

STRENGTH

... ..

GDC

March 20-24, 2023
San Francisco, CA

Charles Tremblay | Engineering Director



8234299 -AXRFG 0001

REDengine 1

REDengine 2 Console Support

REDengine 3 Open World Support

REDengine 4



The Witcher 2: Assassins of Kings



The Witcher 2: Assassins of Kings Enhanced Edition



The Witcher 3: Wild Hunt



Cyberpunk 2077



March 20-24, 2023
San Francisco, CA



CYBERPUNK REQUIREMENTS



- **Night City – Living metropolis in a dystopian future**
 - 16x16km world and vertical
 - High density
 - No loading screen
- **Vehicles**
 - 140km/hr
- **Scalability**
 - Various platform support
 - From Xbox One to High-End PC

RED ENGINE PILLARS

- Engine / Gameplay agnostic of Editor/Tools code
 - RPC backend ↔ Editor
- Systems Scalability
 - Maximize platform hardware utilization
 - Adapt according to game state or current quest
- Quest is King



GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

RED ENGINE RULES



No code can actively wait for anything



Code should assume it never runs in isolation

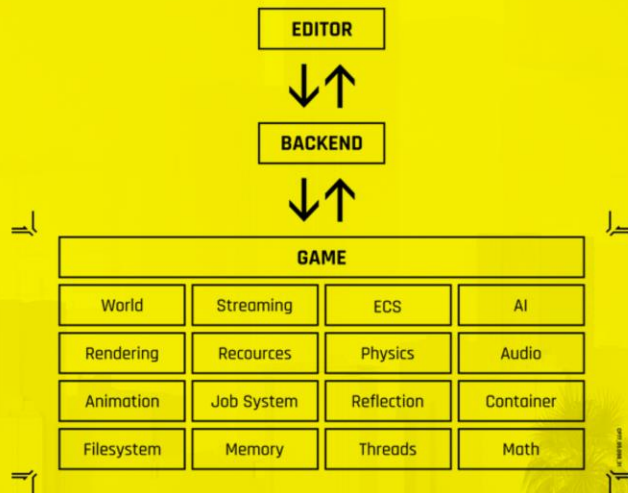


Runtime memory allocation limited to ≤ 512 bytes



No STL containers, Exception, RTTI

ENGINE – DIAGRAM



AGENDA

- Memory
- Job Systems
- Resources & IO
- Graphics
- World & Streaming
- ECS
- Systems
- Frame & Performance
- Conclusion

GDC March 20-24, 2022
San Francisco, CA



MEMORY MANAGEMENT



MEMORY — ALLOCATOR FOR SPECIFIC NEEDS

- Slab
- TLSF
- Fixed Size
- Linear
- Buddy
- Stack
- Job & Frame Allocator
- And more ... !



GDC

March 20-24, 2022
San Francisco, CA



MEMORY — DEFAULT ALLOCATOR

- Custom Slab allocator with explicit thread registration for $\leq 512b$ allocations
 - From Witcher 3 experiences: 75% + of all allocations are less than 512b
 - Even distribution across our job/task threads
 - Good locality
 - 10ns on PS4
 - Average waste of 6% per thread
- TLSF allocator for allocations between 512b a 512kb
- "BigSize" allocator for allocations $> 512kb$

MEMORY — EASY TO EXTEND

- Allocator code does not have to reside in memory project
- Clear and minimal static interface to fulfill
- All utilities provided by memory system are available

```
struct SimpleAllocatorMetrics{};

class SimpleAllocator
{
public:
    RED_MEMORY_DECLARE_ALLOCATOR( SimpleAllocator, SimpleAllocatorMetrics, 16 );

    red::memory::Block Allocate( uint32_t size );
    red::memory::Block AllocateAligned( uint32_t size, uint32_t alignment );
    red::memory::Block Reallocate( red::memory::Block& block, uint32_t size );
    red::memory::Block ReallocateAligned( red::memory::Block& block, uint32_t size, uint32_t alignment );
    void Free( red::memory::Block& block );
    uint64_t GetBlockSize( uint64_t block ) const;
    void SerializeMetrics( red::memory::Serializer& serializer );
};
```

MEMORY — EASY TO USE & UNDERSTAND

- Fully documented
- Code consistency
- Easy to read!

```
////////////////////////////////////////////////////////////////////
// 3.2 Allocating from a specific Allocator
// Like with the new operator replacement (RED_NEW), you can also provide an allocator explicitly.
// For example:
//
void Sample_3_2_Allocator()
{
    const int32_t allocatorBufferSize = RED_KILO_BYTE( 64 );
    void * allocatorBuffer = RED_ALLOCATE( red::PoolDefault, allocatorBufferSize );
    const red::memory::StaticTLSAllocatorParameter param = { allocatorBuffer, allocatorBufferSize };
    red::memory::StaticTLSAllocator allocator;
    allocator.Initialize( param );

    void * buffer = RED_ALLOCATE( allocator, 128 );
    void * reallocBuffer = RED_REALLOCATE( allocator, buffer, 256 );
    RED_FREE( allocator, buffer );
    RED_FREE( allocator, reallocBuffer );
}
```

MEMORY — POOLS

- All allocations needs to be associated to a Pool
- Pools define budgets
- Pools can be parented

```
RED_MEMORY_POOL( PoolAI_Behaviour, red::memory::DefaultAllocator, AI_API );

void MemoryPoolSnippet()
{
    auto& allocator = red::memory::AcquireDefaultAllocator();
    RED_INITIALIZE_MEMORY_POOL( PoolAI_Behaviour, AI::PoolAI, allocator, RED_MEGA_BYTE( 7 ) );

    auto * scalar = RED_NEW( int32_t, PoolAI_Behaviour )( 123 );
    RED_DELETE( scalar, PoolAI_Behaviour );

    void* buffer = RED_ALLOCATE( PoolAI_Behaviour, RED_KILO_BYTE( 64 ) );
    RED_FREE( PoolAI_Behaviour, buffer );

    red::DynArray< int32_t > myArray( PoolAI_Behaviour() );
    myArray.Reserve( 16 );
}
```

MEMORY – METRICS & TRACKING

- All memory allocations can be tracked
- Report can be use for automated tools

name	usage	inclusive	exclusive count	exclusive	exclusive count	peak	budget	inc. alloc. bytes/frame	inc. alloc. count/frame	debug alloc	control to parent	handle	parenting
Pooled Pools													
PooledCPU	100%	2.84 GB	8412737	0 B	0	0 B	1.54 GB	4.60 MB	11061	no	yes	0x1130510F	yes
Children Pools													
PooledCpuFeatures	14%	1.64 KB	3	1.64 KB	3	1.64 KB	0.07 KB	0 B	0	no	yes	0x71a03009	yes
Pooled	0%	122.11 MB	134289	816.89 KB	3844	817.00 KB	140.01 MB	328 B	8	no	yes	0x0050A41A	yes
PooledCountdown	0%	173.16 MB	129239	10.04 MB	10913	10.04 MB	195.01 MB	2.24 KB	41	no	yes	0x77862613	yes
PooledAes	100%	196.65 MB	892610	22.71 MB	1310	26.70 MB	195.01 MB	69.10 KB	81	no	yes	0x871a00af	yes
PooledCompression	0%	0 B	0	0 B	0	0 B	1.02 KB	0 B	0	no	yes	0x1828342E	yes
PooledAesCbc	148%	1.48 MB	8950	1.48 MB	8950	1.56 MB	1.00 MB	0 B	0	no	yes	0x10424676	yes
PooledAesCbcAesCtr	0%	7.40 KB	1	7.40 KB	1	82.06 KB	82.00 MB	23.25 KB	22	no	yes	0x71a0b0fa	yes
PooledAesCtr	148%	793.08 MB	444180	64.18 MB	264347	81.05 MB	432.01 MB	101.44 KB	8729	no	yes	0x71a0b0fa	yes
PooledAesCtr	0%	0 B	0	0 B	0	215.00 KB	10.00 MB	48.10 KB	102	no	yes	0x81a7a1a8	yes
PooledAesCtrAesCtr	0%	432 B	1	432 B	1	432 B	1.00 MB	0 B	0	no	yes	0x1a432061	yes
PooledAesCtrAesCtr	127%	249.11 MB	1617056	0 B	0	0 B	145.01 MB	6.80 KB	91	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtr	0%	52.13 KB	28	24 B	1	360 B	1.00 MB	0 B	0	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtr	0%	0 B	0	0 B	0	405.53 KB	1.02 KB	0 B	0	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtr	0%	391.61 KB	7333	391.61 KB	7333	589.17 KB	1.00 MB	240 B	30	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtr	0%	984 B	3	0 B	0	0 B	220.01 MB	0 B	0	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtrAesCtr	0%	0 B	0	0 B	0	0 B	68.00 MB	0 B	0	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtrAesCtrAesCtr	0%	0 B	0	0 B	0	0 B	2.00 MB	0 B	0	no	yes	0x71a071a0	yes
PooledAesCtrAesCtrAesCtrAesCtrAesCtrAesCtr	0%	93.72 MB	101149	4.32 MB	1293	4.37 MB	100.00 MB	99.00 KB	68	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtr	252%	716.36 MB	779634	0 B	0	0 B	285.00 MB	3.10 MB	1051	no	yes	0x0090AC1F	yes
PooledAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtr	20%	5.47 MB	25628	77.36 KB	927	77.36 KB	24.00 MB	3.04 KB	12	no	yes	0x1a432061	yes
PooledAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtrAesCtr	0%	10.94 MB	79612	0 B	0	0 B	15.00 MB	181.88 KB	202	no	yes	0x0090AC1F	yes

GDC

March 30–31, 2022
San Francisco, CA

CD PROJEKT RED

8234299 - AXRC6 0001

MEMORY – BUDGETS

CPU - 1.5gb

Rendering – 300mb
Animation – 200mb
Audio – 200mb
Streaming – 160mb
Gameplay – 160mb
AI – 140mb
Resources – 100mb
Physics – 100mb
Archives – 64mb
UI – 55mb

GPU - 3gb

Texture Generic – 700mb
Texture Multilayer – 350mb
Render Targets – 640mb
Mesh – 700mb
GI – 300mb



March 20-24, 2022
San Francisco, CA



CD PROJEKT RED

17



March 20-24, 2023
San Francisco, CA



- Each content teams was able to monitor their budget
- It was up to content team to balance things out
- Various heat map were generated also to assist finding hotspot

CONTENT BUDGET

18

194mb Environment Textures	33mb Weapon Textures	33mb Vehicle Textures	35mb FX Textures	221mb Code Based Textures	72mb Surface Textures
107mb Environment Meshes	50mb Weapon Meshes	50mb Vehicle Meshes	10mb FX Meshes	151mb Code Based meshes	210mb Multilayer library
42mb Environment MLMasks	10mb Weapon MLMasks	20mb Vehicle MLMasks	5mb FX MLMasks	36mb Code Based MLMasks	50mb Microblends
217mb Character Textures	30mb UI Textures	30mb Vehicle Proxies			
57mb Character Meshes	50mb UI Advertisements				
27mb Character MLMasks	10mb Videos				

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED



There was a good question that I feel I failed to explain correctly.

How Jobs are better than custom thread.

I can answer this with a real example.

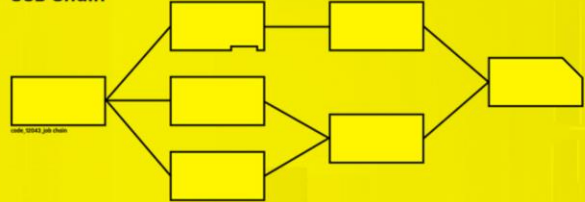
On Witcher3, the render thread will issue the culling work to be done on another thread. In meantime, render thread will continue to do its work but at one point it will need to wait for the result of the culling. With some luck it's finished, and result is available. However, very often it is not, causing the render thread to completely stop until result is finally available. In some case this cause up to 5ms completely wasted on render thread.

Now with proper job chain, this problem never occur. See Graphics part about render graph ☺

JOB SYSTEM—REQUIREMENTS

- Unshackle Main & Render thread
- Everything should be a job
No more custom threads
- Easy to build a job chain
- Easy to write continuation jobs
- Easy to use

Job Chain



JOB BUILDER

- Main utility to dispatch and manage jobs chain
- Allows to create complex job chain
- Used by every single system

JOB BUILDER — SIMPLE JOB CHAIN

DispatchJob creates dependant job by default



```

class Object
{
public:
    void FirstJob();
    void SecondJob();
    void ThirdJob();
};

void SimpleJobDependencySnippet( const red::SharedPtr< Object >& object )
{
    job::Builder builder( job::Priority::CriticalPath );
    builder.DispatchJob( "First_Job", [object]( const job::RunContext& ) {
        object->FirstJob();
    } );

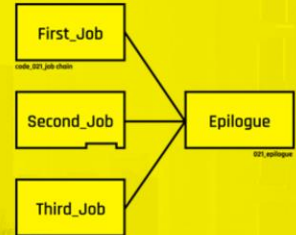
    builder.DispatchJob( "Second_Job", [object]( const job::RunContext& ) {
        object->SecondJob();
    } );

    builder.DispatchJob( "Third_Job", [object]( const job::RunContext& ) {
        object->ThirdJob();
    } );
}
  
```

JOB BUILDER – PARALLEL JOB CHAIN

- DispatchJob can create jobs to be run in parallel
- DispatchParallelForJob can also be used

```
void ParallelJobWithEpilogueJobSnippet( const red::SharedPtr< Object >& object )
{
    Job::Builder builder( Job::Priority::CriticalPath );
    builder.DispatchJob< Job::Fence::None >( "First_Job", [object]( const Job::RunContext& ) {
        object->FirstJob();
    } );
    builder.DispatchJob< Job::Fence::None >( "Second_Job", [object]( const Job::RunContext& ) {
        object->SecondJob();
    } );
    builder.DispatchJob< Job::Fence::None >( "Third_Job", [object]( const Job::RunContext& ) {
        object->ThirdJob();
    } );
    builder.DispatchFenceExplicitly();
    builder.DispatchJob( "Epilogue_Job", [object]( const Job::RunContext& ) {
        object->EpilogueJob();
    } );
}
```



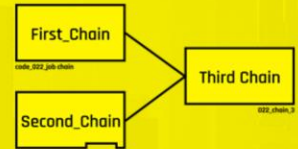
JOB BUILDER — HOW TO LINK JOB CHAIN

- Job Builder can be linked to existing job chain
- Subsequent job dispatches follow regular rules

```
void LinkingJobGraphShippet( const red::SharedPtr< Object >& object )
{
    job::Counter counter;
    job::Builder firstBuilder( job::Priority::CriticalPath );
    firstBuilder.DispatchJob( "First_Job", [object]( const job::RunContext& ) {
        object->FirstJob();
    } );

    counter += firstBuilder.ExtractWaitCounter();
    job::Builder secondBuilder( job::Priority::CriticalPath );
    secondBuilder.DispatchJob( "Second_Job", [object]( const job::RunContext& ) {
        object->SecondJob();
    } );

    counter += secondBuilder.ExtractWaitCounter();
    job::Builder thirdBuilder( job::Priority::CriticalPath );
    thirdBuilder.DispatchWait( counter );
    thirdBuilder.DispatchJob( "Third_Job", [object]( const job::RunContext& ) {
        object->ThirdJob();
    } );
}
```



JOB BUILDER — HOW TO DO A CONTINUATION JOB

25

Job can be dispatched as a continuation of the current job

```
void ContinuationJobSnippet( const red::SharedPtr< Object >& object )
{
    job::Builder builder( job::Priority::CriticalPath );
    builder.DispatchJob( "First_Job", [ object ]( const job::RunContext& context )
    {
        object->FirstJob();
        job::Builder builder( context );
        builder.DispatchJob( "Second_Job", [ object ]( const job::RunContext& context )
        {
            object->SecondJob();
            job::Builder builder( context );
            builder.DispatchJob( "Third_Job", [ object ]( const job::RunContext& )
            {
                object->ThirdJob();
            } );
        } );
    } );
}
```

JOB SYSTEM – CANCELLING JOBS?

- It is not possible to safely cancel a complete job chain
- However it can be done on the user side

CYBERPUNK
2077

IO & RESOURCE MANAGEMENT



IO & RESOURCE MANAGEMENT — REQUIREMENTS

- Lockless resource loading request
- No IO if resource is already loaded
- Fully compatible with job system
- Dependant resources can be merged up and duplicated
- Only one instance of any resource can be alive at any time
- No sync operation is allowed
- IO request dependency should be known up front



March 20-26, 2022
San Francisco, CA



RESOURCE LOADER

- Resource loading request can be made virtually anywhere
- You cannot actively wait for completion
- With great power comes great responsibility

```
void ResourceLoadingSnippet( res::ResourceLoader* resourceLoader )
{
    res::IssueLoadingRequestParameter param;
    param.path = RED_CONST_RESOURCEPATH( "base\\gameplay\\devices\\vending_machines\\vending_machine_1.ent" );
    param.priority = io::eAsyncPriority_Background;

    res::ResourceTokenHandle token = resourceLoader->IssueLoadingRequest( param );
    if( token->IsLoaded() )
    {
        // Resource was most likely already loaded. You can use right away !
        auto& resource = token->GetResource();
    }
    else if( token->HasFailed() ) // Resource loading request failed?
    {
        auto errorType = token->GetError(); // It could be invalid path, extension or resource wasn't found, etc..
    }
    else
    {
        /* Resource loading request on going! */
    }

    // If refcount goes to 0, resource will be schedule for unload, or loading request will be cancelled.
    token.Reset();
}
```

CONTINUATION JOB WHEN RESOURCE LOADED

30

- It is possible to link job to a resource-loading job chain
- But it's important to validate result of request!

```
void ResourceLoadingAndContinuationJobSnippet( res::ResourceLoader* resourceLoader )
{
    res::IssueLoadingRequestParameter param;
    param.path = RED_CONST_RESOURCEPATH( "base\\gameplay\\devices\\vending_machines\\vending_machine_1.ent" );
    param.priority = 10::eAsyncPriority_Background;

    res::ResourceTokenHandle token = resourceLoader->IssueLoadingRequest( param );
    job::Builder builder( job::Priority::Latent );
    builder.DispatchWait( token->GetWaitCounter() );
    builder.DispatchJob( "OnResourceLoadingRequestCompleted", [token]( const job::RunContext& ) {

        if ( !token->HasFailed() )
        {
            // Resource is ready! It can be use safely.
            auto resource = token->GetResource();
            // Note! if resource ownership is not taken, token could trigger unload when refcount will go to 0.
        }
    } );
}
```

RESOURCE REQUEST UNDER HEAVY CONTENTION

- Multiple concurrent requests
- Resources could be requested to be unloaded at the same time
- Avoid locks as much as possible

RESOURCE REQUEST UNDER HEAVY CONTENTION

32

```
ResourceTokenHandle ResourceLoader::TrySchedulingLoadingJobs( const res::ResourcePath resolvedPath, const IssueLoadingRequestParameter& param )
{
    ResourceTokenHandle token;
    Job::CompletionDeferral deferral;
    bool isNewToken = false;

    RED_SCOPE_SHARED_LOCK( m_resourceTokenLock ); // SHARED LOCK: Try to acquire already created token
    token = TryAcquireResourceToken_NoLock( resolvedPath );

    if( !token ) {
        // Create new resource token outside the lock.
        auto tokenDeferralPair = CreateResourceToken( resolvedPath );

        RED_SCOPE_LOCK( m_resourceTokenLock ); // EXCLUSIVE LOCK: Try to create new token
        token = TryAcquireResourceToken_NoLock( resolvedPath ); // Got lock. Did someone beat us to it ?
        if( !token ) { // If we got here, we have the lock, no one managed to beat us to it also. Insert safely
            token = std::move(tokenDeferralPair.first);
            token->Internal_SetPriority( param.priority );
            deferral = std::move(tokenDeferralPair.second);
            isNewToken = true;
            m_resourceTokenDictionary[ resolvedPath ] = token;
        }

        if( !isNewToken ) {
            // If we got here, we have token and we created it. First, is the Resource already loaded?
            const Thandle< CResource > resource = m_resourceBank->FindResource( resolvedPath );
            if( !resource ) {
                ScheduleLoadingJob( token, std::move( deferral ), resolvedPath, param ); // No? Kickstart loading job!
            }
            else {
                token->Internal_AssignLoadedResource( resource, std::move( deferral ) ); // Yes? Assign to token.
            }
        }

        return token;
    }
}
```



March 20-24, 2022
 San Francisco, CA





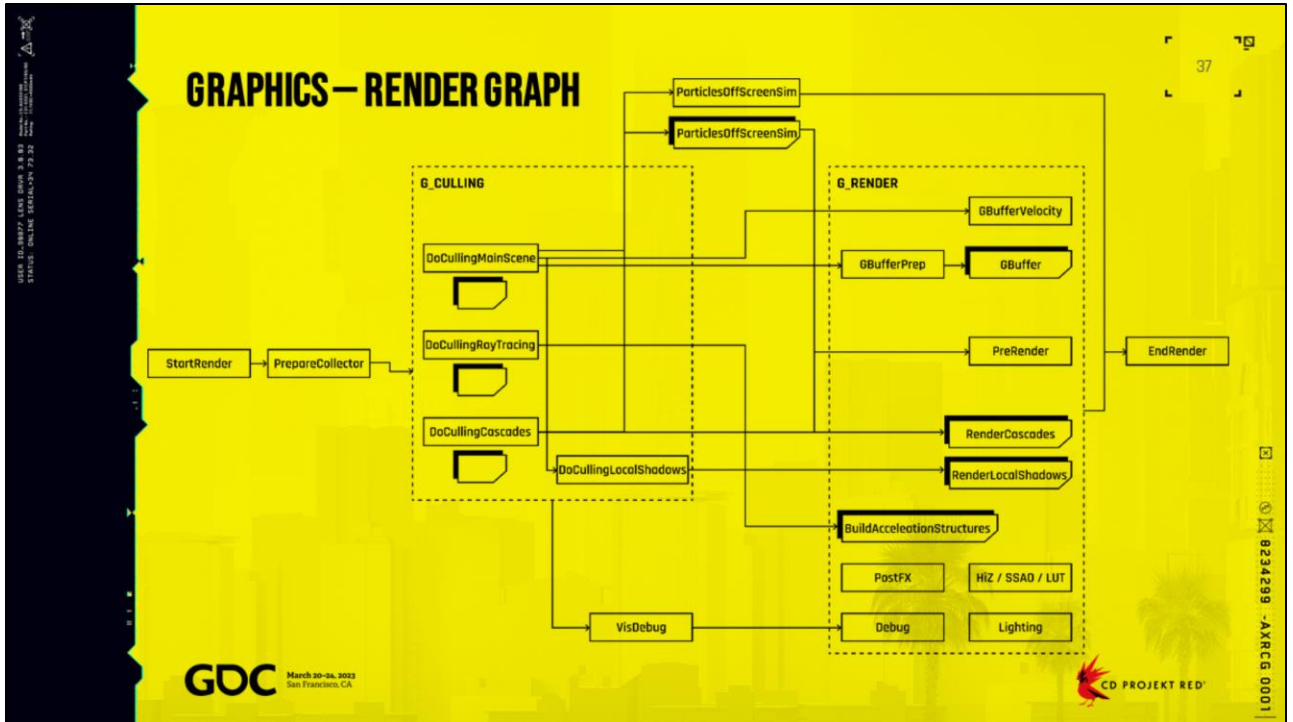
Very good presentation by Tim Green, SIGGRAPH 2021



GRAPHICS—RENDER GRAPH

- Complete Render Graph is declared in code
- Multiple graphs for different use cases
- Runs across threads at lower priority than game jobs
- Async Compute
 - Limited to fork-join model
 - SSAO, Hi-Z generation, Acceleration Structure etc...

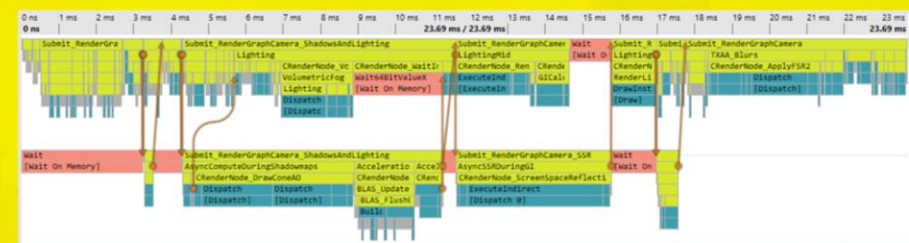
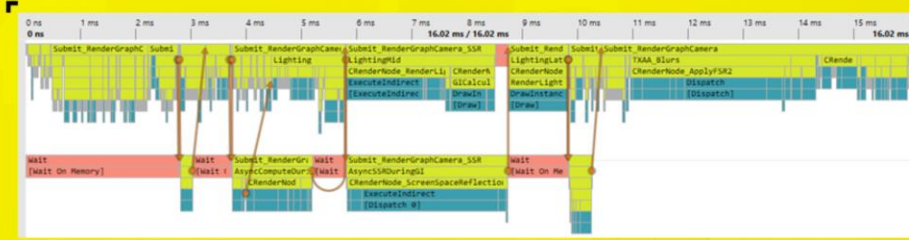




See Samples section at the end. I've added full code example on how it looks like in code

GRAPHICS—GPU FRAME

38



GDC

March 30-31, 2022
 San Francisco, CA

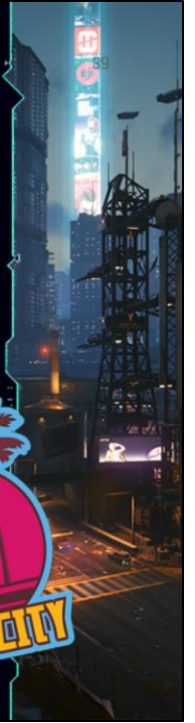
CD PROJEKT RED

8234299 -AXRCG 0001

USER: 12.18077 LAST: 0000 3.4.82 100% 100% 100%
STATUS: ONLINE SERIAL: 00000000000000000000000000000000

CYBERPUNK
2-0-7-7

WORLD & STREAMING





CYBERPUNK 2077 IN NUMBERS

- One world, 16x16 km
- 15 million + Objects / Nodes / Entities
- 30 million + Foliage instances
- 31-38 main quests, 80 side quests, 74 gig quests
- 100+ NCPD Scanner Hustles, hidden gems, mini stories, & other small content pieces
- 2200+ Quest prefabs



HOW WERE PRECISION ISSUES RESOLVED?

- **Fixed Point is your friend!**
 - Int32, 15 / 17 for our World Position
- **Physics scene origin needs to be updated**
 - Every 1024m from last origin update
 - Every physics proxy needs to be updated
- **Camera translation as an origin for rendered objects**



WORLD NODE & NODEINSTANCE

- NodeInstances are the units that are streamed in
- NodeInstances are NOT updated directly
- Node is the payload provided to an instance when streaming in
- Node can be shared to multiple node instances

```
struct SimpleMeshNodeInstance : public NodeInstance
{
    RTTI_DECLARE_TYPE( SimpleMeshNodeInstance );
    virtual bool OnInitialize( const Context& context ) override final;
    virtual void OnAttach( RuntimeScene& scene ) override final;
    virtual void OnDetach( RuntimeScene& scene ) override final;

    RenderProxyPtr m_renderProxy;
    MeshResourceHandle m_loadedMesh;
    ResourceTokenHandle m_loadingToken;
};

struct SimpleMeshNode : public Node
{
    RTTI_DECLARE_TYPE( SimpleMeshNode );
    virtual const rtti::ClassType* GetInstanceClass() const override final
    {
        return ClassID< SimpleMeshNodeInstance >();
    }
    TResAsyncRef< Mesh > mesh;
};
```

WORLD NODE & NODEINSTANCE

43

```
bool SimpleMeshNodeInstance::OnInitialize( const Context& context ) {
    const auto meshNode = GetSourceNodeAs< SimpleMeshNode >();
    m_loadingToken = meshNode->mesh.IssueLoadingRequest( io::eAsyncPriority_Streaming );
    context.builder.DispatchWait( m_loadingToken->GetWaitCounter() );

    const WeakHandle< SimpleMeshNodeInstance > weakHandle = HandleFromThis< SimpleMeshNodeInstance >();
    context.builder.DispatchJob( "OnMeshLoaded", [weakHandle, this]( const job::RunContext& ) {

        if( auto handle = weakHandle.ToHandle() ) {

            m_loadedMesh = red::StaticCast< Mesh >( m_loadingToken->GetResource() );

            RenderProxyMeshComponentInitData data;
            data.m_renderMesh = m_loadedMesh->GetRenderResource();
            data.m_localToWorld = { GetInitialTransform(), GetInitialScale() };
            data.m_boundingBox = data.m_localToWorld.TransformBox( m_loadedMesh->GetBoundingBox() );
            data.m_type = RPT_Mesh;
            RenderProxyInitInfo initInfo;
            initInfo.m_componentData = &data;

            m_renderProxy = GetSystem< RuntimeSystemRendering >()->CreateAndRegisterRenderProxy( initInfo, true );

        }
    });
    return true;
}

void SimpleMeshNodeInstance::OnAttach( RuntimeScene& scene ) {
    m_loadingToken.Reset();
}

void SimpleMeshNodeInstance::OnDetach( RuntimeScene& scene ) {
    GetSystem< RuntimeSystemRendering >()->DissolveAndDestroyRenderProxy( std::move( m_renderProxy ) );
}
```

GDC

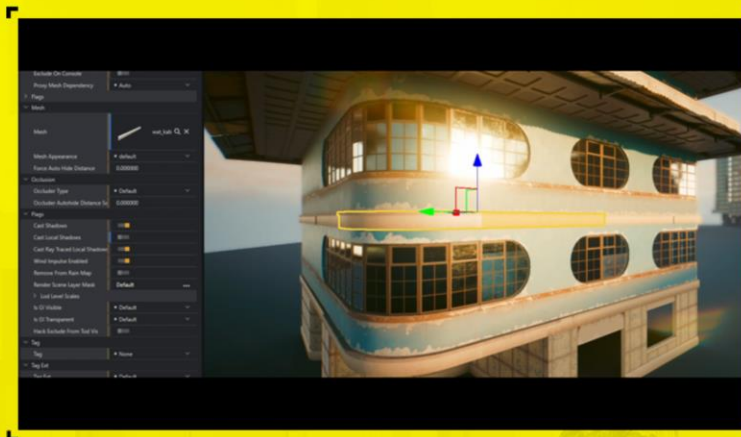
March 20-24, 2022
 San Francisco, CA

CD PROJEKT RED

8234299 -AXRCG 0001

PREFABS EDITOR

44



GDC

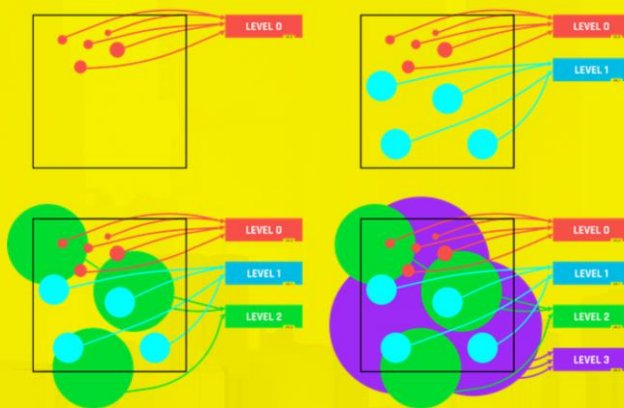
March 20-26, 2022
San Francisco, CA

CD PROJEKT RED

8234299 - AXRCG 0001

STREAMING GRID—DEVELOPMENT

45



- World is split into sector of a 256m cube
 - Multiple sectors can have the same position
 - Exterior, Interior, quests
 - Streaming range
- Runtime cost of node payload on non-optimized grid was around 200-250mb

GDC

March 20-24, 2022
San Francisco, CA



STREAMING GRID—OPTIMIZED

- Sector dimensions are much smaller
 - 64m cube for exteriors
 - 32m cube for interiors
- Sectors are rebalanced to eliminate “almost empty” sectors
 - Nodes will always “move up” to higher level sector
- Quest sectors are now merged into a single sector per quest
- Resources are now embedded in sectors
 - Minimap
 - Simplified far distance mesh (we called them proxy)
 - Foliage
- Instancing nodes are generated replacing Mesh nodes using same mesh
- Runtime cost of node payload on optimized grid was around 60-80mb



March 20-24, 2022
San Francisco, CA



RUNTIME STREAMING PROCESS

1. **Compute which sector needs to be loaded / unloaded**
 - a. Request Async Load on sector in range
 - b. Remove sector from grid that needs to be unloaded
 - c. Add loaded sector to grid
2. **Compute which nodes that need to be streamed-in / streamed-out**
 - a. Request each node in range to start streaming
 - b. Cancel streaming for nodes that aren't in range anymore
 - c. Accumulate nodes that are ready to be attached / detached
3. **Attach / detach nodes**

COMPUTE SECTORS IN RANGE

48

```
__m128 xxxx = _mm_loadl_ps( &position.X ), yyyy = _mm_loadl_ps( &position.Y ), zzzz = _mm_loadl_ps( &position.Z );
uint32 index = 0;
while( index + 8 < sectorCount )
{
    char firstMask = 0;
    char secondMask = 0;
    {
        __m128 vectorMinX = _mm_load_ps( streamMinX + index ); __m128 compareMinX = _mm_cmplt_ps( xxxx, vectorMinX );
        __m128 vectorMinY = _mm_load_ps( streamMinY + index ); __m128 compareMinY = _mm_cmplt_ps( yyyy, vectorMinY );
        __m128 vectorMinZ = _mm_load_ps( streamMinZ + index ); __m128 compareMinZ = _mm_cmplt_ps( zzzz, vectorMinZ );

        __m128 vectorMaxX = _mm_load_ps( streamMaxX + index ); __m128 compareMaxX = _mm_cmpgt_ps( xxxx, vectorMaxX );
        __m128 vectorMaxY = _mm_load_ps( streamMaxY + index ); __m128 compareMaxY = _mm_cmpgt_ps( yyyy, vectorMaxY );
        __m128 vectorMaxZ = _mm_load_ps( streamMaxZ + index ); __m128 compareMaxZ = _mm_cmpgt_ps( zzzz, vectorMaxZ );

        __m128 resultX = _mm_or_ps( compareMinX, compareMaxX );
        __m128 resultY = _mm_or_ps( compareMinY, compareMaxY );
        __m128 resultZ = _mm_or_ps( compareMinZ, compareMaxZ );
        __m128 results = _mm_or_ps( resultX, resultY );
        firstMask = ~_mm_movemask_ps( _mm_or_ps( results, resultZ ) );
    }

    index += 4;

    // repeat same code than above for the secondMask. Omitted for space reason.

    // Combine result into one byte, assign to bitset.
    char combinedMask = secondMask << 4;
    combinedMask |= (firstMask & 0xf);
    *sectorMaskStream |= combinedMask;
    ++sectorMaskStream;
    index += 4;
}
```

GDC

March 30-31, 2022
San Francisco, CA

CD PROJEKT RED

Orbis number
700us simple point contains in box
300us AOS
150us SOA

COMPUTE NODES IN RANGE

49

```
const UInt32 c_proxyStep = 16 * 1024;
for( UInt32 firstProxyIndex = 0, end = proxyCount; firstProxyIndex < end; firstProxyIndex += c_proxyStep )
{
    const UInt32 proxyCount = std::min( c_proxyStep, end - firstProxyIndex );
    builder.DispatchJob( job::Fence::None >{ "StreamingCollectNodeParallel", [0]( const job::RunContext& context ) {

        const float* streamX = m_x + firstProxyIndex, streamY = m_y + firstProxyIndex, streamZ = m_z + firstProxyIndex;
        const float* streamRadius2 = m_r2 + firstProxyIndex;
        const __m128 streamingDistanceScale128 = _mm_set1_ps( distanceMultiplier );
        __m128 xxxx = _mm_load_ps( &position.X ), yyyy = _mm_load_ps( &position.Y ), zzzz = _mm_load_ps( &position.Z );
        UInt64 maskStream = outMask + ( firstProxyIndex / 8 );
        UInt32 index = 0;

        while( index + 8 <= proxyCount )
        {
            char firstMask = 0, secondMask = 0;

            __m128 vectorX = _mm_sub_ps( _mm_load_ps( streamX + index ), xxxx ); __m128 vectorX2 = _mm_mul_ps( vectorX, vectorX );
            __m128 vectorY = _mm_sub_ps( _mm_load_ps( streamY + index ), yyyy ); __m128 vectorY2 = _mm_mul_ps( vectorY, vectorY );
            __m128 vectorZ = _mm_sub_ps( _mm_load_ps( streamZ + index ), zzzz ); __m128 vectorZ2 = _mm_mul_ps( vectorZ, vectorZ );
            __m128 result = _mm_add_ps( _mm_add_ps( vectorX2, vectorY2 ), vectorZ2 );
            __m128 vectorRadius2 = _mm_mul_ps( streamingDistanceScale128, _mm_load_ps( streamRadius2 + index ) );
            firstMask = _mm_movemask_ps( _mm_sub_ps( result, vectorRadius2 ) );

            index += 4;
            // repeat same code than above for the secondMask. Omitted for space reason.

            char combinedMask = secondMask << 4;
            combinedMask |= ( firstMask & 0xf );
            *maskStream = combinedMask;
            ++maskStream;
            index += 4;
        }
    } );
}
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

More or less 50us per job on PS4



ECS— RULES

- Reserved for more complex composition
- Can be spawned at runtime
- Spawn cannot be sync
- Cannot actively wait on spawn completion
- Visuals should be decoupled from logic if possible
- Entity / component update logic should be managed by proper systems
- Entity / component cannot communicate directly to other instance



March 20-24, 2022
San Francisco, CA



ECS— COMMUNICATION

- Events are the main communication method
- Events are safe to use at any point of the frame
- Events are broadcast only during specific frame times
- They can be sent to/from code or script
- Rely heavily on reflection

GDC

March 20-26, 2022
San Francisco, CA

 CD PROJEKT RED

ECS—EVENT



```

struct PhysicalImpulseEvent : public red::Event
{
    RTTI_DECLARE_TYPE( PhysicalImpulseEvent );
    uint32_t m_bodyIndex;
    Vector3 m_worldImpulse;
    Vector3 m_worldPosition;
    float m_radius;
    uint32_t m_shapeIndex;
    physics::ProxyID m_proxyId;
};

RTTI_BEGIN_TYPE( MeatBag );
    RTTI_PARENT_TYPE( game::Object );
    RTTI_PROPERTY_CATEGORY( "Physics" );
    RTTI_PROPERTY( m_kinematicBodyBoneName ).editable();
    RTTI_PROPERTY( m_bagBodyBoneName ).editable();
    RTTI_PROPERTY( m_physicalComponentName ).editable();
    RTTI_PROPERTY( m_bagHitComponentName ).editable();
    RTTI_PROPERTY( m_bagDestroyComponentName ).editable();
    RTTI_PROPERTY_CATEGORY( "Effects" );
    RTTI_PROPERTY( m_destructionEffectName ).editable();
    RTTI_PROPERTY( m_jiggleEffectName ).editable();
    RED_EVENT_CONNECTOR( OnSetup );
    RED_EVENT_CONNECTOR( OnControl );
    RED_EVENT_CONNECTOR( OnPhysicalImpulse );
RTTI_END_TYPE();

```

ECS—EVENT

54

```
void MeatBag::OnPhysicalImpulse( const PhysicalImpulseEvent& evt )
{
    if( evt.m_bodyIndex == m_bagBodyIndex ) {
        ent::SpawnEffectSetup effectSetup;
        if( --m_hitPoints > 0 ) {
            effectSetup.effectName = m_jiggleEffectName;
        }
        else {
            m_physicalComponent->ToggleCollisions( false, m_bagBodyIndex );
            m_physicalComponent->ToggleQueries( false, m_bagBodyIndex );
            m_bagHitComponent->Enable( false );
            m_bagDestroyComponent->SetForceInitAsVisible( true );
            m_bagDestroyComponent->Enable( true );
            m_bagDestroyComponent->SetForceInitAsVisible( false );
            effectSetup.effectName = m_destructionEffectName;
        }

        QueueEvent( CreateHandle< ent::SpawnEffectEvent >( effectSetup ) )
    }
}
```

- SYM

ECS—ASYNC SPAWNING

56

```
const RuntimeScene* scene = AcquireRuntimeScene();
RuntimeSystemEntity* entitySystem = scene->GetSystem< RuntimeSystemEntity >();
ent::EntitySpawnService* spawnService = entitySystem->AcquireSpawnService();

ent::EntityStaticSpawnContext context = {
    m_entityTemplate.GetPath(),
    m_transform,
    m_globalId,
    m_appearanceName,
    m_instanceData,
    editorService,
    ent::EntityLODInitialSetup( m_entityLod ),
    m_ioPriority
};

ent::EntitySpawnTokenHandle token = spawnService->SpawnStaticEntity( context );
builder.DispatchWait( token->GetWaitCounter() );

builder.DispatchJob( "OnSpawnReady", [token, entitySystem]( const job::RunContext& ) {
    if( !token->IsCancelled() && !token->HasFailed() )
    {
        auto entity = token->ExtractSpawnedEntity();
        entitySystem->ScheduleEntityAttach( { entity } );
    }
});
```

GDC

March 20-26, 2021
 San Francisco, CA

CD PROJEKT RED

8234299 -AXRC6.0001

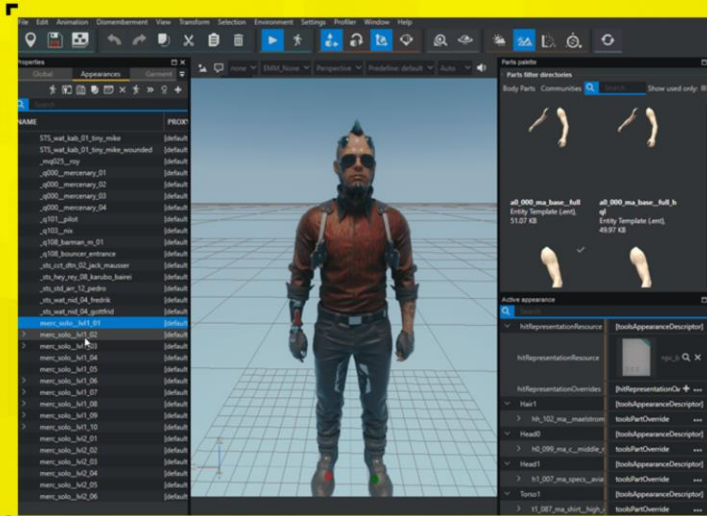
ECS— APPEARANCE

- Entity can have visuals decoupled from entity logic
- Appearance can be specified when scheduling spawning
- Over 8000+ different appearances for NPCs and vehicles



ECS — APPEARANCE EDITOR

58



GDC

March 20-26, 2022
San Francisco, CA

CD PROJEKT RED

8234299 - AXRC6 0001



SYSTEMS

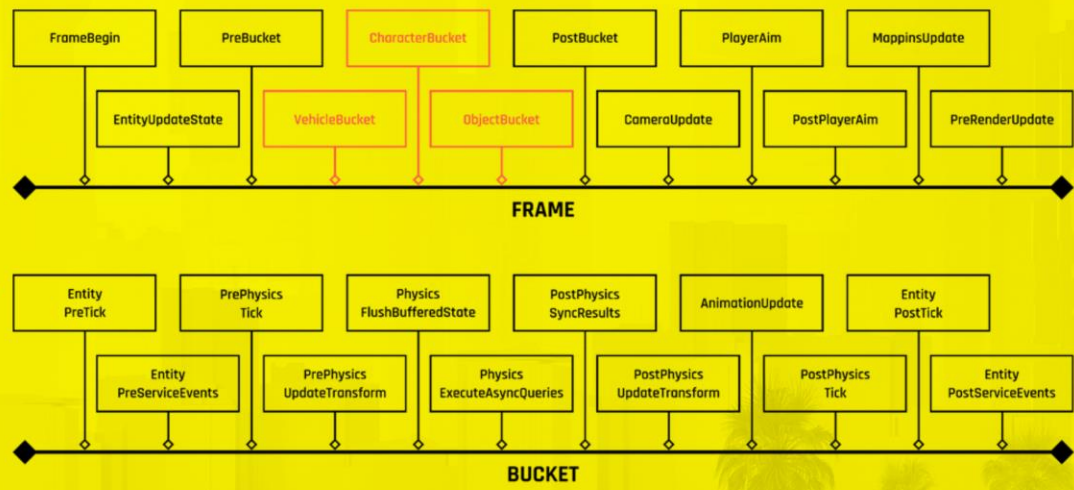
- World Systems – lifetime limited to world
 - Also available for editor preview
- Game System – available during the whole game process
- Main method to register to frame update
- Systems can communicate with each other
 - Public interface thread safety needs to be considered



March 20-24, 2022
San Francisco, CA



SYSTEMS – FRAME UPDATE & BUCKET GROUP



March 20-24, 2022
San Francisco, CA



SYSTEMS—REGISTRATION

- Any system can register itself to any update group and bucket
- By default, systems will be constrained to group/bucket
 - However it is possible to unshackle a system

```
void RuntimeSystemAudio::RegisterUpdateFunctions( UpdatableSystemRegistrar& registrar )
{
    registrar.Register( UpdateTickGroup::PreBuckets, *this, "Audio/Components",
    [this]( const UpdateContext& updateContext, job::Builder& builder ) {
        UpdateComponents( builder, updateContext.m_timeDelta, updateContext.m_tickInfo );
        if( m_isActive ) {
            m_dynamicReverbSystem->Update();
            UpdateEditorHedgehog( updateContext.m_timeDelta );
        }
    });

    registrar.RegisterInBucket( UpdateBucket::CharacterMask(), UpdateBucketPhase::PrePhysicsTick, *this, "Audio/BatchRaycastResolverCall",
    [this]( UpdateBucket::Enum bucket, const UpdateContext& updateContext, job::Builder& builder ) {
        if( m_isActive ) {
            m_batchRaycastResolver->ResolvePreRaycast();
        }
    });
}
```

PHYSICS

63

- Physics systems are built on top of PhysX
- PhysX tasks were adapted to be compatible with our job system
- Simple "C style" public API
- Safe to read and write states
- State writes are buffered and applied to all modified proxies at specific points in frame
- Concurrent writes could have been supported. However it wasn't needed

```
REDPHYSICS_API ProxyID CreateProxy( ObjectDesc& desc );
REDPHYSICS_API void DestroyProxy( ProxyID id );

REDPHYSICS_API Vector3 GetPosition( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Quaternion GetRotation( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Transform GetTransform( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Vector3 GetLinearVelocity( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Vector3 GetAngularVelocity( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Float GetLinearSpeed( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Vector3 GetDisplacement( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API ImpulseData GetImpulseAccumulator( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Float GetAngularDamping( ProxyID proxyId, ActorIndex subPart = 0 );
REDPHYSICS_API Float GetLinearDamping( ProxyID proxyId, ActorIndex subPart = 0 );
```

GDC

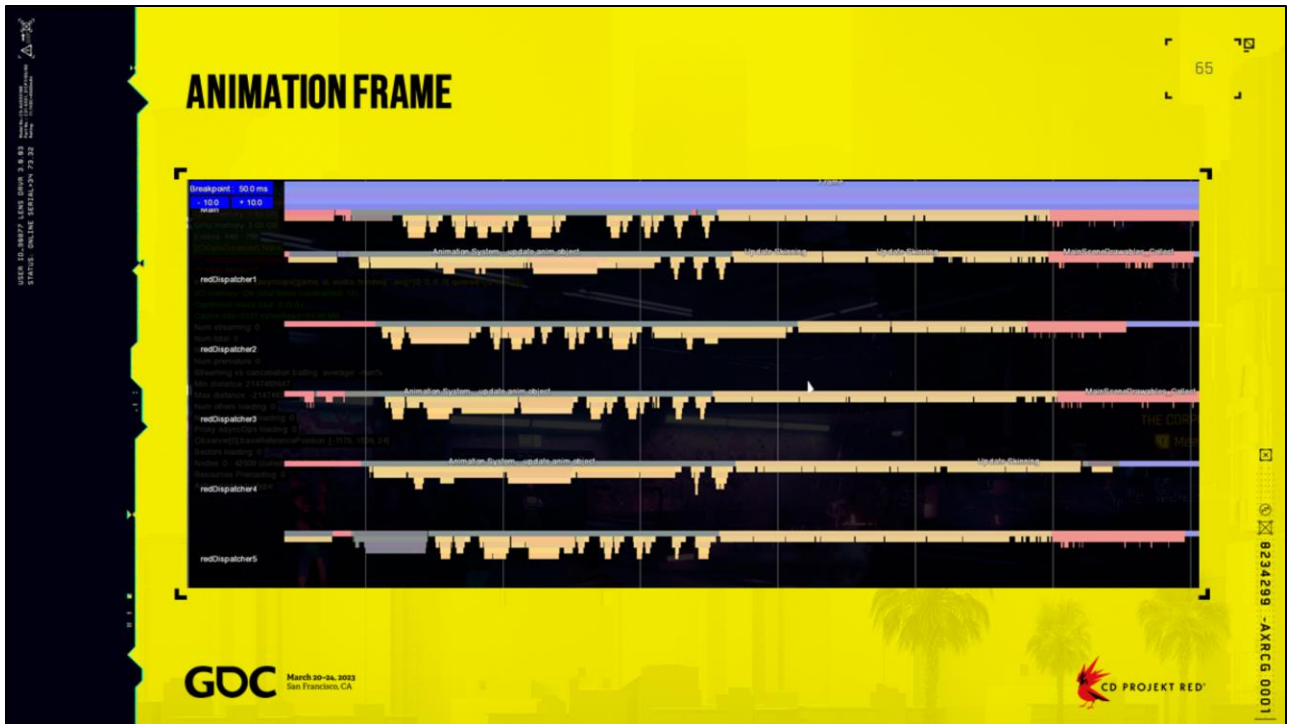
March 20-26, 2022
 San Francisco, CA



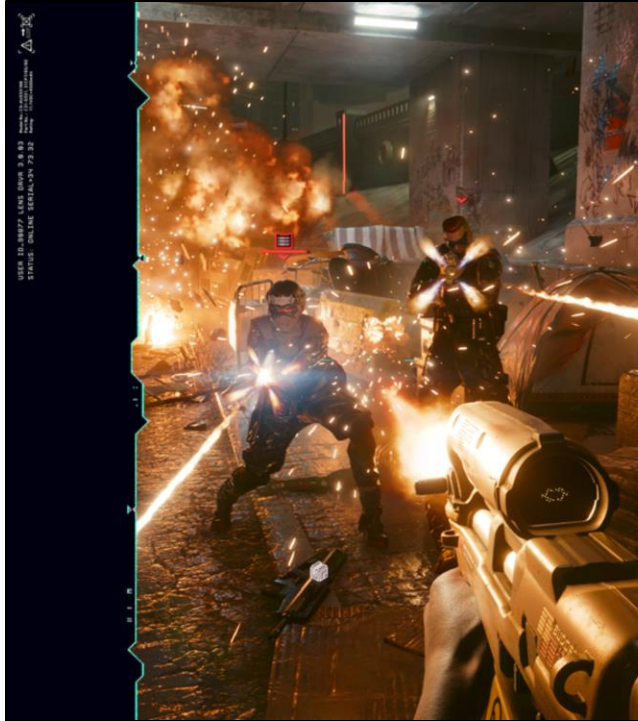
ANIMATION IN A NUTSHELL

- **Parallel update of characters**
 - As soon as character pose is calculated, schedule skinning job and send result to rendering
- **Animation instancing for massive standing crowd**
- **No update if occluded**
 - Exception if closer than 5m from player
- **Sleep mode for doors & vehicles if no movement**
- **Temporary allocation solved using scratchpad buffer**
- **Animation Streaming of 40mb budget**
 - 3k-4k animations in game





PS4 capture, a bit less than 3ms, Lizzies bar 20 npc in view, 40npc in surrounding



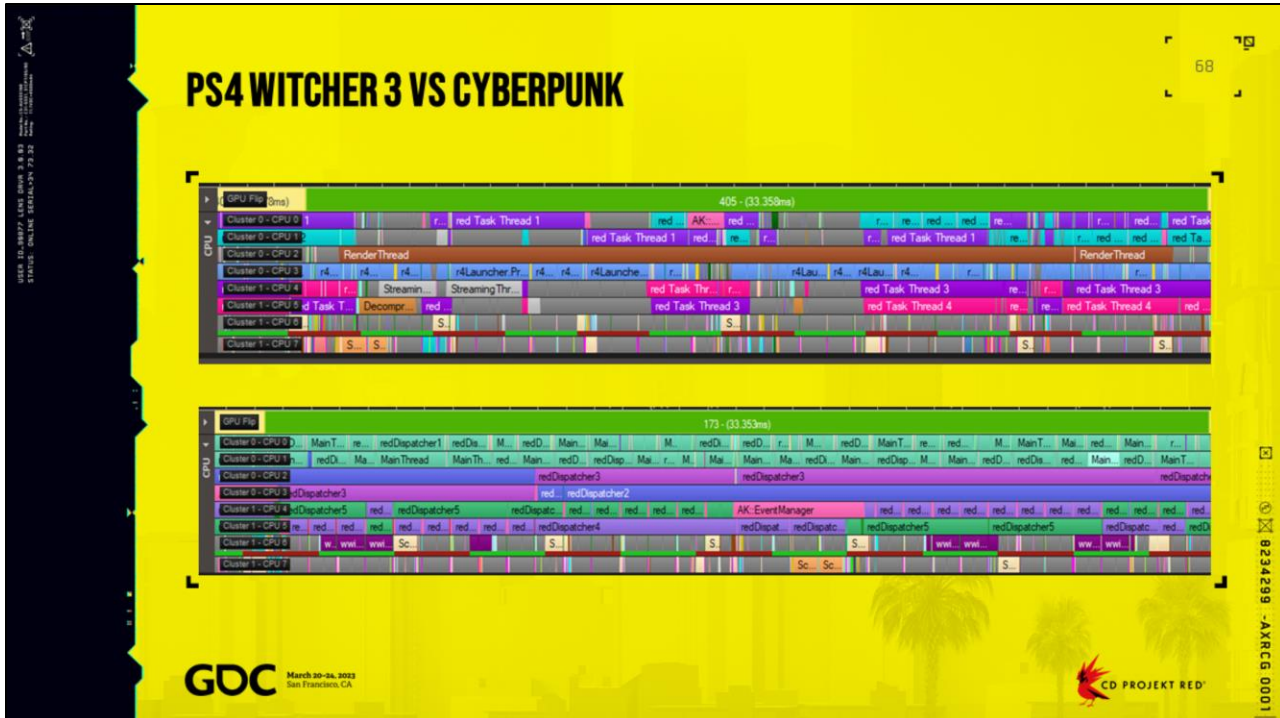
AI — LIVING ON THE EDGE

- Low hanging fruit & optimization of algorithms used
- Reducing behavior trees update frequency
- Parallelizing processing
- Fixing cache misses
- Logic LOD

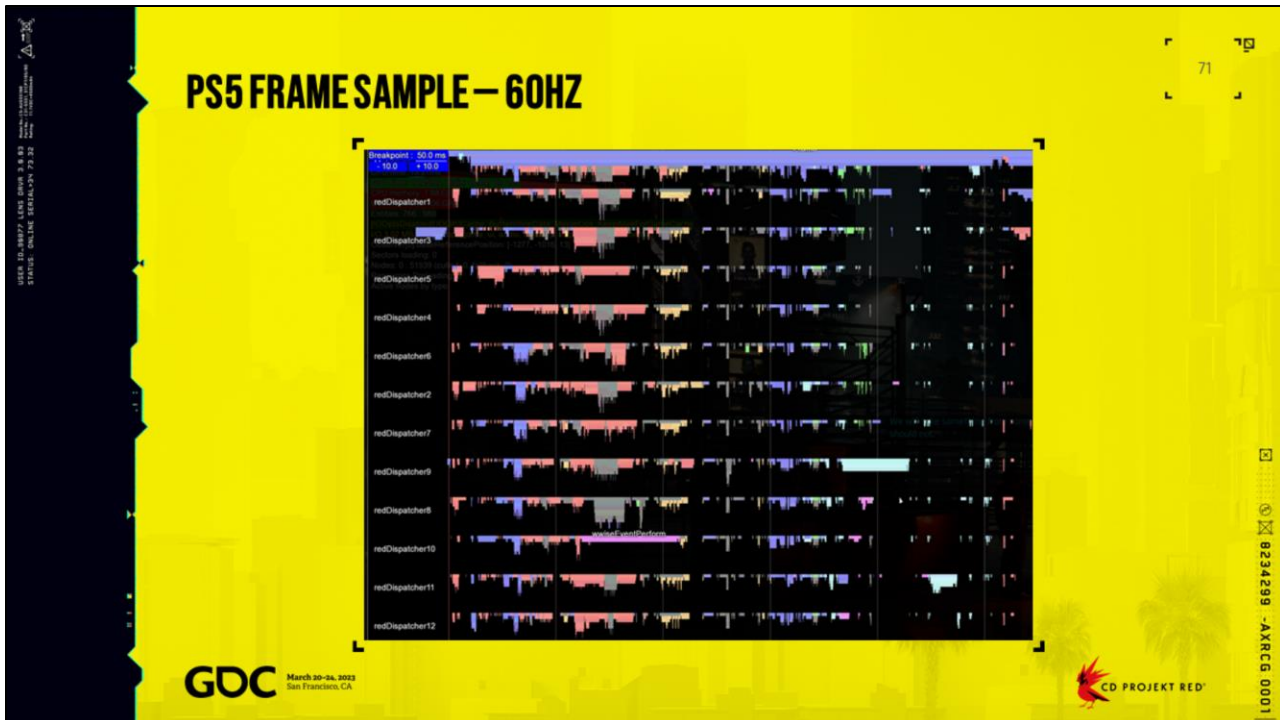




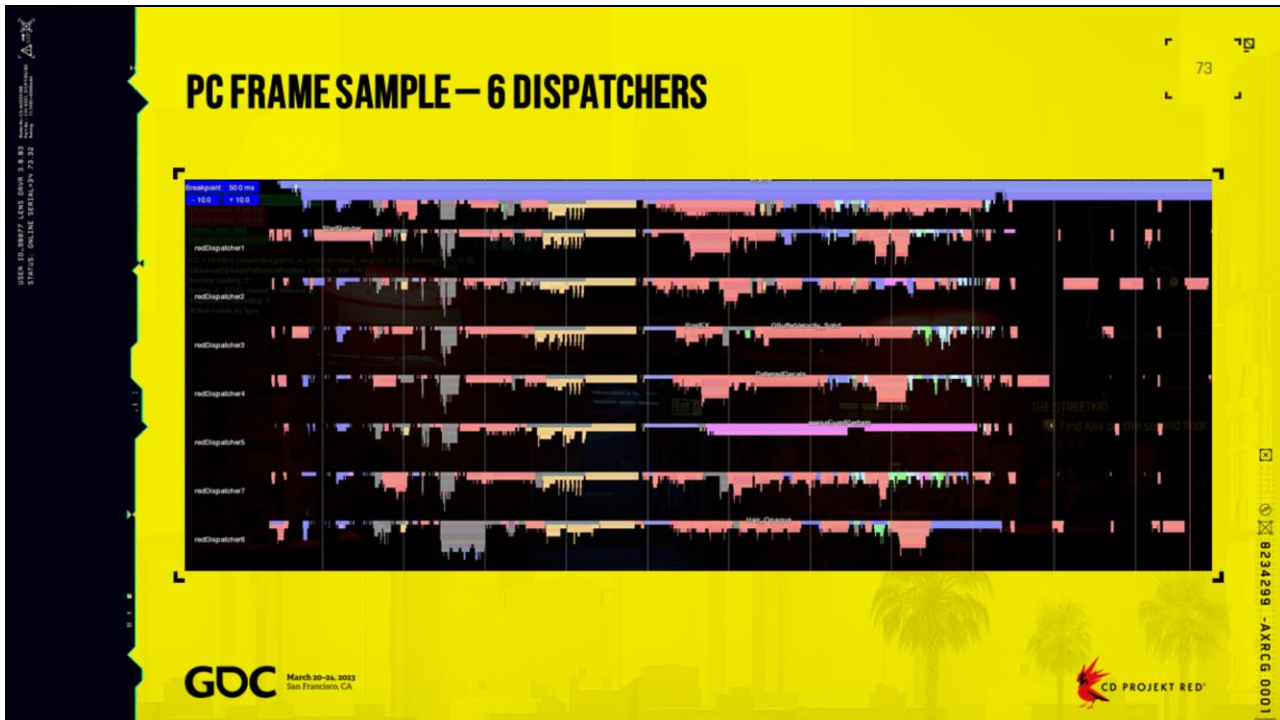
You can change the img on the right



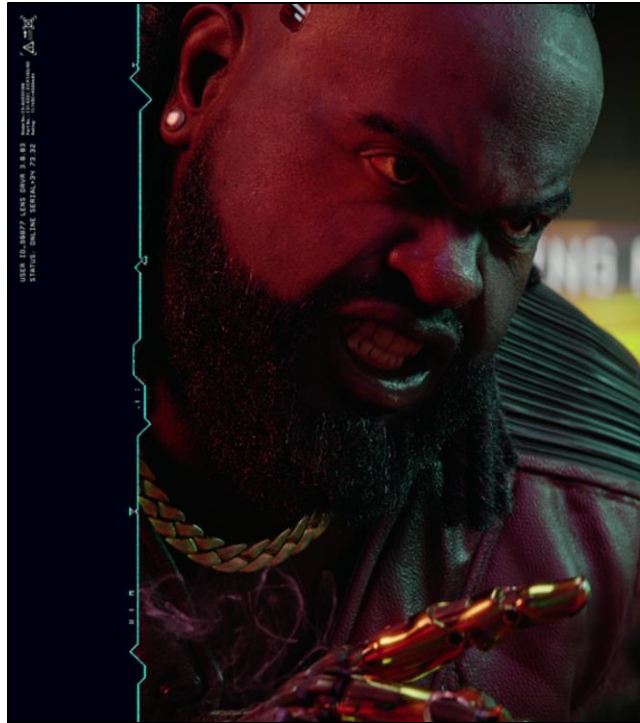
PS4, Witcher3 on top, Cyberpunk bottom one



More or less 12ms on CPU



Force 6 dispatcher,
i9 - 7980XE @2.6ghz

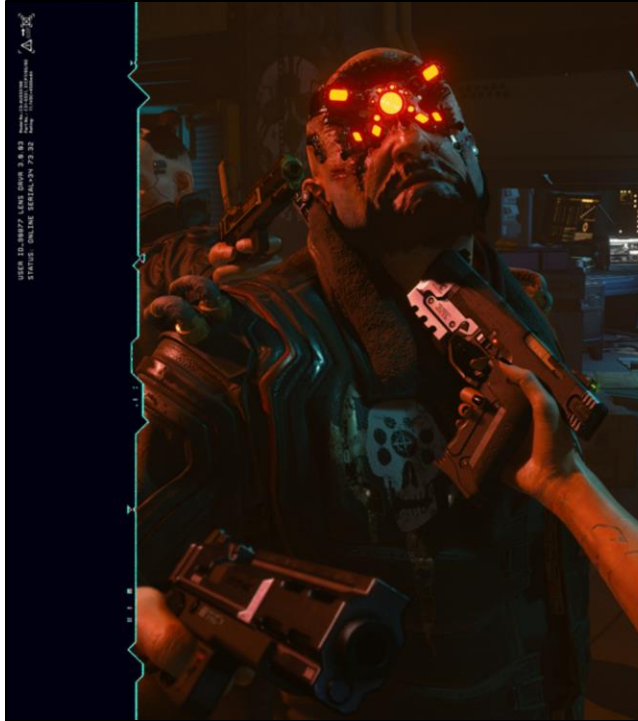


WHAT ABOUT THE MAIN THREAD?

- We did not manage to completely eliminate it
- It only schedules beginning of frame and waits for completion
- Acts as a regular job worker while waiting



You can change the img on the right



WHAT WENT WRONG?

- Some critical engine changes came in late
- Keeping scalability in mind is hard
- Developing multithreaded code is very hard
- Flexibility has a cost
- Some areas of the game were badly made
- Critical tools were remade



WHAT WENT RIGHT ?

- The technological leap from The Witcher 3 to Cyberpunk was crazy
- PS5 & XSX/XSS port went smoothly
- No more main & render thread limitation
- Multithreaded gameplay & script
- Great scalability



87204

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098

GDC March 20-24, 2023
San Francisco, CA



8234299 -AXBFG 0001



GRAPHICS — RENDER GRAPH — SETUP & CULLING

```
auto NStartRender = RENDER_UNIQUE_SIMPLE_COMMAND_LIST( G_None, "StartRender", UNIQUE_RENDERNODE_StartRender, CRenderNode_StartRender );
auto NPrepareCollector = ADD_NODE( G_None, "PrepCollector", CRenderNode_PrepareCollector );
factory.Link( NStartRender, NPrepareCollector );

auto NCullingScene = ADD_NODE( G_Culling, "DoCullingMainScene", CRenderNode_DoCulling, CullingMode::MainScene );
auto NCullingRayTracing = ADD_NODE( G_Culling, "DoCullingRayTracing", CRenderNode_DoCulling, CullingMode::RayTracedObjects );
auto NCullingCascades = ADD_NODE( G_Culling, "DoCullingCascades", CRenderNode_DoCulling, CullingMode::Cascades );
auto NCullingLocalShadows = ADD_NODE( G_Culling, "DoCullingLocalShadows", CRenderNode_DoCulling, CullingMode::DynamicShadows );

factory.Link( NPrepareCollector, G_Culling );
factory.Link( NCullingScene, NCullingLocalShadows );

// Visibility system might generate some debug geometry, so after all the culling is finished we flush it all to vertex/index data.
auto NVisDebug = ADD_NODE( G_None, "FlushVisiDebug", CRenderNode_FlushVisibilityDebug );
factory.Link( G_Culling, NVisDebug );

// After we've done culling that might include particles, we can kick off simulation for any particle systems that are visible.
for( UInt32 i = 0; i < NUM_PARTICLE_THREADS; ++i )
{
    ADD_NODE( G_ParticlesOnScreenSim, "SimulateOnScreenCPUParticles", CRenderNode_SimulateOnScreenCPUParticles,
        vis::CollectThreadingSetup( i, NUM_PARTICLE_THREADS ) );
}
factory.Link( NCullingScene, G_ParticlesOnScreenSim );
factory.Link( NCullingCascades, G_ParticlesOnScreenSim );
```



March 20-24, 2022
San Francisco, CA



Set up for render, and kick off culling

"Unique" nodes are for special points in the frame. Not too important in this example, but used when merging multiple graphs -- unique stay unique (all unique from all subgraphs are de-duplicated), regular nodes are copied over.

"Simple command list" just means it creates a command list that only runs a single subnode.

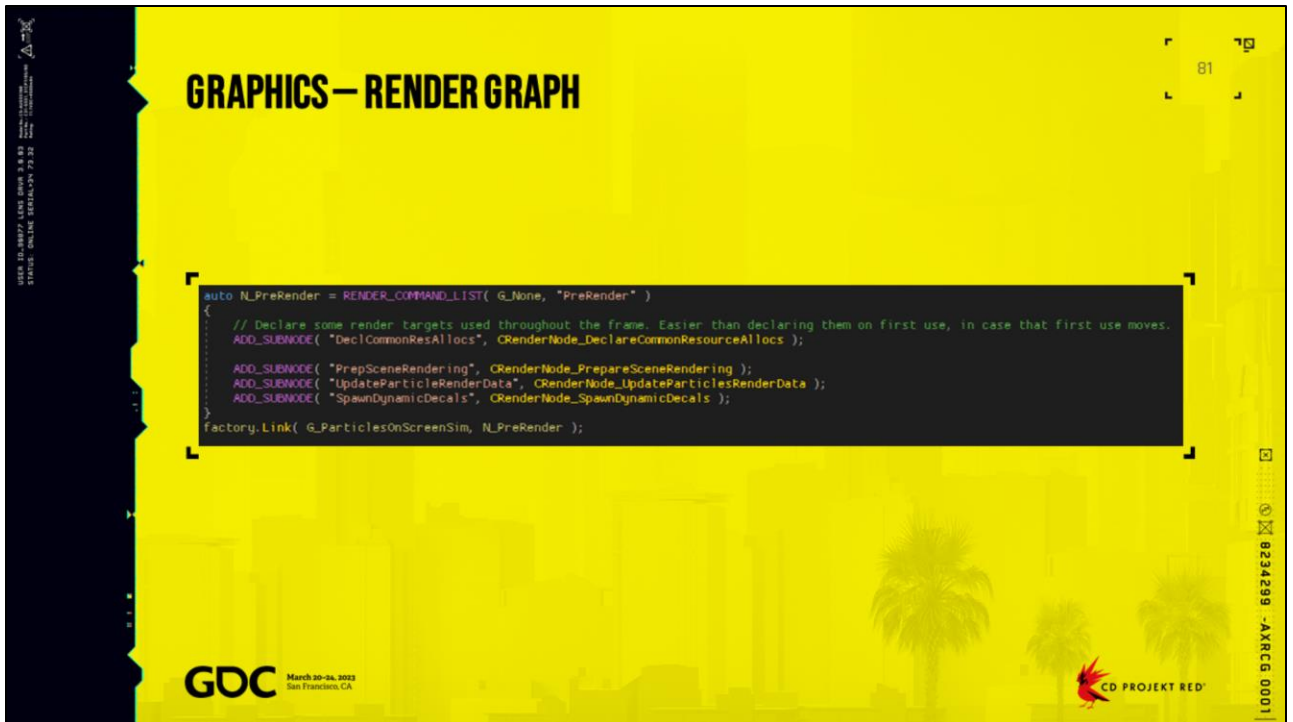
"Add node" is for pure CPU work, no command list created.

Add CPU dependency, so PrepareCollector won't run until StartRender is finished.

Add nodes for doing different types of culling. Main scene (player camera), RT (inflated frustum, and area around camera), Cascades, and Local shadows (spot lights).

All culling needs to wait for PrepareCollector to finish.

In addition, LocalShadows needs to wait for main scene culling, in order to know what lights are visible.



Initial setup is ready, so we start doing some rendering work.

Command lists are explicitly defined. All subnodes in a command list node with run sequentially, although not necessarily on the same thread. Subnodes are able to branch off into additional parallel work if needed (but only for CPU work, the command list is only accessible from a single thread at a time).

In this case, UpdateParticleRenderData needs the results of the particle sim, since it's sending the final particle data to the GPU.

GRAPHICS—RENDER GRAPH—GBUFFER

```
auto N_GBufferPrepare = ADD_NODE( G_Render, "GBuffer_Prepare", CRenderNode_PrepareRenderElements, "renderstage_gbuffer_regular",  
    SPL_OptimizedDistanceBatching, UsePreparedChunks( c_gbufferSplit ) );  
factory.Link( N_CullingScene, N_GBufferPrepare );  
factory.Link( G_ParticlesOnScreenSim, N_GBufferPrepare );  
  
for( UInt32 i = 0; i < c_gbufferSplit; ++i )  
{  
    auto N_GBuffer = RENDER_COMMAND_LIST( G_Render, "GBuffer" )  
    {  
        ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );  
        ADD_SUBNODE( "SetRenderToGbuffer", CRenderNode_SetRenderTargetsGBuffer, i == 0 ? rt_GBuffer_Clear : rt_GBuffer_NoClear );  
        ADD_SUBNODE( "RenderElements", CRenderNode_RenderElements, "renderstage_gbuffer_regular", SPL_OptimizedDistanceBatching,  
            UsePreparedChunks( c_gbufferSplit, i ) );  
        ADD_SUBNODE( "EndRenderToGbuffer", CRenderNode_EndRenderTargetsGBuffer, i == 0 ? rt_GBuffer_Clear : rt_GBuffer_NoClear );  
        ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );  
    }  
    factory.Link( N_GBufferPrepare, N_GBuffer );  
}  
  
SYNC_SUBMIT( G_None, "RenderGraphCamera_GBuffer", GpuApi::CommandListSyncType::None );
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

GBuffer

Since we tend to have a lot of work to do in rendering the static meshes in the scene, gbuffer is split across multiple command lists.

First a CPU-only node to build and sort the list of objects to be drawn. This needs the results of main scene culling, as well as on-screen particles.

Then several command lists that each take a portion of the collected objects, drawing them to the gbuffer. These all depend on GBufferPrepare.

"BindGlobalConstants" / "UnbindGlobalConstants" are reused in many places, they set up some global constant buffers, resource bindings, etc.. Using subnodes allows that to be reused easily. Similar with setting some common render target setups.

Since we have a pretty hefty amount of work built up with the GBuffer, we might want to submit it to the GPU already, so that it can keep busy with that while we prepare more.

`SYNC_SUBMIT` will add a node that automatically has a dependency on any GPU-related nodes before it, and will submit all of them to the GPU. Here we don't need to do any additional synchronization on the GPU, so we pass `None` for `sync` type.


83


GRAPHICS — RENDER GRAPH — GBUFFER + VELOCITY BUFFER

```
auto NLGBufferVelocity = RENDER_COMMAND_LIST( G_Render, "GBufferVelocity" )
{
    ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );
    ADD_SUBNODE( "InitVelocityBuffer", CRenderNode_InitVelocityBuffer );
    ADD_SUBNODE( "BeginRenderToGBufferWithVel", CRenderNode_SetRenderTargetsGBufferWithVelocityBuffer, true );

    ADD_SUBNODE( "RenderElements", CRenderNode_RenderElements, "renderstage_gbuffer_velbuff_regular", SPL_OptimizedDistanceBatching );

    ADD_SUBNODE( "EndRenderToGBufferWithVel", CRenderNode_SetRenderTargetsGBufferWithVelocityBuffer, false );
    ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
}
factory.Link( NL_CullingScene, NLGBufferVelocity );
factory.Link( G_ParticlesOnScreenSim, NLGBufferVelocity );
```

 March 20-24, 2022
San Francisco, CA

 CD PROJEKT RED

GBuffer + Velocity Buffer

Dynamic objects generate normal GBuffer outputs, plus they write motion vectors to a velocity buffer.

As with GBuffer, we need to wait for scene culling and particles, but it can run on the CPU in parallel with the static GBuffer.

GRAPHICS — RENDER GRAPH — ASYNC COMPUTE

```
SYNC_SUBMIT( G_None, "RenderGraphCamera_FinishGBuffer", GpuApi::CommandListSyncType::ForkAsyncCompute );

auto N_LASUpdateStatic = COMPUTE_COMMAND_LIST( G_Render, "AccelerationStructureUpdateStatic" )
{
    ADD_SUBNODE( "AccelerationStructureUpdateStatic", CRenderNode_AccelerationStructureUpdateStatic );
}
auto N_LASUpdateDynamic = COMPUTE_COMMAND_LIST( G_Render, "AccelerationStructuresUpdateDynamic" )
{
    ADD_SUBNODE( "AccelerationStructureUpdateDynamic", CRenderNode_AccelerationStructureUpdateDynamic );
}
auto N_LASUpdateEpilogue = COMPUTE_COMMAND_LIST( G_Render, "AccelerationStructuresUpdateEpilogue" )
{
    ADD_SUBNODE( "AccelerationStructureUpdateEpilogue", CRenderNode_AccelerationStructureUpdateEpilogue );
}

factory.Link( N_CullingRayTracing, N_LASUpdateStatic );
factory.Link( N_CullingRayTracing, N_LASUpdateDynamic );

factory.Link( N_LASUpdateStatic, N_LASUpdateEpilogue );
factory.Link( N_LASUpdateDynamic, N_LASUpdateEpilogue );

COMPUTE_COMMAND_LIST( G_Render, "AsyncComputeDuringShadowmaps" )
{
    ADD_SUBNODE( "BuildDepthChain", CRenderNode_BuildDepthChain );
    ADD_SUBNODE( "RenderSSAO", CRenderNode_DrawConeAO );
    ADD_SUBNODE( "GenerateTonemappingLUT", CRenderNode_GenerateTonemappingLUT );
    ADD_SUBNODE( "GenerateAsyncDynamicTextures", CRenderNode_GenerateAsyncDynamicTextures );
}
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

Async Compute work during shadowmap rendering

With the GBuffer finished, we can run some passes like a Hi-Z generation, SSAO, as well as some independent work. We run those things on async compute, while on the graphics queue we fill in shadow maps.

Our async is limited to a fork-join model, where we branch off to run specific graphics and compute work in parallel, and then sync the queues at the end. So we need to submit everything we have so far, and indicate that we want to Fork.

If we're doing ray tracing, we can build acceleration structures. Static and dynamic bottom levels have different work involved, so we split them into separate command lists. In addition those subnodes can branch off into further parallel work to prepare the acceleration structures for building. After the bottom levels are built, we have an additional step to build the top level and shader table.

We also have another compute command lists for other compute workloads, there aren't any CPU dependencies

so we don't need to link it to anything.

GRAPHICS – RENDER GRAPH – SHADOWMAPS

```
for( UInt32 cascadeIndex = 0; cascadeIndex < Config::GlobalRenderingSettings.NumShadowCascades; cascadeIndex++ )
{
    auto cascadeID = RENDER_COMMAND_LIST( G_Render, "RenderCascade" )
    {
        ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );
        ADD_SUBNODE( "RenderCascade", CRenderNode_RenderShadowCascade, cascadeIndex );
        ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
    }

    factory.Link( N_CullingCascades, cascadeID );
    factory.Link( G_ParticlesOnScreenSim, cascadeID );
}

for( UInt32 lightIndex = 0; lightIndex < Config::GlobalRenderingSettings.LocalShadowsProcessedPerFrame; lightIndex++ )
{
    auto localShadowsID = RENDER_COMMAND_LIST( G_Render, "RenderLocalShadows" )
    {
        ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );
        ADD_SUBNODE( "RenderLocalShadows", CRenderNode_RenderLocalShadowMaps, lightIndex );
        ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
    }

    factory.Link( N_CullingLocalShadows, localShadowsID );
    factory.Link( G_ParticlesOnScreenSim, localShadowsID );
}

SYNC_SUBMIT( G_None, "RenderGraphCamera_Shadows", GpuApi::CommandListSyncType::JoinAsyncCompute );
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

Shadowmaps

With the async compute work defined, we can do the shadowmap rendering. There's a command list for each cascade, and for each local light we plan to update. We need to wait for the appropriate culling to finish, as well as particle sim, in case there were some shadow-casting mesh particles.

And now with the compute and graphics work set up, we can submit that to the GPU, indicating that we're doing a Join sync.

GRAPHICS — RENDER GRAPH — LIGHTING & POSTFX

```
RENDER_COMMAND_LIST( G_Render, "Lighting" )
{
    ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );
    ADD_SUBNODE( "BindLightingGlobalConstants", CRenderNode_BindLightingGlobalConstants );

    ADD_SUBNODE( "SkyScattering", CRenderNode_RenderSkyScattering );
    ADD_SUBNODE( "VolumetricFog", CRenderNode_VolumetricFog );

    ADD_SUBNODE( "RTShadows", CRenderNode_RenderRayTracedGlobalShadow );
    ADD_SUBNODE( "RTAmbientOcclusion", CRenderNode_RenderRayTracedAmbientOcclusion );
    ADD_SUBNODE( "RTReflections", CRenderNode_RenderRayTracedReflections );
    ADD_SUBNODE( "RTFilterOutput", CRenderNode_RayTracingFilterOutput );

    ADD_SUBNODE( "LightBuffers", CRenderNode_RenderLightBuffers );
    ADD_SUBNODE( "CalculateGI", CRenderNode_CalculateGI );

    ADD_SUBNODE( "LightIntegrate", CRenderNode_RenderLightsIntegrate );

    ADD_SUBNODE( "UnbindLightingGlobalConstants", CRenderNode_UnbindLightingGlobalConstants );
    ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
}

RENDER_COMMAND_LIST( G_Render, "PostFX" )
{
    ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );

    ADD_SUBNODE( "ApplyTAA", CRenderNode_ApplyTAA );
    ADD_SUBNODE( "ApplyDof", CRenderNode_DepthOfFieldSeparable );
    ADD_SUBNODE( "ApplyBloomAndTonemapping", CRenderNode_ApplyBloomAndTonemapping );

    ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
}
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

Lighting, PostFX

After we have the shadowmaps and async compute joined, we can calculate our lighting, post processes, etc.

If there are no CPU dependencies, we don't need to link anything and these command lists can be built at any time.

USER: 10.1007/978-1-4939-9834-2
STAGE: ONLINE SERIAL-ON-72 32

87


GRAPHICS – RENDER GRAPH – DEBUG

```
auto debugID = RENDER_COMMAND_LIST( G_Render, "Debug" )
{
    ADD_SUBNODE( "BindGlobalConstants", CRenderNode_BindGlobalConstants );
    ADD_SUBNODE( "BindLightingGlobalConstants", CRenderNode_BindLightingGlobalConstants );

    ADD_SUBNODE( "ApplyDebugPreview", CRenderNode_ApplyDebugPreview );
    ADD_SUBNODE( "RenderDebugWorld", CRenderNode_RenderDebugFragments, DebugFragments::WorldAndWorldOverlay );

    ADD_SUBNODE( "UnbindLightingGlobalConstants", CRenderNode_UnbindLightingGlobalConstants );
    ADD_SUBNODE( "UnbindGlobalConstants", CRenderNode_UnbindGlobalConstants );
}
factory.Link( N_VisDebug, debugID );
```

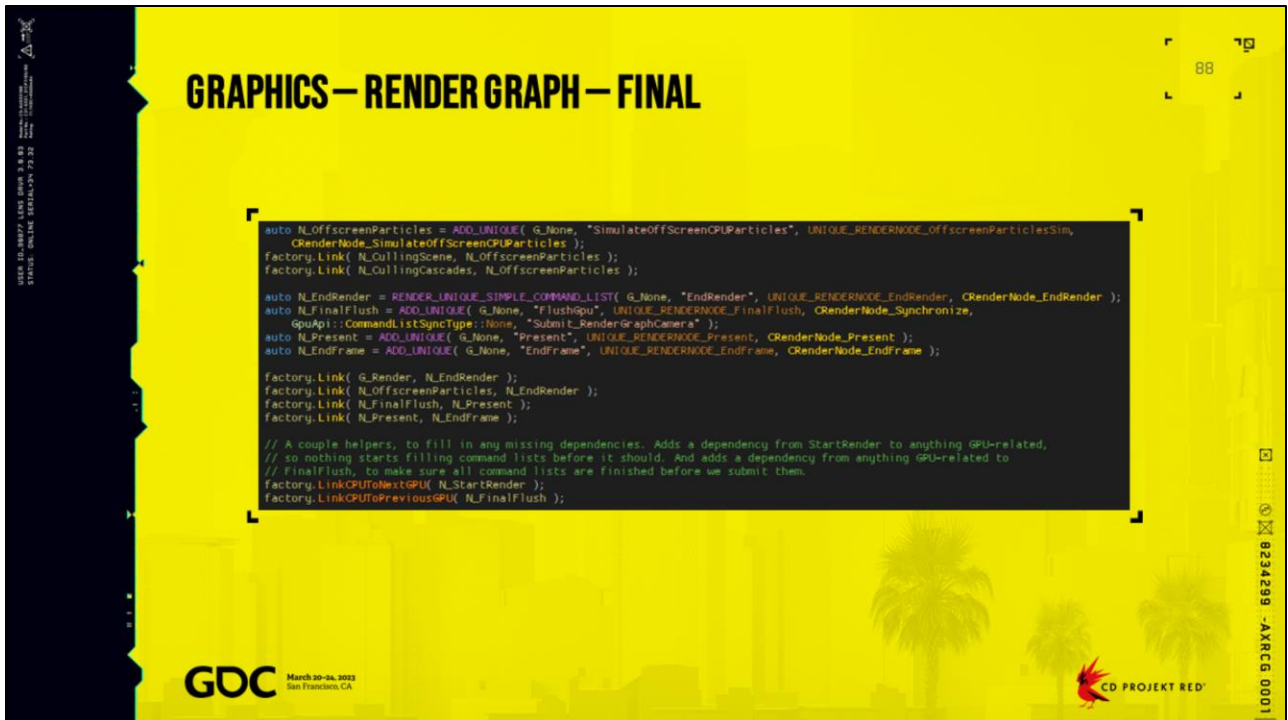
 March 20-24, 2022
San Francisco, CA



8234299 - AXRC6 0001

We might have some debug geometry to draw after everything else is done.

For this, we have a dependency on VisDebug from earlier, in case it needed to add anything.



Main rendering is done, so we just have some assorted finalization.

We have a separate simulation pass for offscreen particles, to keep them updating but at a throttled pace.

This is another unique node, which is only important for cases where we have multiple renders in a frame (such as when a mirror is up). We only want a single offscreen simulation, and it will wait for culling from all subgraphs so it knows what's actually offscreen everywhere.

There's a final command list with some last-minute book-keeping in it, then submit everything to the GPU that hasn't been submitted yet, and present to the screen. Everything at that point is sequential on CPU, and needs to wait for all the rendering work to complete.

GRAPHICS— RENDER NODE

- Render graph runs in two phases – prepare and consume
- Prepare phase records resource lifetime events
- The lifetime events in prepare / consume phases must match exactly (same order, same resource descs, etc.)
- Node declares that it requires a Command List
 - This allows to issue GPU commands on consume phase
- Job Builder can be safely used to execute parallel work, or continuation work



March 20-24, 2022
San Francisco, CA



GRAPHICS—RENDER NODE

```
struct RenderNode_SomeEffect : public RenderNodeBase
{
    virtual RenderNodeCommandListUsage GetCommandListUsage() const override { return RenderNodeCommandListUsage::Require; }
    virtual void Execute( const RenderNodeImplContext& rctx, job::Builder* builder ) const final override
    {
        SRenderFlowTargetDesc colorDesc = rctx.GetRegularPrecisionColorTargetDesc();
        const RenderFlowNameTag tempTex = rctx.RTTempAlloc( RENDER_NAME_TAG( "TempTarget" ), colorDesc );
        // First pass, copy main scene to an intermediate texture.
        auto tempRT = rctx.RTC_RFLUF_Write >( tempTex );
        auto colorRT = rctx.RTC_RFLUF_Read >( RENDER_NAME_TAG( "color" ) );

        if( rctx.IsConsumePhase() ) {
            struct CB_PARAMS_1 { Vector4 colorTint; };
            const CB_PARAMS_1 cbData; // Fill in parameters as needed
            rctx.BindTextureUAV< 0 >( tempRT );
            rctx.BindTexture< 0, eCOMPUTE >( colorRT );
            rctx.SetConstant< eCOMPUTE >( cbData );
            GetRenderer()->GetShader( SHADER_NAME( "m_postfxSomeEffect_pass1" ) )->Dispatch( ... );
        }

        // Second pass, copy it back to the main scene target.
        auto tempRT = rctx.RTC_RFLUF_Read >( tempTex );
        auto colorRT = rctx.RTC_RFLUF_Write >( RENDER_NAME_TAG( "color" ) );
        if( rctx.IsConsumePhase() ) {
            struct CB_PARAMS_2 { Matrix uvTransform; };
            const CB_PARAMS_2 cbData; // Fill in parameters as needed
            rctx.BindTextureUAV< 0 >( colorRT );
            rctx.BindTexture< 0, eCOMPUTE >( tempRT );
            rctx.SetConstant< eCOMPUTE >( cbData );
            GetRenderer()->GetShader( SHADER_NAME( "m_postfxSomeEffect_pass2" ) )->Dispatch( ... );
        }
    }
};
```



March 20-24, 2022
San Francisco, CA



8234299 - AXRC6 0001

First example, some sort of post process

This is using a temporary texture to hold some intermediate results.

The graph is run in two phases, Prepare and Consume. During Prepare phase, resource lifetime events are recorded, so that before running the Consume phase we can analyze total lifetime, resource sizes, etc. and alias multiple resources over the same chunks of GPU memory.

The lifetime events in Prepare and Consume must match exactly (same order, same resource desc), so occur outside of any checks for the current phase.

RTTempAlloc doesn't necessarily cause the resource to be allocated, it will be available only after the first use (in this case, marked by `rctx.RT<>`). We specify how the resource is intended to be used (read or write), but this metadata ended up not really being used.

The node declares that it requires a Command List, which allows it to issue GPU commands during the Consume phase.

GRAPHICS—RENDER NODE

```
struct RenderNode_DoCulling : public RenderNodeBase
{
    virtual RenderNodeCommandListUsage GetCommandListUsage() const override { return RenderNodeCommandListUsage::None; }
    virtual bool GetJobBuilderUsage() const override { return true; }
    virtual void Execute( const RenderNodeImplContext& rctx, job::Builder* builder ) const override
    {
        if( rctx.IsConsumePhase() ) {
            if( m_cullingMode == CullingMode::MainScene ) {
                // issue a job as part of this node. The node will not be finished until all additional work is done.
                builder->DispatchJob( "DoCull_SceneLayer", [rctx]( const job::RunContext& context ) {
                    // Do culling, which may spawn additional jobs as children of this one.
                } );
            }
            // etc.
        }
        CullingMode m_cullingMode;
    }
};
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

Next example is a CPU-only node. Because it declares that it won't use a Command List, it would be invalid to try to issue any GPU commands during its execution.

In this case, we also declare that we would like to use a `job::Builder`, which allows for spawning additional parallel work. Any dependencies on this node will need to wait until any additional jobs are also finished.

Non-GPU node, which spawns additional jobs as part of the work.

GRAPHICS—RENDER NODE

92

```
struct RenderNode_AccelerationStructureUpdateEpilogue : public RenderNodeBase
{
    virtual RenderNodeCommandListUsage GetCommandListUsage() const override { return RenderNodeCommandListUsage::Require; }
    virtual Bool GetJobBuilderUsage() const final override { return true; }
    virtual void Execute( const RenderNodeImplContext& rctx, job::Builder* builder ) const final override
    {
        if( rctx.IsConsumePhase() )
        {
            // Issue some GPU commands first

            // Unbind the command list from the current thread. Another thread could run on this thread!
            GpuApi::BindCommandList( {} );

            // Spawn several jobs to do some processing in parallel over all instances.
            CollectTLAS_Parallel( rctx, *builder );

            // After the above jobs are finished, run another job which will finish up with some more GPU work.
            builder->DispatchJob( "Finalize", [ this, rctx ]( const job::RunContext& context ) {
                // Bind this node's command list to whatever thread we're running on.
                GpuApi::BindCommandList( rctx.GetCommandList() );
                UpdateDataGPU( rctx );

                // Unbind it, because the remaining nodes in this command list can end up on a different thread.
                GpuApi::BindCommandList( {} );
            } );
        }
    }
};
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

Finally we have a GPU node, which also can spawn additional parallel work.

We only have a single Command List, so we need to be careful when accessing it so multiple threads don't try to access it. But there's no restriction on which thread it can be used.

A Command List must be bound to a thread, and can only be bound to a single thread at a time. So in order to access it from a separate job, we need to unbind it from the current thread first.

PHYSICS— STATE BUFFERING

93

- **Predefined states:**
 - Position, Rotation, Linear Velocity, Angular Velocity... and more!
- **State Block**
 - All information to set a state in a proxy
 - Up to 4 states in a single block
 - If more are needed, linked list of StateBlocks
- **State Allocator**
 - Can allocate a block (max 64k)
 - Can allocate data for states (preallocated 1mb)
 - In both cases, a linear allocator flushed every frame
- **State Flush**
 - At given point(s) in frame, all valid and buffered states are applied to proxies

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

PHYSICS—STATE BLOCK

All write operations are wrapped in ProxyWriter interface

1. Takes ProxyID as argument
2. Allocates new block or get current one if already exists
3. If there is an existing block, create full copy of all states so far
4. Writes all needed states to block, or allocate new one if there's no space
5. Commit states. If pointer from the beginning didn't change, swap pointers. Otherwise go back to 3 *

```
struct StateBlock
{
    enum { MAX_STATES = 4 };
    ProxyID proxyId;
    uint16_t numStates;

    union {
        uint8_t all;
        struct {
            uint8_t headOfList : 1;
            uint8_t flushed : 1;
            uint8_t outdated : 1;
        };
    };
    StateValueFlag_ stateValueFlags;
    StateBlock* next;
    uint32_t frameNo;

    struct {
        struct Offset {
            uint32_t offset : 24;
            uint32_t size : 8;
        };

        Offset offset [MAX_STATES];
        uint8_t state [MAX_STATES];
        uint8_t subpart [MAX_STATES];
        uint8_t nbShapes [MAX_STATES];
    } states;
};
```



March 20-24, 2022
San Francisco, CA



Copy of all states can be done on multiple threads

* In practice there was no concurrent writers. We just swap pointers on commit. But other threads can safely read states at same time.

PHYSICS— STATE ALLOCATOR & FLUSH

95

- **StateAllocator**
 - Acts like a single linear allocator. Allocated from fixed, preallocated memory area
 - All content is discarded after flush
- **State Flush**
 - For each proxy, check if there is a StateBlock allocated. Should be marked as "headOfList"
 - Traverse through all states and connected blocks and apply data to proxy*

```
void FlushBufferedProxyStates( StateAllocator* stateAllocator ) {  
    StateBlock* blockIt = stateAllocator->BlockBegin();  
    StateBlock* blockEnd = stateAllocator->BlockEnd();  
    while (blockIt != blockEnd) {  
        if( StateBlock* proxyCurrentBlock = g_proxyLookup->m_stateBlocks[blockIt->proxyId.Index()].GetValue() ) {  
            proxy_internal::FlushProxyState( stateAllocator, proxyCurrentBlock->proxyId, proxyCurrentBlock );  
            proxy_internal::CommitStateBlock( proxyCurrentBlock->proxyId, nullptr );  
        }  
        ++blockIt;  
    }  
    stateAllocator->Clear();  
}
```

GDC

March 20-24, 2022
San Francisco, CA

CD PROJEKT RED

* This operation is single threaded. All writes to physics scene are not thread-safe.