제해도 Scene에는 영향이 없기 때문에 삭제된 오브젝트에 접근해서 문제가 발생하는 경우를 방지할 수 있습니다.

그럼 Scene의 A는 어떻게 삭제해야 할까요?

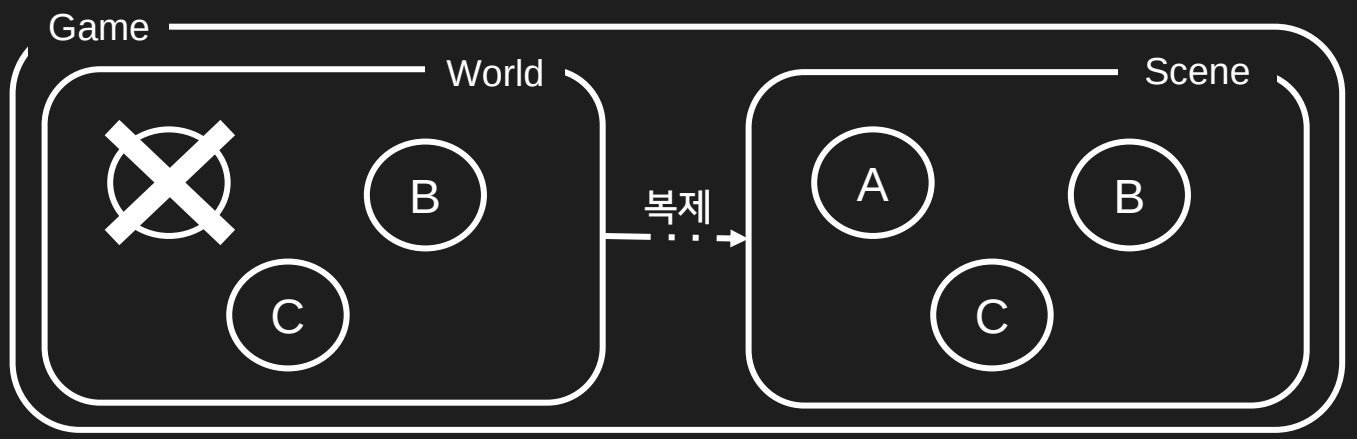


Data Race

경합 해결 전략

A는 게임 로직에 따라서 삭제되었습니다. World는 게임 스레드에서 수정할 수 있으니 삭제가 가능했지만 Scene은 렌더링 스레드에 의해서만 수정되야 하기 때문에 게임 로직에서 이를 삭제할 수 없습니다.

그러므로 스레드 접근 제한을 준수하기 위해서 게임 스레드는 렌더링 스레드가 A를 삭제하도록 요청해야 합니다.

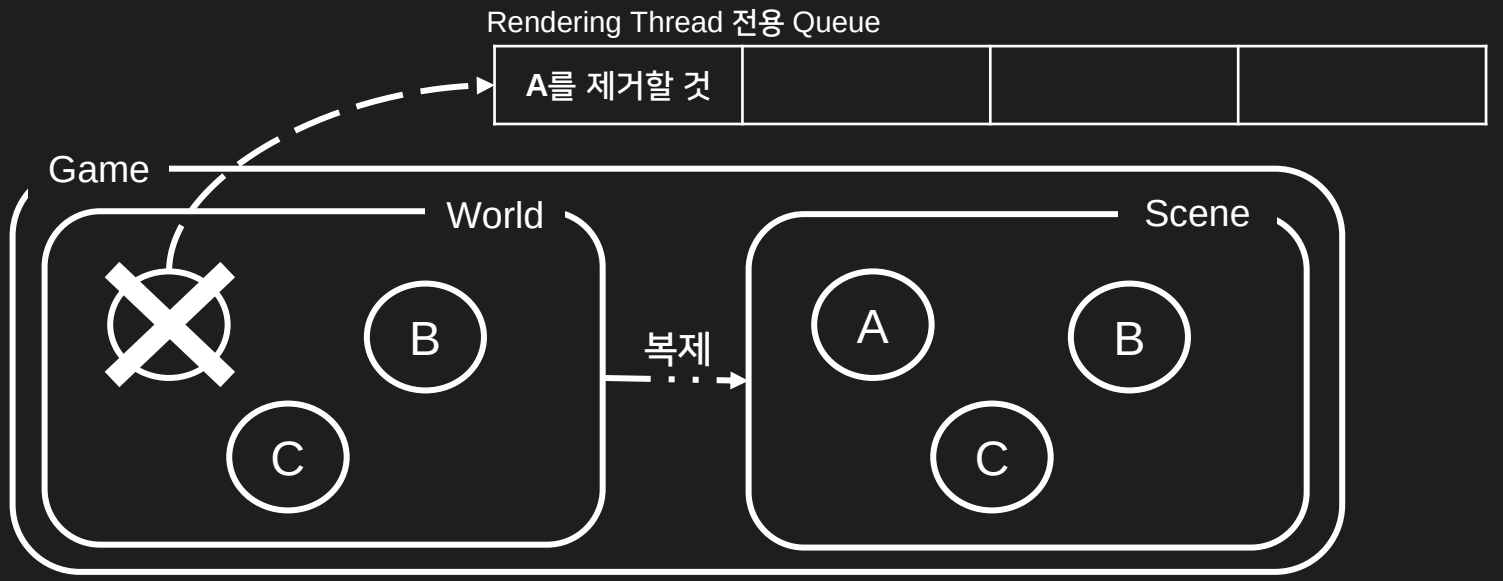


Data Race

경합 해결 전략

스레드에 대한 요청은 스레드의 전용 큐를 통해서 이뤄집니다.

A가 삭제 될 경우 게임 스레드는 A에 대한 삭제 요청을 렌더링 스레드 큐에 집어 넣고 렌더링 스레드는 적절한 때에 큐의 요청을 처리하게 됩니다.



Data Race

경합 해결 전략

실제 코드를 통해서 렌더링 스레드에 오브젝트의 삭제를 요청하는 예시를 보겠습니다.

```
1  void Scene::RemovePrimitive( PrimitiveComponent* primitive )
2  {
3      PrimitiveProxy* proxy = primitive->m_sceneProxy;
4
5      if ( proxy )
6      {
7          PrimitiveSceneInfo* primitiveSceneInfo = proxy->m_primitiveSceneInfo;
8          primitive->m_sceneProxy = nullptr;
9
10         EnqueueRenderTask( [this, primitiveSceneInfo]( )
11         {
12             RemovePrimitiveSceneInfo( primitiveSceneInfo );
13         } );
14     }
15 }
```

렌더링 스레드에서 RemovePrimitiveSceneInfo() 함수를 호출하도록 요청

Data Race

경합 해결 전략

EnqueueRenderTask() 함수는 렌더링 스레드에 태스크를 제출하는 함수이며 렌더링 스레드에서 호출한 경우에는 해당 태스크를 바로 실행합니다.

```cpp
template <typename Lambda>
void EnqueueRenderTask( Lambda lambda )
{
    if ( IsInRenderThread( ) )
    {
        lambda( );
    }
    else
    {
        auto* task = Task<LambdaTask<Lambda>>::Create( WorkerAffinityMask<ThreadType::RenderThread>( ), lambda );
        EnqueueRenderTask( static_cast<TaskBase*>( task ) );
    }
}
```

렌더링 스레드에서 호출한 경우 바로 실행

```cpp
void EnqueueRenderTask( TaskBase* task )
{
    assert( task->WorkerAffinity( ) == WorkerAffinityMask<ThreadType::RenderThread>( ) );
    auto taskScheduler = static_cast<EngineTaskScheduler*>( GetInterface<ITaskScheduler>( ) );
    TaskHandle taskGroup = taskScheduler->GetExclusiveTaskGroup( ThreadType::RenderThread );
    taskGroup.AddTask( task );
    [[maybe_unused]] bool success = taskScheduler->Run( taskGroup );
    assert( success );
}
```

전용 스레드 큐에 접근

Data Race

경합 해결 전략

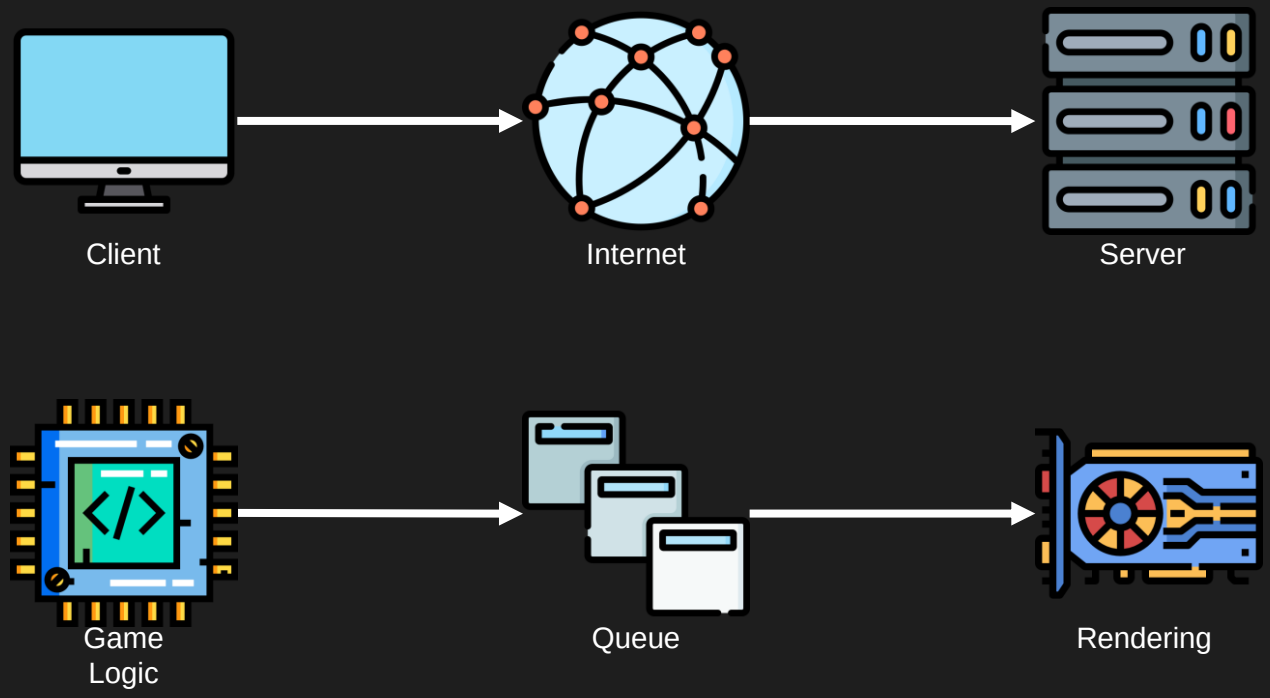
전용 큐는 각 스레드당 하나로 제한하였는데 이는 다른 스레드의 요청이 순서를 지켜 실행돼야 하기 때문입니다.

‘A 물체의 위치 업데이트 -> 게임 장면 그리기 -> A 물체의 삭제’
와 같은 요청이 순서가 보장되지 않아
‘A 물체의 삭제 -> A 물체의 위치 업데이트 -> 게임 장면 그리기’
와 같은 순서로 실행되면 의도하지 않은 동작이기 때문입니다.

Data Race

경합 해결 전략

이러한 방법은 게임 로직과 렌더링을 마치 클라이언트 서버 모델과 유사하게 다루게 합니다.



Data Race

Code Reading

이제 게임 물체가 추가될 때 실행되는 코드를 보면서 지금까지 이야기한 내용을 정리하도록 하겠습니다.

새로 생성된 게임 물체는 World 클래스의 SpawnObject() 함수를 통해서 게임 세상에 추가됩니다.

```
1 void World::SpawnObject( CGameLogic& gameLogic, Owner<CGameObject*> object )
2 {
3     object->Initialize( gameLogic, *this );
4     object->SetID( m_gameObjects.size( ) );
5     m_gameObjects.emplace_back( object );
6 }
```

Data Race

Code Reading

World 클래스는 게임 스레드에서 참조할 수 있는 게임 물체인 CGameObject 객체들을 보관하고 있으며 World와 쌍을 이루는 렌더링 스레드 전용 세상인 Scene을 참조하고 있습니다.

```
1  // 설명에 중요하지 않은 부분은 생략되어 있습니다.
2  class World
3  {
4  public:
5      const std::vector<std::unique_ptr<CGameObject>>& GameObjects( )
6      {
7          return m_gameObjects;
8      }
9
10     IScene* Scene( ) const
11     {
12         return m_scene;
13     }
14
15 private:
16     std::vector<std::unique_ptr<CGameObject>> m_gameObjects;
17 };
```

Data Race

Code Reading

Scene 클래스는 World 와 유사하게 렌더링 스레드에서 참조할 수 있는 게임 물체인 PrimitiveSceneInfo 객체를 보관하고 있으며 이 객체는 게임 스레드의 요청에 의해 Scene에 추가되거나 삭제됩니다.

```
1  // 설명에 중요하지 않은 부분은 생략되어 있습니다.
2  class Scene final : public IScene
3  {
4  public:
5      Scene( );
6
7      virtual void AddPrimitive( PrimitiveComponent* primitive ) override;
8      virtual void RemovePrimitive( PrimitiveComponent* primitive ) override;
9
10 private:
11     std::vector<PrimitiveSceneInfo*> m_primitives;
12 };
```

Data Race

Code Reading

게임 스레드는 렌더링 할 필요가 있는 경우에 SpawnObject() 함수에서 오브젝트를 초기화 할 때 필요한 에셋이 모두 갖춰졌는지 판단하여 렌더링 스레드에 PrimitiveSceneInfo 객체의 추가를 요청합니다.

코드 흐름은 다음과 같습니다.

```
1 void CGameObject::Initialize( CGameLogic& gameLogic, World& world )
2 {
3     m_pWorld = &world;
4
5     for ( Component* component : m_components )
6     {
7         component->RegisterComponent( );
8     }
9 }
```

Data Race

Code Reading

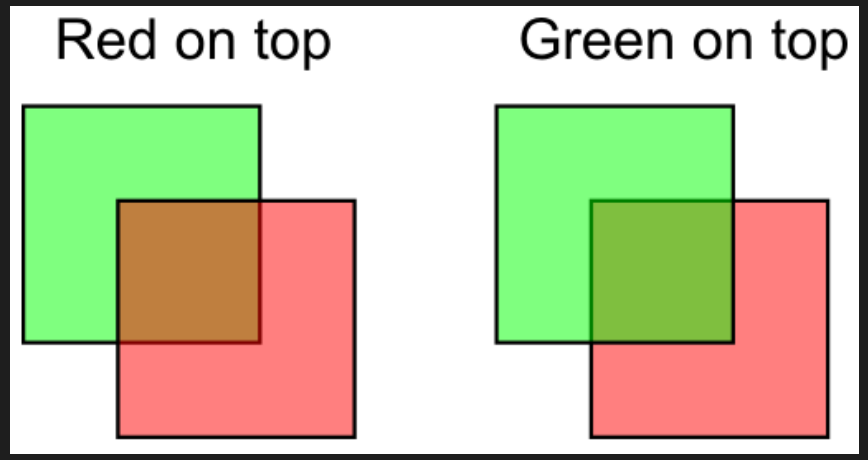
```
1  void Component::RegisterComponent( )
2  {
3      RegisterComponent( m_pOwner->GetWorld( ) );
4  }
5
6  void Component::RegisterComponent( World* pWorld )
7  {
8      m_pWorld = pWorld;
9
10     if ( m_renderStateCreated == false && ShouldCreateRenderState( ) )
11     {
12         CreateRenderState( );
13     }
14 }
15
16 // 렌더링 기능을 가지고 있는 컴포넌트 ( Component를 상속 )
17 void PrimitiveComponent::CreateRenderState( )
18 {
19     SceneComponent::CreateRenderState( );
20     m_pWorld->Scene( )->AddPrimitive( this );
21 }
22
23 // 정적 메시의 컴포넌트
24 PrimitiveProxy* StaticMeshComponent::CreateProxy( ) const
25 {
26     // 메시 렌더링에 필요한 에셋이 세팅되었는지 검사
27     if ( m_pStaticMesh == nullptr
28         || m_pStaticMesh->RenderData( ) == nullptr
29         || m_pRenderOption == nullptr )
30     {
31         return nullptr;
32     }
33
34     return new StaticMeshPrimitiveProxy( *this );
35 }
```

```
1  void Scene::AddPrimitive( PrimitiveComponent* primitive )
2  {
3      // 프록시 생성을 시도 렌더링에 필요한 에셋이 갖춰지면 nullptr이 아님
4      PrimitiveProxy* proxy = primitive->CreateProxy( );
5      primitive->m_sceneProxy = proxy;
6
7      if ( proxy == nullptr )
8      {
9          return;
10     }
11
12     PrimitiveSceneInfo* primitiveSceneInfo = new PrimitiveSceneInfo( primitive, *this );
13
14     proxy->m_primitiveSceneInfo = primitiveSceneInfo;
15
16     // 렌더링에 필요한 정보를 복사 현재는 월드 변환 행렬만 사용중
17     struct AddPrimitiveSceneInfoParam
18     {
19         CXMFLOAT4X4 m_worldTransform;
20     };
21     AddPrimitiveSceneInfoParam param = {
22         primitive->GetRenderMatrix( )
23     };
24
25     // 렌더링 스레드에서 Scene에 추가하도록 요청
26     EnqueueRenderTask( [this, param, primitiveSceneInfo]( )
27     {
28         PrimitiveProxy* sceneProxy = primitiveSceneInfo->m_sceneProxy;
29
30         sceneProxy->SetTransform( param.m_worldTransform );
31         sceneProxy->CreateRenderData( );
32
33         AddPrimitiveSceneInfo( primitiveSceneInfo );
34     } );
35 }
```

Rendering Command 생성

그리기 작업은 때때로 순서가 중요한 경우가 있습니다.

예를 들면 반투명 물체와 같이 카메라에 먼 순서부터 물체를 그려야 하는 경우 (Z Sorting)로 순서를 보장하기 위해서 어떻게 병렬화를 하는 것이 좋을지 고려해야 합니다.

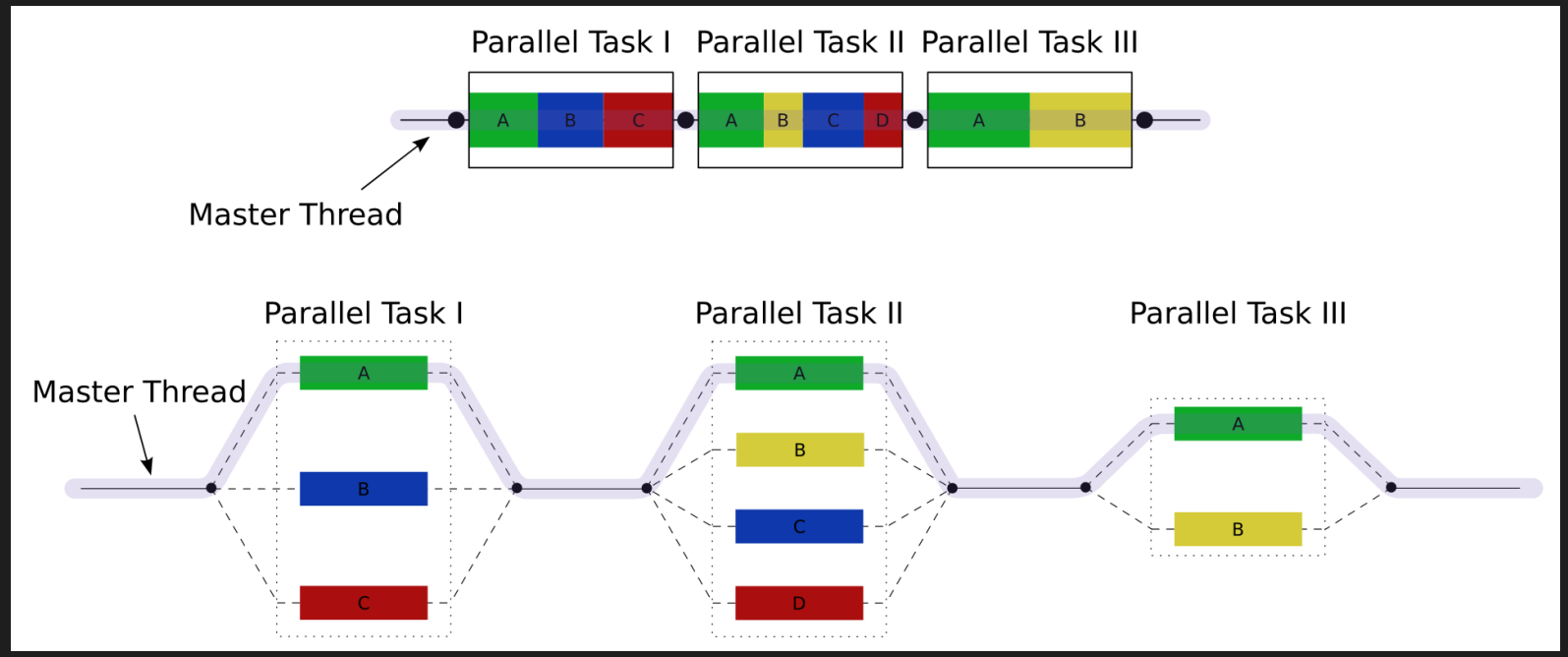


50% 투명도를 가진 두 물체를 서로 다른 순서로 그렸을 때 최종 색상이 달라지는 것을 확인할 수 있음.

출처 : http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-10-transparency/

Rendering Command 생성

여기서는 전형적인 Fork-join 모델을 사용하여 그려야 할 전체 리스트를 작업 스레드의 개수로 나눠 처리하였습니다.



출처 : https://en.wikipedia.org/wiki/Fork–join_model

Rendering Command 생성

깊이 렌더링이나 그림자 맵 렌더링과 같이 Z 버퍼를 사용할 수 있는 상황에서는 그리기 순서가 그리 중요하지 않기 때문에 다른 병렬화 전략을 취할 수 있습니다.

예를 들면 Join시 모든 태스크의 완료를 기다리지 않고 완료된 태스크부터 GPU에 명령을 제출할 수도 있습니다.

현재 코드는 모든 스레드를 기다리도록 구현되어 있지만 모든 상황에 알맞은 방법은 아니라는 점을 언급하고 싶습니다.

Rendering Command 생성

Direct3D11 Deferred Context

실제 코드를 보기 전에 Direct3D11의 Deferred Context에 대한 한 가지 특이점을 언급하고자 합니다.

GPU에 명령을 즉시 제출하는 Immediate Context는 일종의 상태 머신과 같아 렌더링 파이프라인의 상태를 바꾸는 명령(RSSetState, OMSetBlendState 등)을 통해 상태가 변경되면 해당 상태가 계속 유지되었습니다.

예를 들어 깊이 테스트를 끄도록 했다면 다시 깊이 테스트를 키는 명령을 제출하기 전까지 해당 상태가 유지되어 다음 그리기에도 영향을 미칩니다.

Rendering Command 생성

Direct3D11 Deferred Context

하지만 Deferred Context의 경우는 생성시 Immediate Context의 파이프라인 상태와 상관 없는 기본 상태로 생성되고 Immediate Context에 제출해도 파이프라인 상태를 변경시키지 않습니다.

잠시 다음 질문을 생각해 보시기 바랍니다.

Q. 이전 그리기에서 Immediate Context를 통해 뷰포트를 설정한 다음에 Deferred Context를 통해 기록된 그리기 명령을 Immediate Context에 제출했을 때 해당 명령들은 Immediate Context에 설정된 뷰포트의 영향을 받을까요?

Rendering Command 생성

Direct3D11 Deferred Context

A. 영향을 받지 않습니다. 그리고 Deferred Context에 뷰포트를 설정하는 명령을 기록하지 않았다면 Deferred Context는 기본 설정으로 생성되기 때문에 정상적으로 렌더링이 이뤄지지 않습니다.

그럼 다음과 같은 경우는 어떨까요?

Q. Deferred Context 2개 D1, D2에 각각 명령을 기록하고 D1 -> D2의 순서로 Immediate Context에 제출하였습니다. D1의 파이프라인 상태는 D2에 영향을 미칠까요?

Rendering Command 생성

Direct3D11 Deferred Context

A. 영향을 미치지 않습니다. D2에 기록된 명령들은 D2의 상태에만 영향을 받습니다.

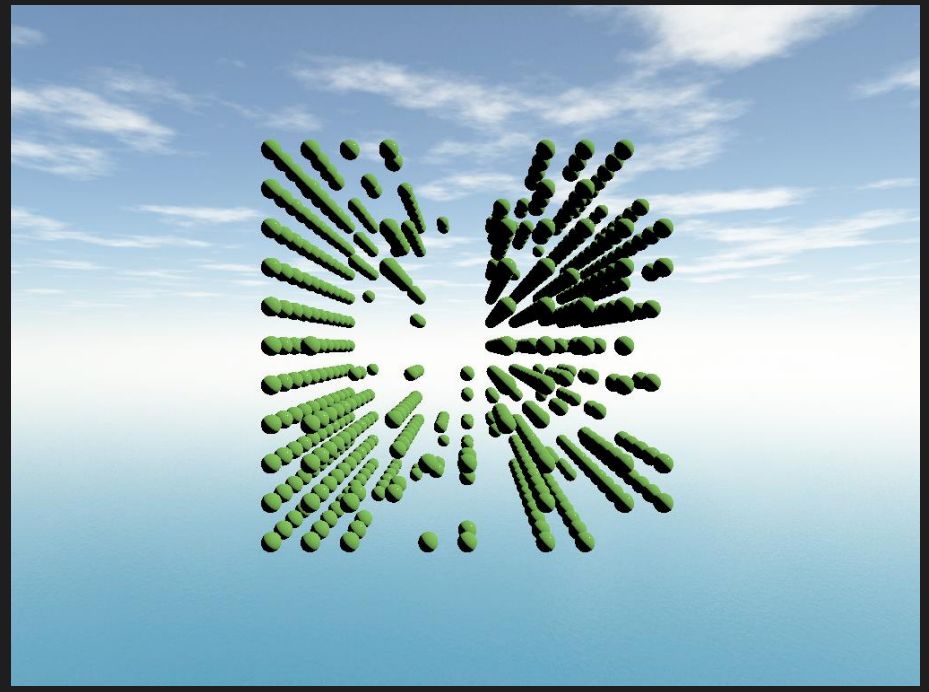
최종적으로 정리해보면 Deferred Context에 명령을 기록할 때는 Viewport, Scissor rectangle, Render Target View 와 같은 상태를 매번 설정해 줘야 합니다. 다음과 같이 일부 설정을 누락하면…

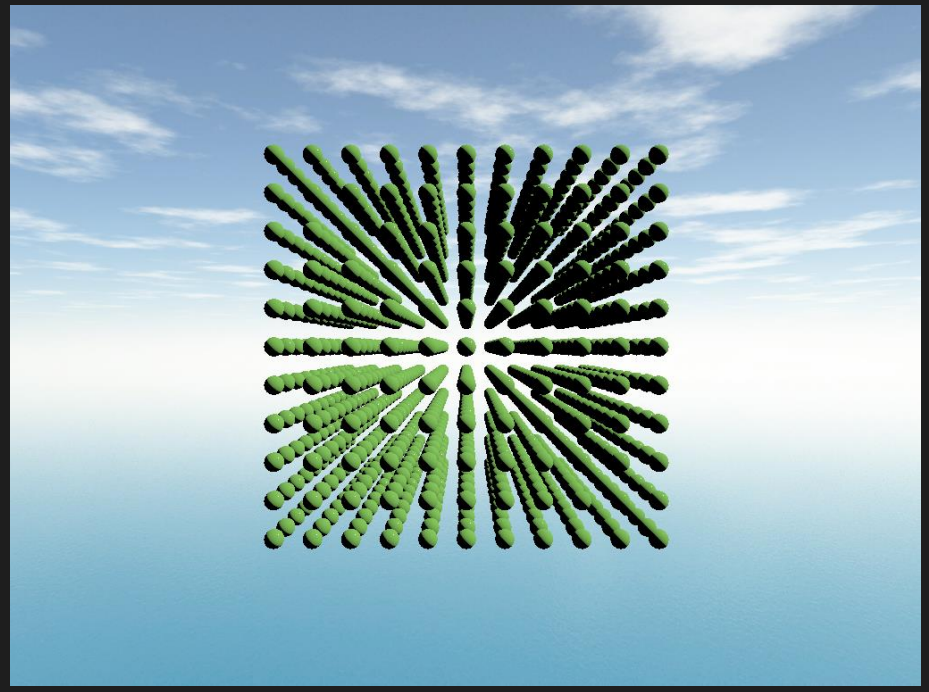
```
1 // 2개의 Deferred Context중 하나에만 Viewport와 RTV를 세팅
2 for ( size_t i = 0; i < 1; ++i )
3 {
4     renderer.SetRenderTarget( *deferredCommandLists[i], renderViewGroup );
5     renderer.SetViewPort( *deferredCommandLists[i], renderViewGroup );
6 }
```

Rendering Command 생성

Direct3D11 Deferred Context

좌측 그림과 같이 비정상적인 화면을 얻게 됩니다.





Rendering Command 생성

Code Reading

이제 관련 코드를 보도록 하겠습니다. 그리기 명령을 병렬로 제출하는 ParallelCommitDrawSnapshot() 함수입니다.

```
1 void ParallelCommitDrawSnapshot( SceneRenderer& renderer, RenderViewGroup& renderViewGroup, size_t
  curView, VertexBuffer& primitiveIds )
2 {
3     auto& view = renderViewGroup[curView];
4     std::vector<VisibleDrawSnapshot>& visibleSnapshots = view.m_snapshots;
5
6     // 실제 Draw call의 갯수를 셈
7     size_t dc = 0;
8     for ( size_t i = 0; i < visibleSnapshots.size( ); )
9     {
10        ++dc;
11        i += visibleSnapshots[i].m_numInstance;
12    }
13
14    // 64개 보다 작다면 단일 스레드로 제출
15    if ( dc < 64 )
16    {
17        CommitDrawSnapshots( renderer, renderViewGroup, curView, primitiveIds );
18    }
19    else
20    {
21        auto taskScheduler = GetInterface<ITaskScheduler>( );
22        // WorkerThread에서 실행하도록 설정
23        constexpr size_t affinityMask = WorkerAffinityMask<WorkerThreads>( );
24        TaskHandle taskGroup = taskScheduler->GetTaskGroup( );
25
26        CommitDrawSnapshotTask* commitTasks[2] = {};
27        std::unique_ptr<aga::IDeferredCommandList> deferredCommandLists[2] = {};
28
29
```

Rendering Command 생성

Code Reading

```
1         for ( size_t i = 0, j = 0; i < std::extent_v<decltype(commitTasks)>; ++i )
2         {
3             // 그리기 명령을 기록할 CommandList 생성
4             deferredCommandLists[i] = GetInterface<aga::IAga>( )->CreateDeferredCommandList( );
5
6             size_t count = ( dc + 1 ) / 2;
7             // 태스크 생성
8             auto task = Task<CommitDrawSnapshotTask>::Create( affinityMask, count,
  *deferredCommandLists[i], primitiveIds );
9             commitTasks[i] = &task->Element( );
10
11            size_t added = 0;
12            // 태스크에 그려야 할 대상을 추가
13            for ( ; j < visibleSnapshots.size( ) && ( added < count ); )
14            {
15                commitTasks[i]->AddSnapshot( &visibleSnapshots[j] );
16                j += visibleSnapshots[j].m_numInstance;
17                ++added;
18            }
19
20            // 태스크 그룹에 태스크 추가
21            taskGroup.AddTask( task );
22        }
23
24        // CommandList의 렌더링 상태가 초기화되어 있기 때문에 렌더타겟, 뷰포트 설정도 각기 해줘야 함
25        for ( size_t i = 0; i < std::extent_v<decltype( deferredCommandLists )>; ++i )
26        {
27            renderer.SetRenderTarget( *deferredCommandLists[i], renderViewGroup );
28            renderer.SetViewPort( *deferredCommandLists[i], renderViewGroup );
29        }
30
31        // 태스크 실행 (Fork)
32        taskScheduler->Run( taskGroup );
33
34        // 태스크 대기 (Join)
35        taskScheduler->Wait( taskGroup );
36
37        auto immediateCommandList = GetInterface<aga::IAga>( )->GetImmediateCommandList( );
38
39        // GPU에 그리기 명령을 순서대로 제출
40        for ( size_t i = 0; i < std::extent_v<decltype( deferredCommandLists )>; ++i )
41        {
42            immediateCommandList->Execute( *deferredCommandLists[i] );
43        }
44    }
45 }
```

Rendering Command 생성

Code Reading

실제로 명령을 기록하는 CommitDrawSnapshotTask는 이렇습니다.

```
1  class CommitDrawSnapshotTask
2  {
3  public:
4      void DoTask( )
5      {
6          // 그리기 명령을 기록
7          for ( auto drawSnapshot : m_drawSnapshot )
8          {
9              // 단일 스레드용 CommandList와 동일한 인터페이스를 사용하므로 단일 스레드용 함수를 재활용
10             CommitDrawSnapshot( m_commandList, *drawSnapshot, m_primitiveIds );
11         }
12         // 기록 종료
13         m_commandList.Finish( );
14     }
15
16     void AddSnapshot( VisibleDrawSnapshot* snapshot )
17     {
18         // 작업 대상을 추가
19         m_drawSnapshot.push_back( snapshot );
20     }
21
22     CommitDrawSnapshotTask( size_t reserveSize, aga::IDeferredCommandList& commandList,
   VertexBuffer& primitiveIds ) : m_commandList( commandList ), m_primitiveIds( primitiveIds )
23     {
24         m_drawSnapshot.reserve( reserveSize );
25     }
26
27 private:
28     aga::IDeferredCommandList& m_commandList;
29     VertexBuffer& m_primitiveIds;
30     std::vector<VisibleDrawSnapshot*> m_drawSnapshot;
31 };
```

Rendering Command 생성

Code Reading

싱글 스레드로 제출한 경우와 2개의 스레드를 통해서 명령을 제출하는데 걸린 시간의 비교표입니다.

스레드 1개	스레드 2개
16ms	7ms
13ms	7ms
14ms	5ms
12ms	6ms
11ms	4ms
7ms	5ms
5ms	4ms
6ms	6ms
9ms	6ms
6ms	5ms
11ms	5ms
7ms	5ms
5ms	6ms
평균 : 9.38461ms	평균 : 5.46154ms

- 사양 -

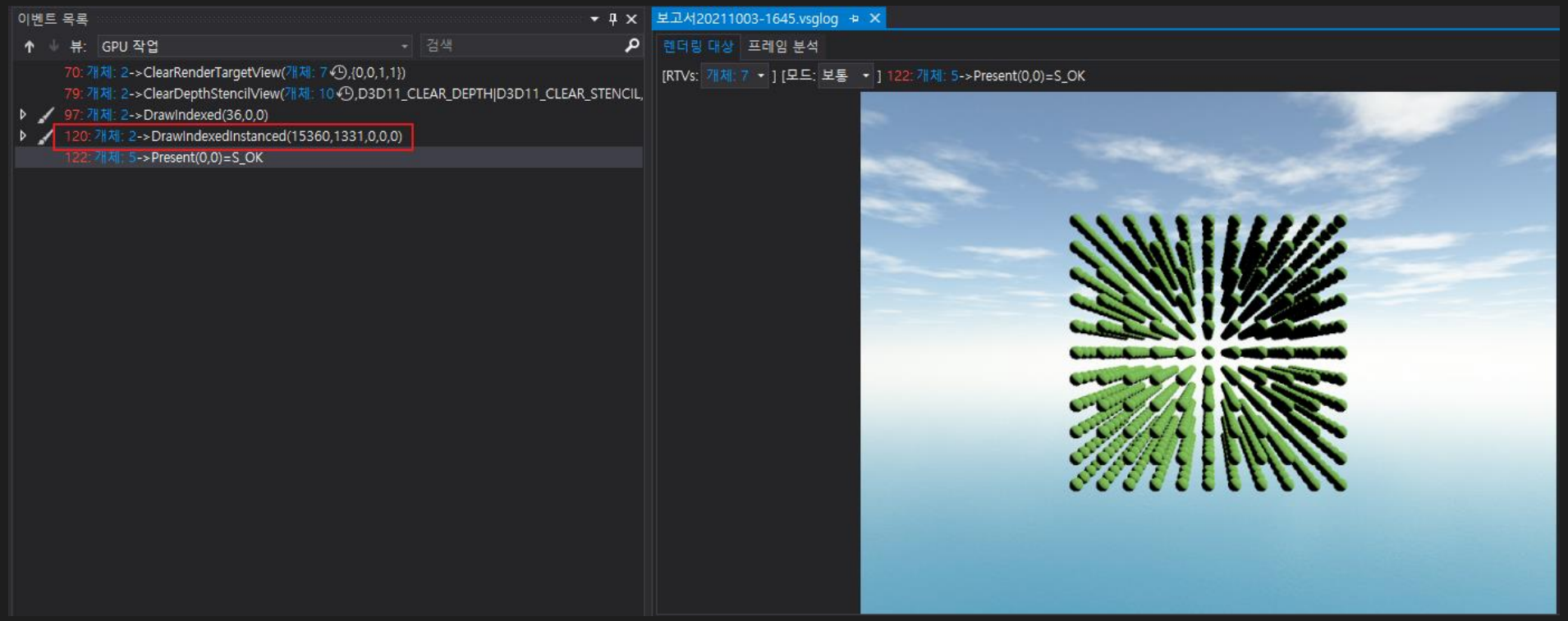
CPU : Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz (2코어 4스레드)

GPU : Intel(R) HD Graphics Family

RAM : 8 GB
Visual Studio 2017 64 bit 빌드

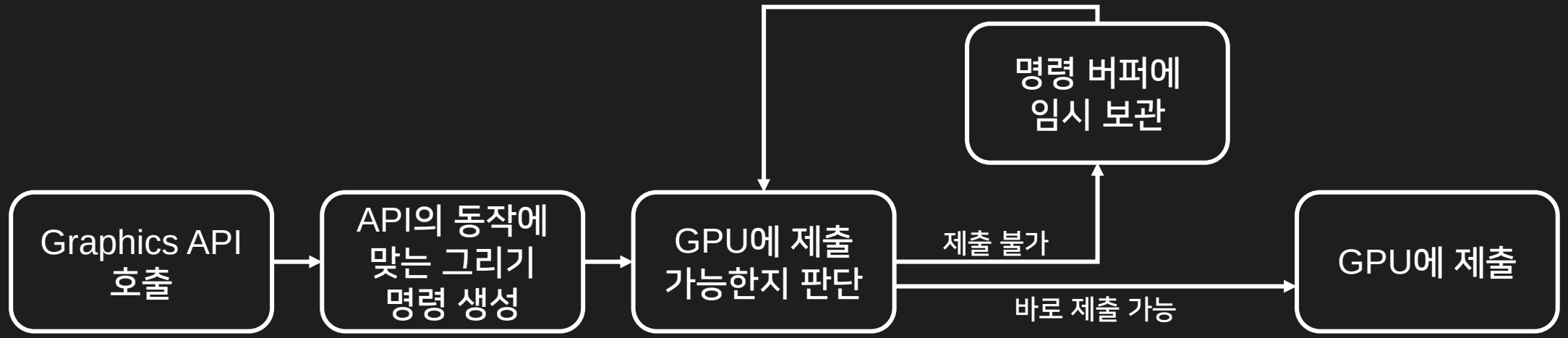
Dynamic Instancing

인스턴싱(Instancing)은 동일한 물체 여러 개를 하나의 드로우 콜로 한번에 그리는 방식을 말합니다. 다음 장면의 구들도 한번의 드로우 콜로 그렸습니다.



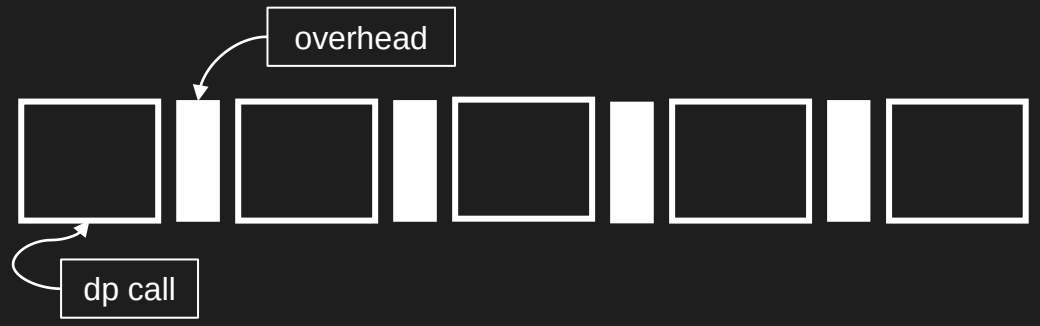
Dynamic Instancing

GPU가 일을 하기 위해서는 CPU로 부터의 명령이 필요한데 명령을 제출하는데 비교적 많은 시간이 걸립니다.



Dynamic Instancing

여러 물체를 그리는 상황을 간단하게 표현해보면 다음과 같이 오버 헤드가 매 드로우 콜마다 발생합니다.



인스턴싱은 동일한 물체들을 한번에 그려 드로우 콜 마다 발생하는 오버 헤드를 줄이게 됩니다.

Dynamic Instancing

다이나믹 인스턴싱은 장면에 추가된 물체를 자동으로 분류해서 동일한 물체가 여러 개 있는 경우 자동으로 인스턴싱을 통해 물체를 그리는 방식으로 '시작하며…' 챕터에서 소개한 'Refactoring the Mesh Drawing Pipeline for Unreal Engine 4.22' 동영상에서 소개된 용어입니다.

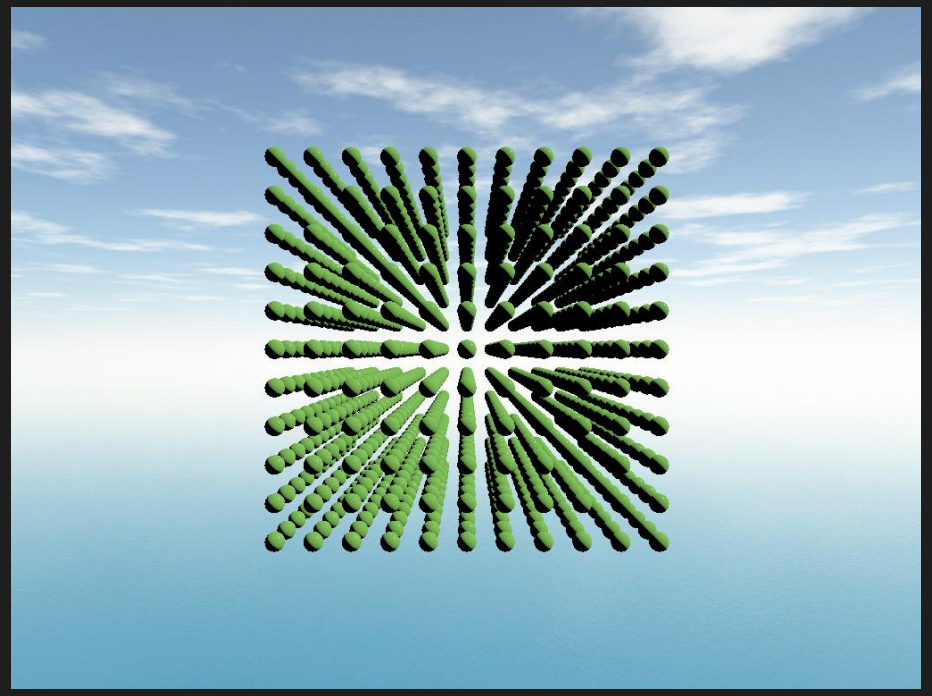
Auto Instancing이라고도 하는 것 같습니다.

이제부터는 인스턴싱을 자동으로 지원하는 환경을 위해서 어떤 작업이 필요했는지 살펴보겠습니다.

Dynamic Instancing

동일한 물체의 정의

우선 어디 까지를 동일한 물체로 취급할 것인지에 대한 정의가 필요합니다. 아래 스크린 샷 처럼 같은 모양의 구들도 서로 다른 위치에 그려야 하기 때문에 한번에 그려지는 물체에도 서로 다른 부분이 존재합니다.



Dynamic Instancing

동일한 물체의 정의

물체에 따라서 다를 수 있는 부분은 대표적으로 물체의 위치가 있을 수 있고 본 애니메이션이 필요한 메시라면 본의 행렬 값 등이 있겠습니다.

이와 같이 동일한 물체 간에 어떤 값이 서로 다를 수 있는지는 경우에 따라 다르게 규정할 수 있습니다.

현재 프로그램에는 고정된 모양의 스태틱 메시만 존재하는데 위치, 크기, 회전 변환을 제외하고 모든 값이 (재질, 메시 모양 등) 같아야 동일한 물체로 취급하고 있습니다.

Dynamic Instancing

동일한 물체의 정의

물체간 서로 다른 정보는 인스턴싱 중에 참조할 수 있도록 미리 그래픽 메모리에 전송해야 합니다.

이 정보는 렌더링 스레드에서 Scene에 물체를 추가할 때나 관련 데이터 변경 시 그래픽 메모리로 업로드하고 셰이더 코드에서는 인풋 어셈블러를 통해 인스턴스 데이터로 전달된 인덱스 값을 통해서 접근하도록 합니다.

```
VS_OUTPUT main( VS_INPUT input )
{
    VS_OUTPUT output = (VS_OUTPUT)0;

    PrimitiveSceneData primitiveData = GetPrimitiveData( input.primitiveId );
    output.worldPos = mul( float4(input.position, 1.0f), primitiveData.m_worldMatrix ).xyz;
    output.viewPos = mul( float4(output.worldPos, 1.0f), g_viewMatrix ).xyz;
    output.position = mul( float4(output.viewPos, 1.0f), g_projectionMatrix );
    output.normal = mul( float4(input.normal, 0.f), transpose( primitiveData.m_invWorldMatrix ) ).xyz;
    output.texcoord = input.texcoord;

    return output;
}
```

Dynamic Instancing

동일한 물체의 정의

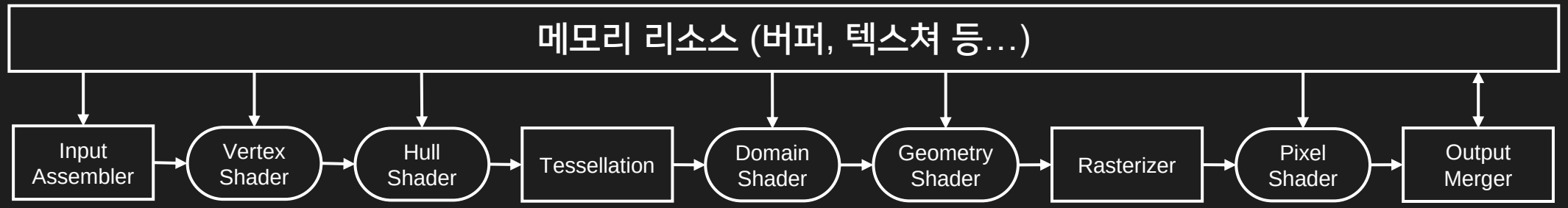
```
1  struct PrimitiveSceneData
2  {
3      matrix m_worldMatrix;
4      matrix m_invWorldMatrix;
5  };
6
7  static const int PRIMITIVE_SCENE_DATA_STRIDE = 8;
8
9  StructuredBuffer<float4> primitiveInfo : register( t0 );
10
11 PrimitiveSceneData GetPrimitiveData( uint primitiveId )
12 {
13     PrimitiveSceneData primitiveData;
14     uint baseOffset = primitiveId * PRIMITIVE_SCENE_DATA_STRIDE;
15
16     primitiveData.m_worldMatrix[0] = primitiveInfo[baseOffset + 0];
17     primitiveData.m_worldMatrix[1] = primitiveInfo[baseOffset + 1];
18     primitiveData.m_worldMatrix[2] = primitiveInfo[baseOffset + 2];
19     primitiveData.m_worldMatrix[3] = primitiveInfo[baseOffset + 3];
20
21     primitiveData.m_invWorldMatrix[0] = primitiveInfo[baseOffset + 4];
22     primitiveData.m_invWorldMatrix[1] = primitiveInfo[baseOffset + 5];
23     primitiveData.m_invWorldMatrix[2] = primitiveInfo[baseOffset + 6];
24     primitiveData.m_invWorldMatrix[3] = primitiveInfo[baseOffset + 7];
25
26     return primitiveData;
27 }
```

Dynamic Instancing

DrawSnapshot

동일한 물체의 기준을 정했다면 이제 물체를 분류하기 위한 모든 정보를 모아야 합니다.

DrawSnapshot은 분류를 위한 클래스로 어떤 물체를 그릴 때의 파이프라인 상태에 대한 스냅샷입니다.



그리기에 필요한 파이프라인 상태

Dynamic Instancing

DrawSnapshot

코드에서는 다음과 같습니다.

```
class DrawSnapshot
{
public:
    // Vertex buffer + Input layout
    VertexInputStream m_vertexStream;

    // Index buffer
    IndexBuffer m_indexBuffer;

    // Shader resource (Constant buffer + SRV + Sampler)
    aga::ShaderBindings m_shaderBindings;

    // Pipeline State (Shader, Rasterizer state, Depth Stencil state, Blend state)
    GraphicsPipelineState m_pipelineState;

    // For Draw method
    uint32 m_indexCount;
    uint32 m_startIndexLocation;
    uint32 m_baseVertexLocation;
};
```

DrawSnapshot만 있으면 이를 통해서 언제든 그리기 명령으로 변환할 수 있습니다.

Dynamic Instancing

DrawSnapshot

CommitDrawSnapshot 함수가 DrawSnapshot을 그리기 명령으로 변환합니다.

```
1  void CommitDrawSnapshot( aga::ICommandList& commandList, VisibleDrawSnapshot& visibleSnapshot,
   VertexBuffer& primitiveIds )
2  {
3      DrawSnapshot& snapshot = *visibleSnapshot.m_drawSnapshot;
4
5      // Set vertex buffer
6      VertexInputStream vertexStream = snapshot.m_vertexStream;
7      vertexStream.Bind( primitiveIds, 1, visibleSnapshot.m_primitiveIdOffset * sizeof( uint32 ) );
8
9      uint32 numVB = vertexStream.NumBuffer( );
10     aga::Buffer* const* vertexBuffers = vertexStream.VertexBuffers( );
11     const uint32* vertexOffsets = vertexStream.Offsets( );
12     commandList.BindVertexBuffer( vertexBuffers, 0, numVB, vertexOffsets );
13
14     // Set index buffer
15     aga::Buffer* indexBuffer = snapshot.m_indexBuffer.Resource( );
16     commandList.BindIndexBuffer( indexBuffer, 0 );
17
18     // Set pipeline state
19     commandList.BindPipelineState( snapshot.m_pipelineState.m_pso );
20
21     // Set shader resources
22     commandList.BindShaderResources( snapshot.m_shaderBindings );
23
24     if ( visibleSnapshot.m_numInstance > 1 )
25     {
26         commandList.DrawInstancing( snapshot.m_indexCount, visibleSnapshot.m_numInstance,
   snapshot.m_startIndexLocation, snapshot.m_baseVertexLocation );
27     }
28     else
29     {
30         commandList.Draw( snapshot.m_indexCount, snapshot.m_startIndexLocation,
   snapshot.m_baseVertexLocation );
31     }
32 }
```

Dynamic Instancing

DrawSnapshot

이제 동일 물체끼리 분류하는 작업만이 남았습니다. 이것은 DrawSnapshot을 정렬하는 것으로 간단하게 해결할 수 있습니다.

다만 DrawSnapshot은 모든 정보를 담고 있기 때문에 클래스의 크기가 매우 큽니다. 64bit에서는 기본 크기만 520Byte에 달합니다. 그리고 셰이더에 설정될 모든 리소스의 참조는 셰이더에 따라 가변적일 수 있기 때문에 더 늘어 날 수 있습니다.

따라서 DrawSnapshot 자체를 정렬 중에 비교하는 것은 좋지 않습니다.

Dynamic Instancing

DrawSnapshot

이를 해결하기 위해서 DrawSnapshot에 아이디를 부여하였습니다.
아이디는 다음과 같은 해시 자료구조를 통해서 부여합니다.

```
class CachedDrawSnapshotBucket
{
public:
    int32 Add( const DrawSnapshot& snapshot );
    void Remove( int32 id );

private:
    std::unordered_map<DrawSnapshot, size_t, DrawSnapshotDynamicInstancingHasher,
  DrawSnapshotDynamicInstancingEqual> m_bucket;
    SparseArray<DrawSnapshot> m_snapshots;
};

int32 CachedDrawSnapshotBucket::Add( const DrawSnapshot& snapshot )
{
    // 일단 해시 테이블에 추가를 시도한다.
    constexpr size_t dummy = 0;
    auto [iter, success] = m_bucket.emplace( snapshot, dummy );
    if ( success )
    {
        // 성공적으로 추가 했다면 아이디가 없으므로 아이디를 발급 받는다.
        size_t id = m_snapshots.Add( snapshot );
        iter->second = id;
        return static_cast<int32>( id );
    }

    return static_cast<int32>( iter->second );
}
```

Dynamic Instancing

DrawSnapshot

실제 DrawSnapshot 정렬코드는 다음과 같습니다.

```
std::sort( std::begin( snapshots ), std::end( snapshots ),
        []( const VisibleDrawSnapshot& lhs, const VisibleDrawSnapshot& rhs )
        {
            return lhs.m_snapshotBucketId < rhs.m_snapshotBucketId;
        } );
```

DrawSnapshot을 정렬하고 나면 이제 동일한 종류끼리 병합합니다.
이는 비교 후 인스턴스 개수를 늘리기만 하면 됩니다.

```
for ( size_t cur = 0, dest = cur + 1; cur < snapshots.size( ) && dest < snapshots.size( ); ++dest )
{
    if ( snapshots[cur].m_snapshotBucketId != -1 &&
        snapshots[cur].m_snapshotBucketId == snapshots[dest].m_snapshotBucketId )
    {
        ++snapshots[cur].m_numInstance;
    }
    else
    {
        cur = dest;
    }
}
```

Dynamic Instancing

DrawSnapshot

인스턴싱으로 그릴 때는 해당 인스턴스 개수만큼 배열을 이동하면서 그리기 명령으로 변환하면 됩니다.

```
for ( size_t i = 0; i < visibleSnapshots.size( ); )
{
    CommitDrawSnapshot( *commandList, visibleSnapshots[i], primitiveIds );
    i += visibleSnapshots[i].m_numInstance;
}
```

더 자세한 코드를 보고 싶으시다면...

- https://github.com/xtozero/SSR/tree/multi-thread

참고자료

- UE4 4.24