
BUILDING NIGHT CITY:
THE TECHNOLOGY OF
CYBERPUNK 2077
GDC March 20–24, 2023
San Francisco, CA
Charles Tremblay | Engineering Director
CD PROJEKT RED
8234299 -AXRCG 0001

BRIEF HISTORY OF RED ENGINE

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

CYBERPUNK REQUIREMENTS

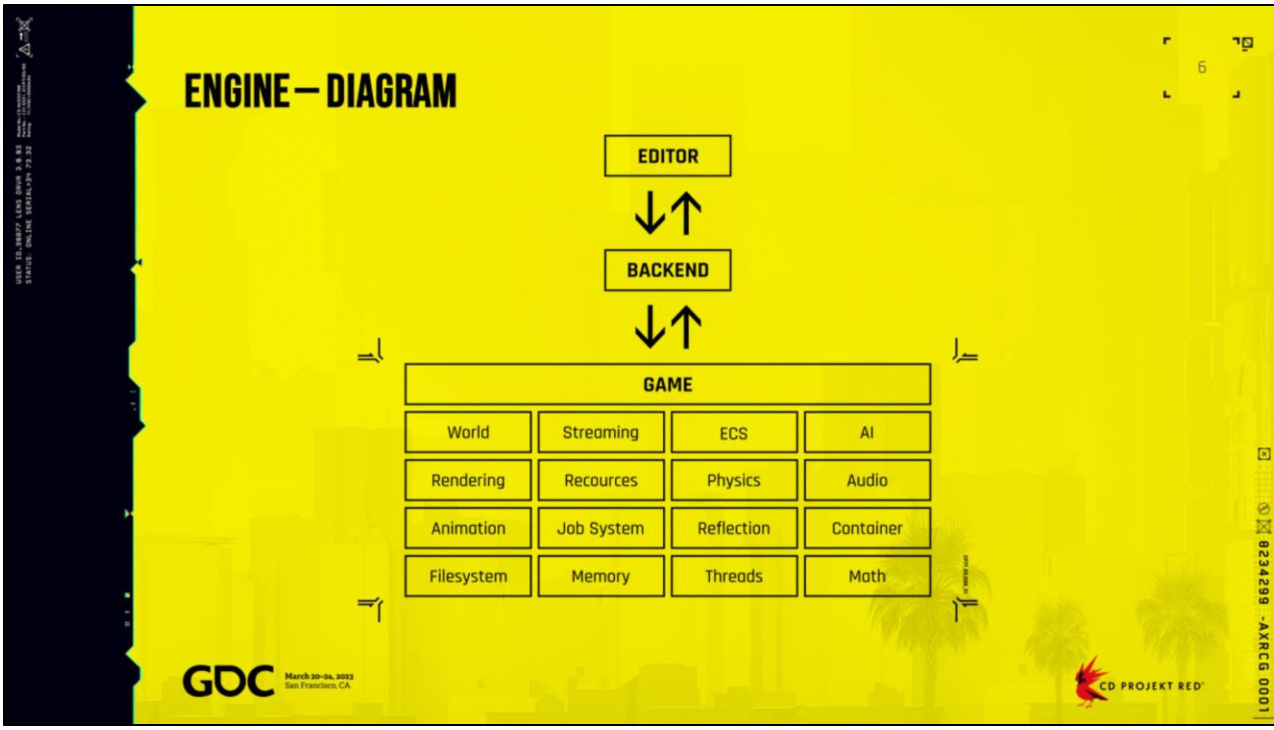
- Night City – Living metropolis in a dystopian future
  - 16x16km world and vertical
  - High density
  - No loading screen
- Vehicles
  - 140km/hr
- Scalability
  - Various platform support
  - From Xbox One to High-End PC

RED ENGINE PILLARS

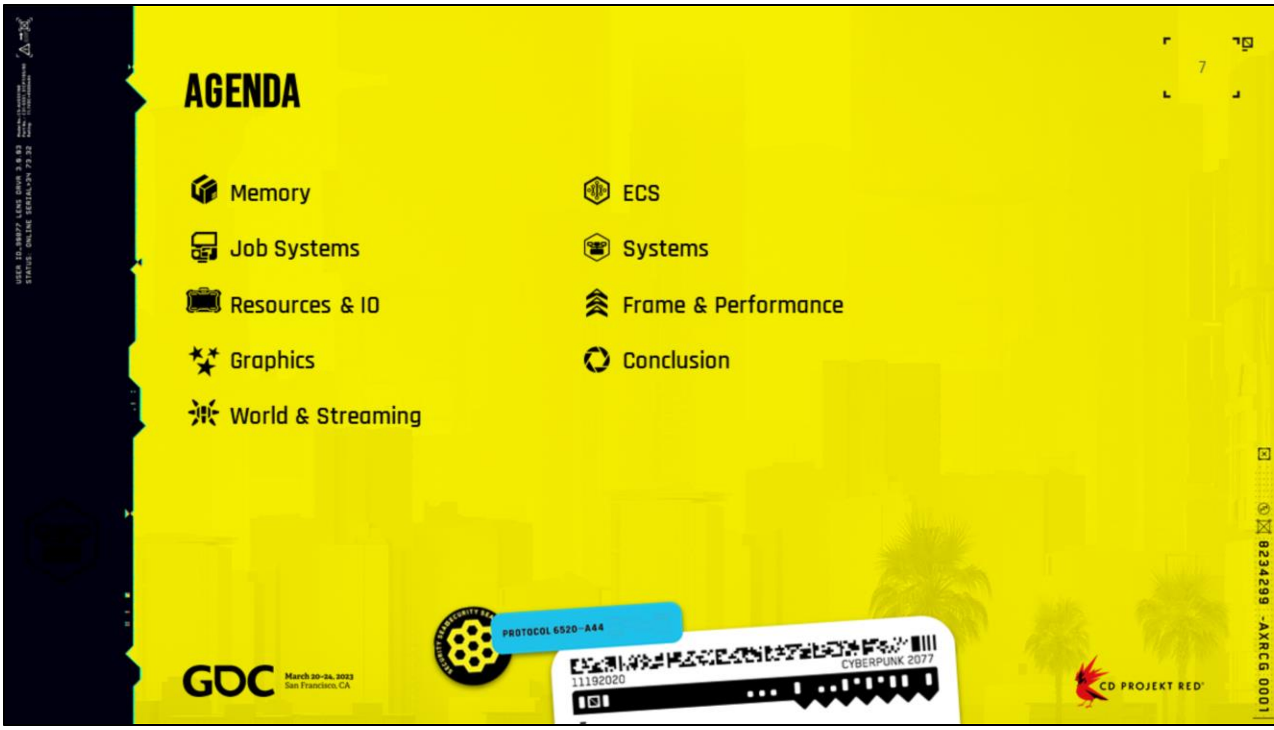
- **Engine / Gameplay agnostic of Editor/Tools code**
  - RPC backend ⇔ Editor
- **Systems Scalability**
  - Maximize platform hardware utilization
  - Adapt according to game state or current quest
- **Quest is King**

RED ENGINE RULES

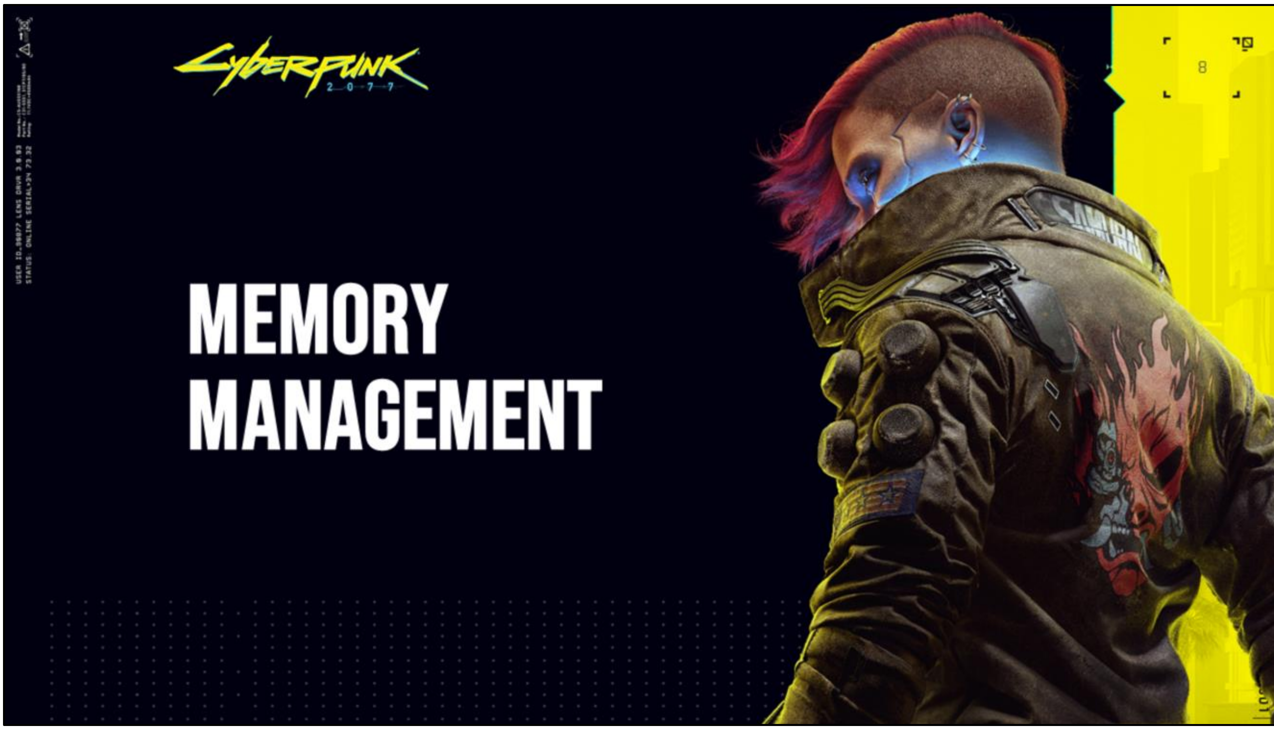
- No code can actively wait for anything
- Code should assume it never runs in isolation
- Runtime memory allocation limited to <= 512 bytes
- No STL containers, Exception, RTTI



ENGINE — DIAGRAM
6
EDITOR
BACKEND
GAME
World
Streaming
ECS
AI
Rendering
Recources
Physics
Audio
Animation
Job System
Reflection
Container
Filesystem
Memory
Threads
Math
GDC
March 20–24, 2023
San Francisco, CA
CD PROJEKT RED
8234299 -AXRCG 0001



AGENDA
7
Memory
Job Systems
Resources & IO
Graphics
World & Streaming
ECS
Systems
Frame & Performance
Conclusion
PROTOCOL 6520-A44
CYBERPUNK 2077
11192020
GDC March 20-24, 2023 San Francisco, CA
CD PROJEKT RED
8234299 -AXRCG 0001


Cyberpunk
2077
MEMORY
MANAGEMENT

MEMORY – REQUIREMENTS

- Allocators for specific needs
- Default allocator is lockless while minimizing fragmentation
- Easy to use and understand
- Easy to extend
- Budgets are easy to define
- Every allocator needs to be under the proper budgets
- Reliable

MEMORY – ALLOCATOR FOR SPECIFIC NEEDS

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

MEMORY – DEFAULT ALLOCATOR

- Custom Slab allocator with explicit thread registration for <= 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!

```cpp
/////////////////////////////////////////////////////////////////////////////
//
// 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::StaticTLSFAllocatorParameter param = { allocatorBuffer, allocatorBufferSize };
    red::memory::StaticTLSFAllocator 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 );
}
```

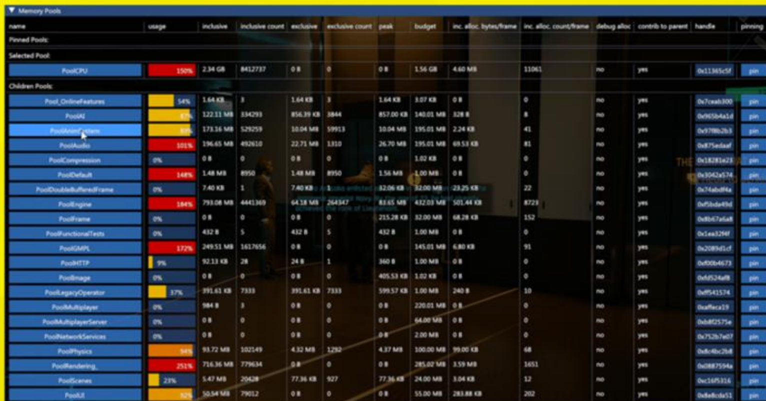
MEMORY – POOLS

- 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



▼ Memory Pools

name	usage	inclusive	inclusive count	exclusive	exclusive count	peak	budget	inc. alloc. bytes/frame	inc. alloc. count/frame	debug alloc	contrib to parent	handle	pinning
Pinned Pools:													
Selected Pool:													
PoolCPU	150%	2.34 GB	8412737	0 B	0	0 B	1.56 GB	4.60 MB	11061	no	yes	0x11365c5f	pin
Children Pools:													
Pool_OnlineFeatures	54%	1.64 KB	3	1.64 KB	3	1.64 KB	3.07 KB	0 B	0	no	yes	0x7ceab300	pin
PoolAI	87%	122.11 MB	334293	856.39 KB	3844	857.00 KB	140.01 MB	328 B	8	no	yes	0x965b4a1d	pin
PoolAnimSystem	89%	173.16 MB	529259	10.04 MB	59913	10.04 MB	195.01 MB	2.24 KB	41	no	yes	0x97f8b2b3	pin
PoolAudio	101%	196.65 MB	492610	22.71 MB	1310	26.70 MB	195.01 MB	69.53 KB	81	no	yes	0x875edaaf	pin
PoolCompression	0%	0 B	0	0 B	0	0 B	1.02 KB	0 B	0	no	yes	0x18281a23	pin
PoolDefault	148%	1.48 MB	8950	1.48 MB	8950	1.56 MB	1.00 MB	0 B	0	no	yes	0x3042a574	pin
PoolDoubleBufferedFrame	0%	7.40 KB	1	7.40 KB	1	32.06 KB	32.00 MB	23.25 KB	22	no	yes	0x74abdf4a	pin
PoolEngine	184%	793.08 MB	4441369	64.18 MB	264347	83.65 MB	432.03 MB	501.44 KB	8723	no	yes	0xf5bda49d	pin
PoolFrame	0%	0 B	0	0 B	0	215.28 KB	32.00 MB	68.28 KB	152	no	yes	0x8b67a6a8	pin
PoolFunctionalTests	0%	432 B	5	432 B	5	432 B	1.00 MB	0 B	0	no	yes	0x1ea329f	pin
PoolGMPL	172%	249.51 MB	1617656	0 B	0	0 B	145.01 MB	6.80 KB	91	no	yes	0x2089d1cf	pin
PoolHTTP	9%	92.13 KB	28	24 B	1	360 B	1.00 MB	0 B	0	no	yes	0xf00b4673	pin
PoolImage	0%	0 B	0	0 B	0	405.53 KB	1.02 KB	0 B	0	no	yes	0xd524af8	pin
PoolLegacyOperator	37%	391.61 KB	7333	391.61 KB	7333	599.57 KB	1.00 MB	240 B	10	no	yes	0xff541574	pin
PoolMultiplayer	0%	984 B	3	0 B	0	0 B	220.01 MB	0 B	0	no	yes	0xaffeca19	pin
PoolMultiplayerServer	0%	0 B	0	0 B	0	0 B	64.00 MB	0 B	0	no	yes	0xb8f2575e	pin
PoolNetworkServices	0%	0 B	0	0 B	0	0 B	2.00 MB	0 B	0	no	yes	0x752b7a07	pin
PoolPhysics	94%	93.72 MB	102149	4.32 MB	1292	4.37 MB	100.00 MB	99.00 KB	68	no	yes	0x8c4bc2b8	pin
PoolRendering_	251%	716.36 MB	779634	0 B	0	0 B	285.02 MB	3.59 MB	1651	no	yes	0x0887594a	pin
PoolScenes	23%	5.47 MB	20428	77.36 KB	927	77.36 KB	24.00 MB	3.04 KB	12	no	yes	0xc16f5316	pin
PoolUI	92%	50.54 MB	79012	0 B	0	0 B	55.00 MB	283.88 KB	202	no	yes	0x8e8cda51	pin

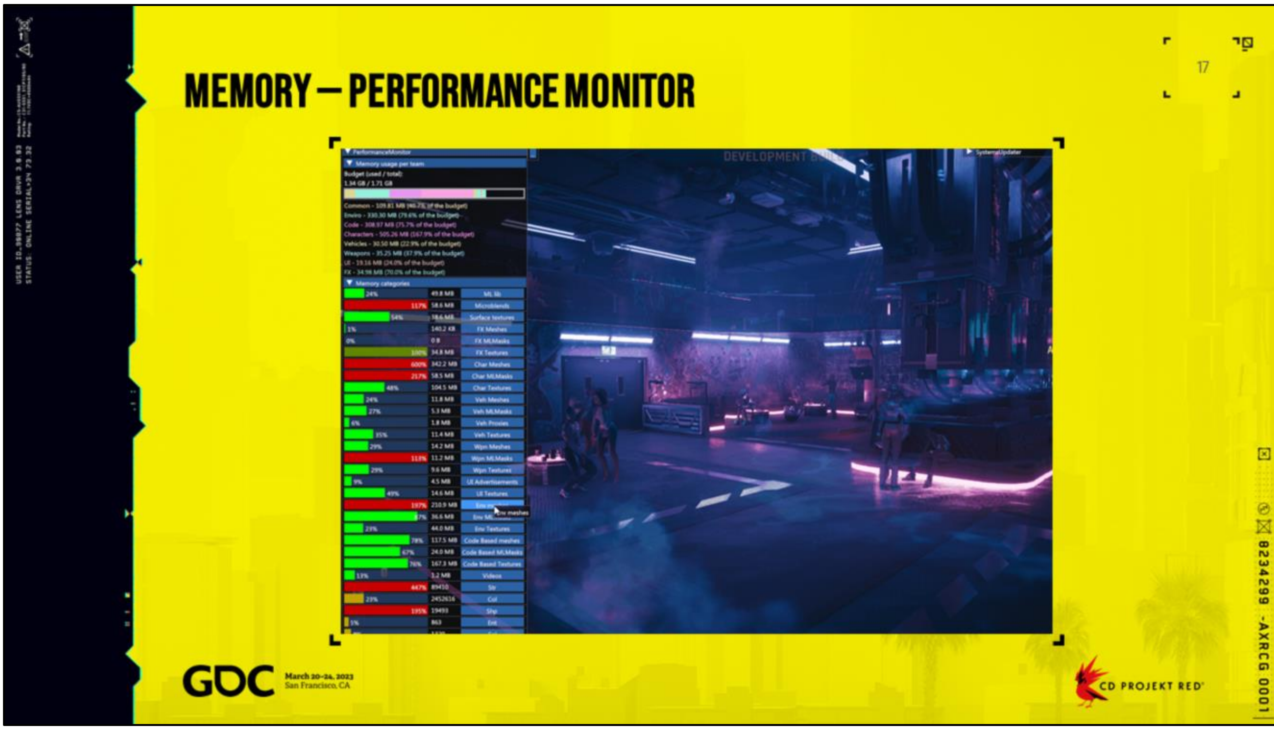
MEMORY – BUDGETS

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



- 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

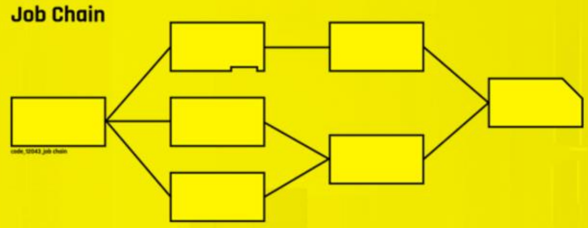
194mb Environment Textures	33mb Weapon Textures	33mb Vehicle Textures	35mb FX Textures	221mb Code Based Textures	72mb Surface Textures
107mb Environment Meshes	50mb Weapon Meshes	50mb Vehicle Meshes	10mb FX Meshes	151mb Code Based meshes	210mb Multilayer library
42mb Environment MLMasks	10mb Weapon MLMasks	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				

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

```
class HeavyObject
{
public:
    void Uninitialize() {}
};

void FireAndForgetSnippet( red::UniquePtr< HeavyObject > object )
{
    job::Builder builder( job::Priority::Latent );
    builder.DispatchJob( "Uninitialize_and_Destroy_Object", [obj = std::move(object)]( const job::RunContext& )
    {
        obj->Uninitialize();
        obj.Reset();
    } );
}
```

JOB BUILDER – SIMPLE JOB CHAIN

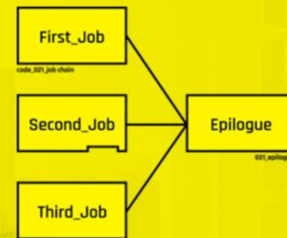
DispatchJob creates dependant job by default


```
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

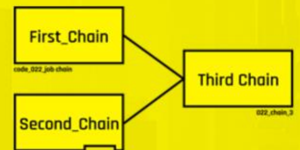
```cpp
void ParallelJobWithEpilogueJobSnippet( const red::SharedPtr< Object >& object )
{
    job::Builder builder( job::Priority::CriticalPath );

    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

```cpp
void LinkingJobGraphSnippet( 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();
    } );
}
```



CD PROJEKT RED

JOB BUILDER – HOW TO DO A CONTINUATION JOB

Job can be dispatched as a continuation of the current job

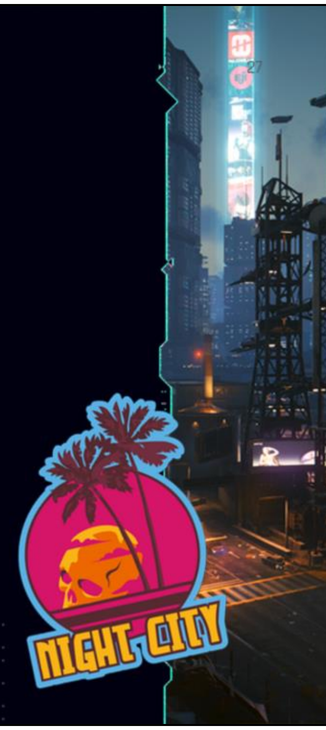
```
void ContinuationJobSnippet( const red::SharedPtr< Object >& object )
{
    job::Builder builder( job::Priority::CriticalPath );
    builder.DispatchJob( "First_Job", [ object ]( const job::RunContext& context )
    {
        object->FirstJob();
        job::Builder builder( context );
        builder.DispatchJob(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

CD PROJEKT RED

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

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

- 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 = io::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

```cpp
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;
}
```

IO SCHEDULER

- Fixed memory buffer for IO and Decompression
  - 64mb on console, 128mb on PC
  - If no memory is available, IO will be put on hold until it is available
- Resource dependencies are known and scheduled up front
- Async IO on all platform
- DirectStorage on XSX/XSS (and equivalent on PS5)

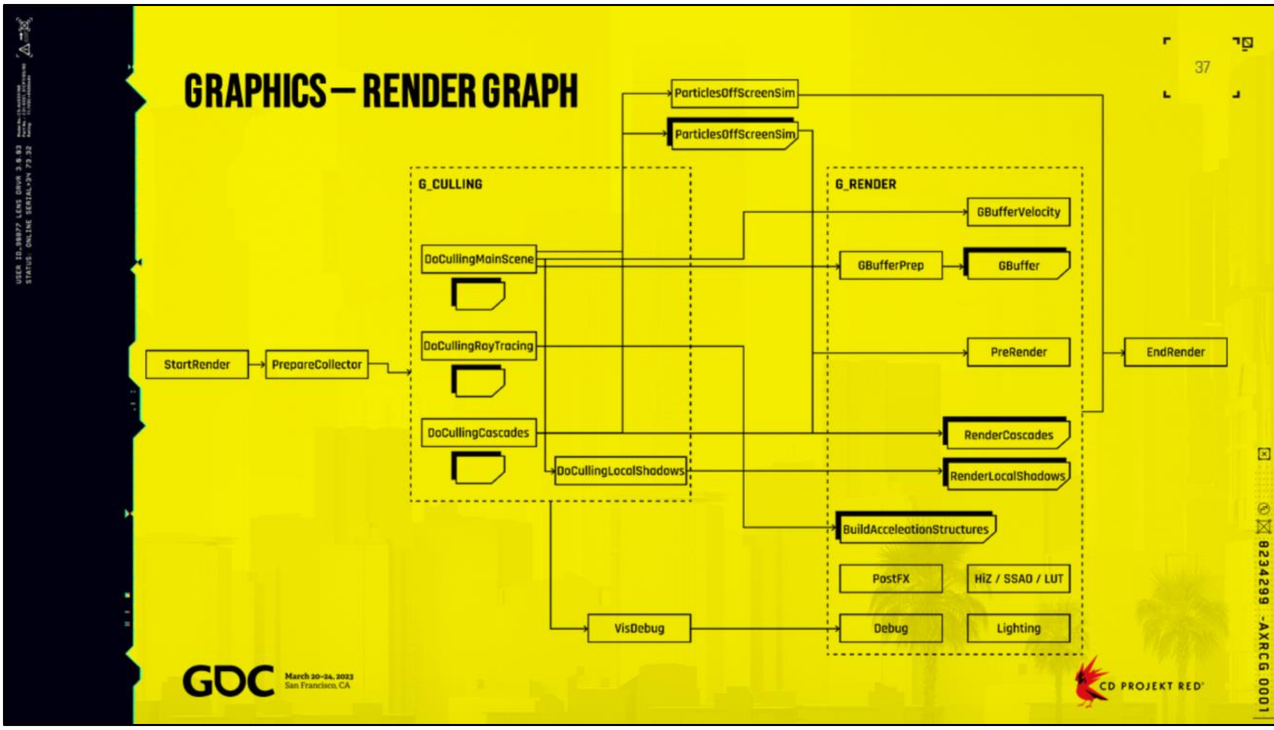
Very good presentation by Tim Green, SIGGRAPH 2021

GRAPHICS – FROM THE WITCHER 3 TO CYBERPUNK 2077

- **DirectX 11 to DirectX 12**
  - "Emulation" layer was developed first
  - Brunt of the work was done on a branch for 6 months
- **Breaking up the Monolithic Render Thread**
  - Introducing Render Graphs
  - Fully parallelized, using job system
- **Multilayer texturing**
  - Extensive material library
  - Texture accessed through Bindless
  - Reduce IO pressure
- **Control over memory**

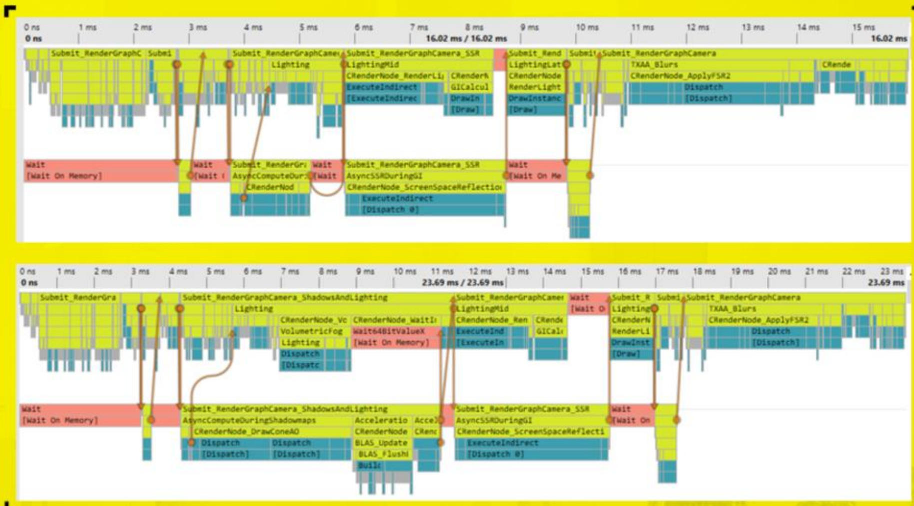
GRAPHICS – RENDER GRAPH

- 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



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

CD PROJEKT RED

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

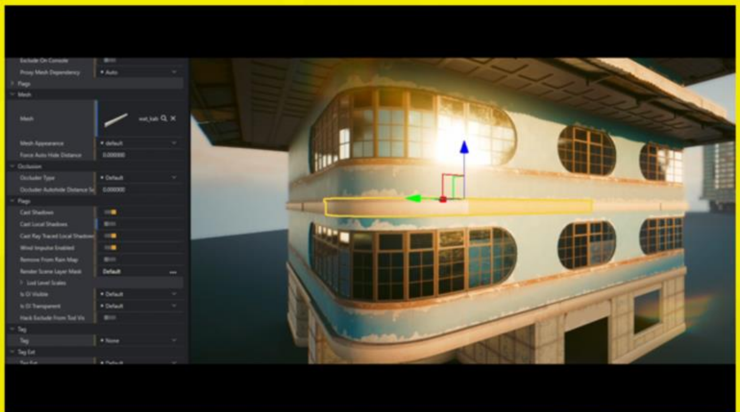
- NodeInstances are the units that are streamed in
- NodeInstances are NOT updated directly
- Node is the payload provided to an instance when streaming in
- Node can be shared to multiple node instances

```
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;
};
```

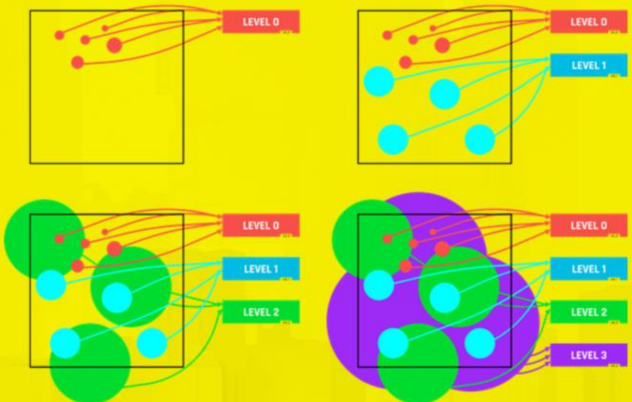
WORLD NODE & NODEINSTANCE

```cpp
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 ) );
}
```

PREFABS EDITOR



STREAMING GRID – DEVELOPMENT



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

GDC March 20-24, 2023 San Francisco, CA

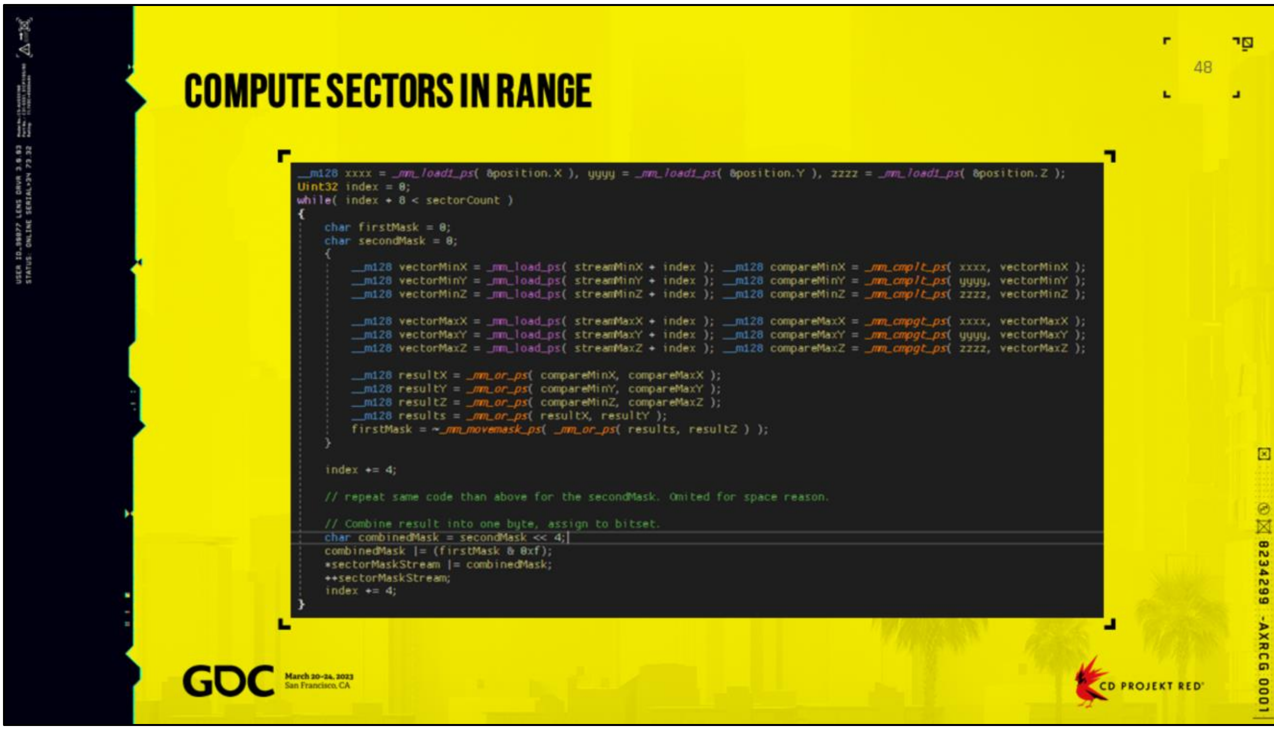
CD PROJEKT RED

STREAMING GRID – OPTIMIZED

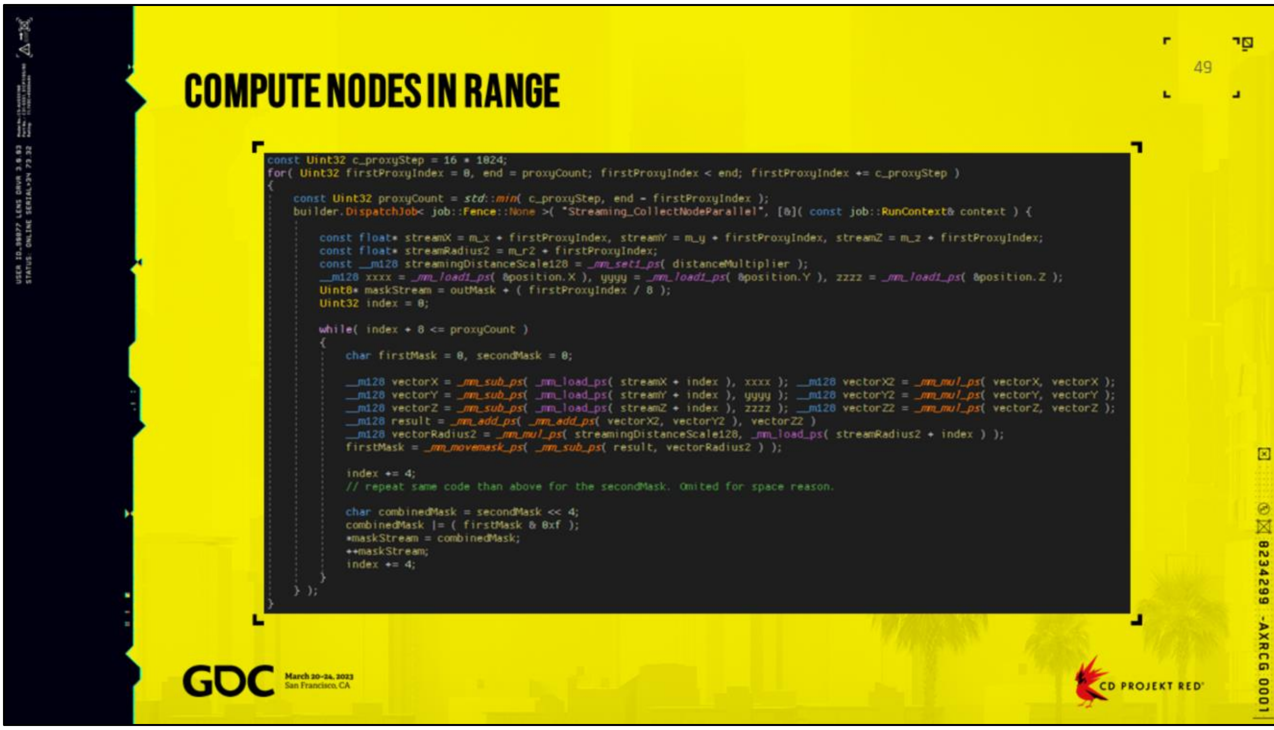
- Sector dimensions are much smaller
  - 64m cube for exteriors
  - 32m cube for interiors
- Sectors are rebalanced to eliminate "almost empty" sectors
  - Nodes will always "move up" to higher level sector
- Quest sectors are now merged into a single sector per quest
- Resources are now embedded in 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

RUNTIME STREAMING PROCESS

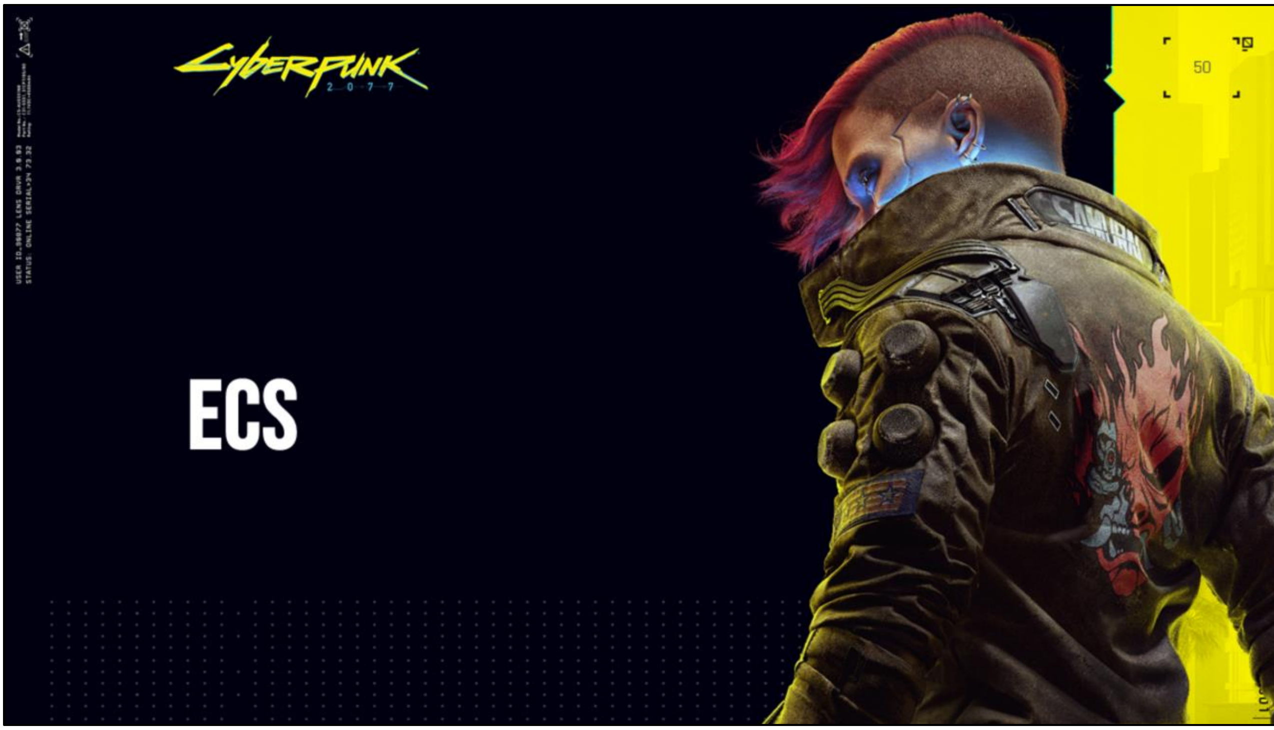
1. **Compute which sector needs to be loaded / unloaded**
    a. Request Async Load on sector in range
    b. Remove sector from grid that needs to be unloaded
    c. Add loaded sector to grid
2. **Compute which nodes that need to be streamed-in / streamed-out**
    a. Request each node in range to start 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**



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



More or less 50us per job on PS4


Cyberpunk
2077
ECS
50

ECS – RULES
51
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
GDC March 20-24, 2023 San Francisco, CA
CD PROJEKT RED
8234299 -AXRCG 0001

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

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

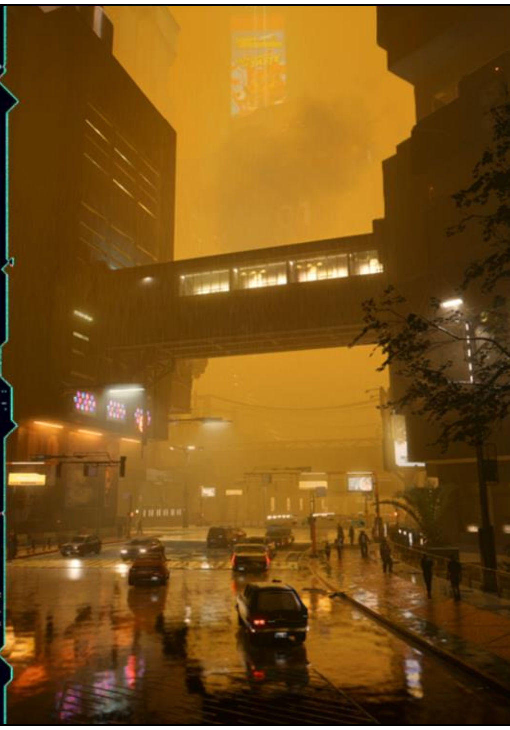
```
void MeatBag::OnPhysicalImpulse( const PhysicalImpulseEvent& evt )
{
    if( evt.m_bodyIndex == m_bagBodyIndex ) {
        ent::SpawnEffectSetup effectSetup;
        if( --m_hitPoints > 0 ) {
            effectSetup.effectName = m_jiggleEffectName;
        }
        else {
            m_physicalComponent->ToggleCollisions( 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 ) )
    }
}
```

ECS – ASYNC SPAWNING

- There is no sync spawn request
- However it does not need to be attached to world
- Scheduling to attach to world is guaranteed to be done before next frame start

ECS – ASYNC SPAWNING

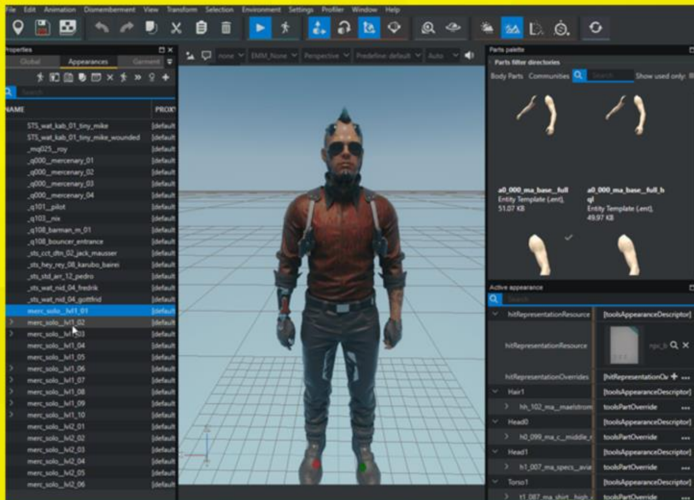

```
const RuntimeScene* scene = AcquireRuntimeScene();
RuntimeSystemEntity* entitySystem = scene->GetSystem< RuntimeSystemEntity >();
ent::EntitySpawnService* spawnService = entitySystem->AcquireSpawnService();

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 } );
    }
} );
```



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

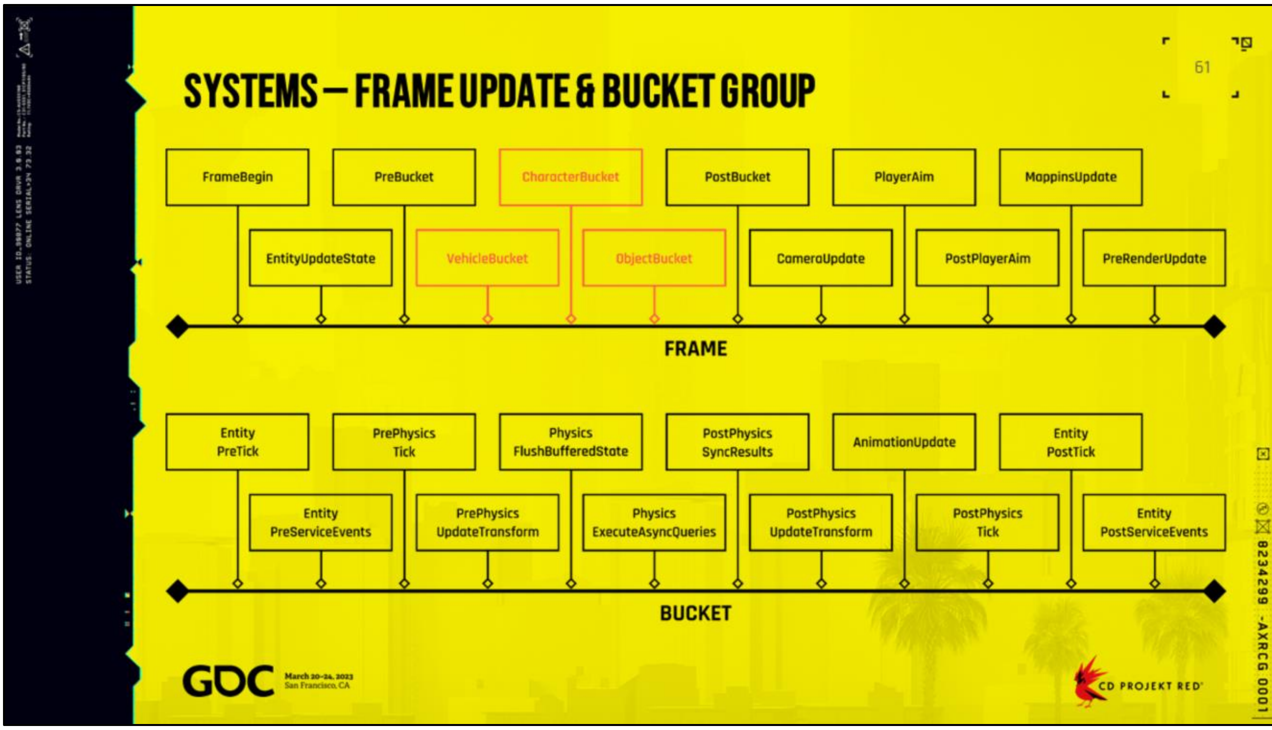
ECS – APPEARANCE EDITOR



SYSTEMS

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



SYSTEMS – FRAME UPDATE & BUCKET GROUP
61
FrameBegin
PreBucket
CharacterBucket
PostBucket
PlayerAim
MappinsUpdate
EntityUpdateState
VehicleBucket
ObjectBucket
CameraUpdate
PostPlayerAim
PreRenderUpdate
FRAME
Entity PreTick
PrePhysics Tick
Physics FlushBufferedState
PostPhysics SyncResults
AnimationUpdate
Entity PostTick
Entity PreServiceEvents
PrePhysics UpdateTransform
Physics ExecuteAsyncQueries
PostPhysics UpdateTransform
PostPhysics Tick
Entity PostServiceEvents
BUCKET
GDC March 20-24, 2023 San Francisco, CA
CD PROJEKT RED

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

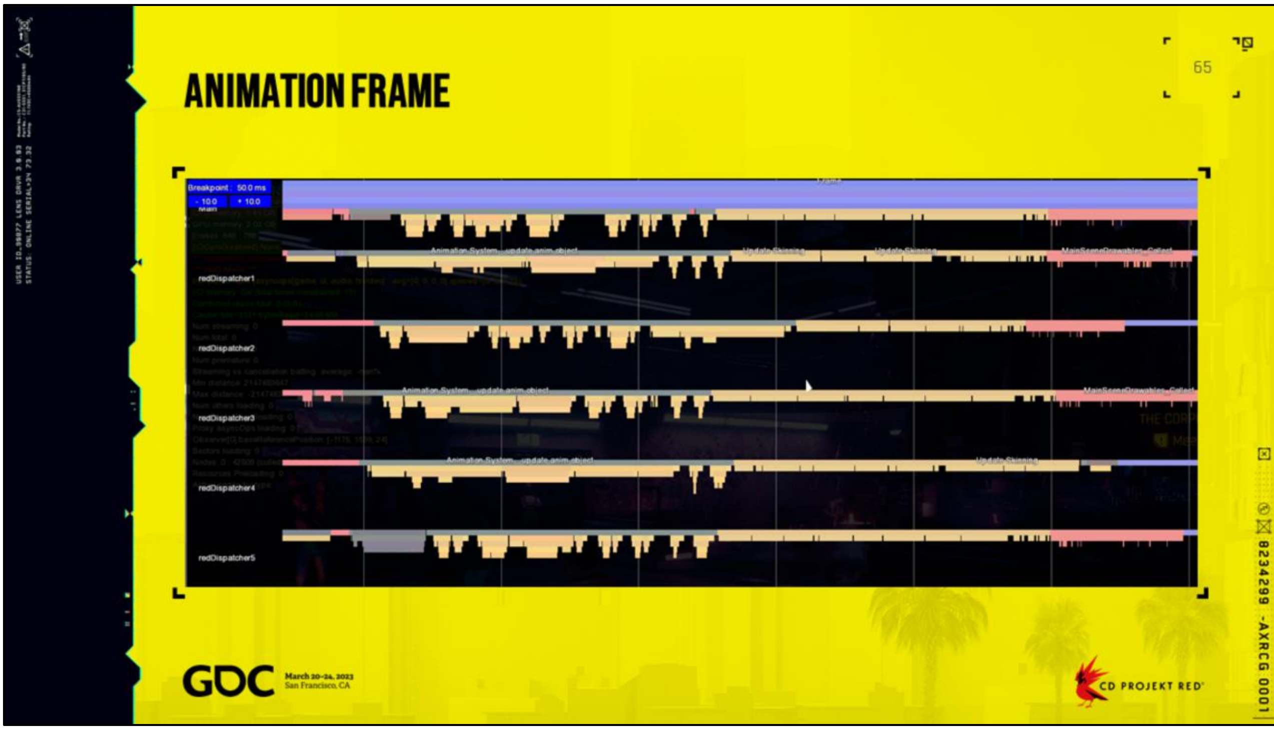
- Physics systems are built on top of PhysX
- PhysX tasks were adapted to be compatible with our job system
- Simple “C style” public API
- Safe to read and write states
- State writes are buffered and applied to all modified proxies at specific points in frame
- Concurrent writes could have been supported. 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 );
```

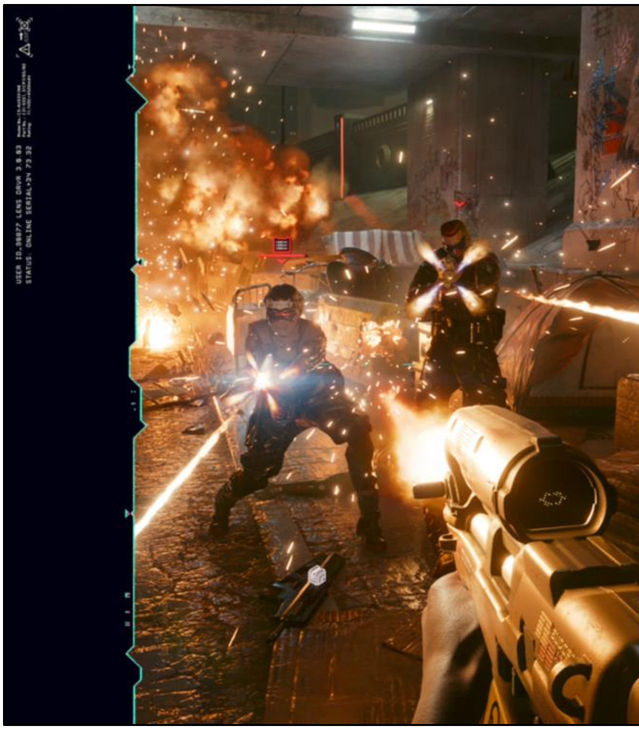
ANIMATION IN A NUTSHELL

- 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

CD PROJEKT RED



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

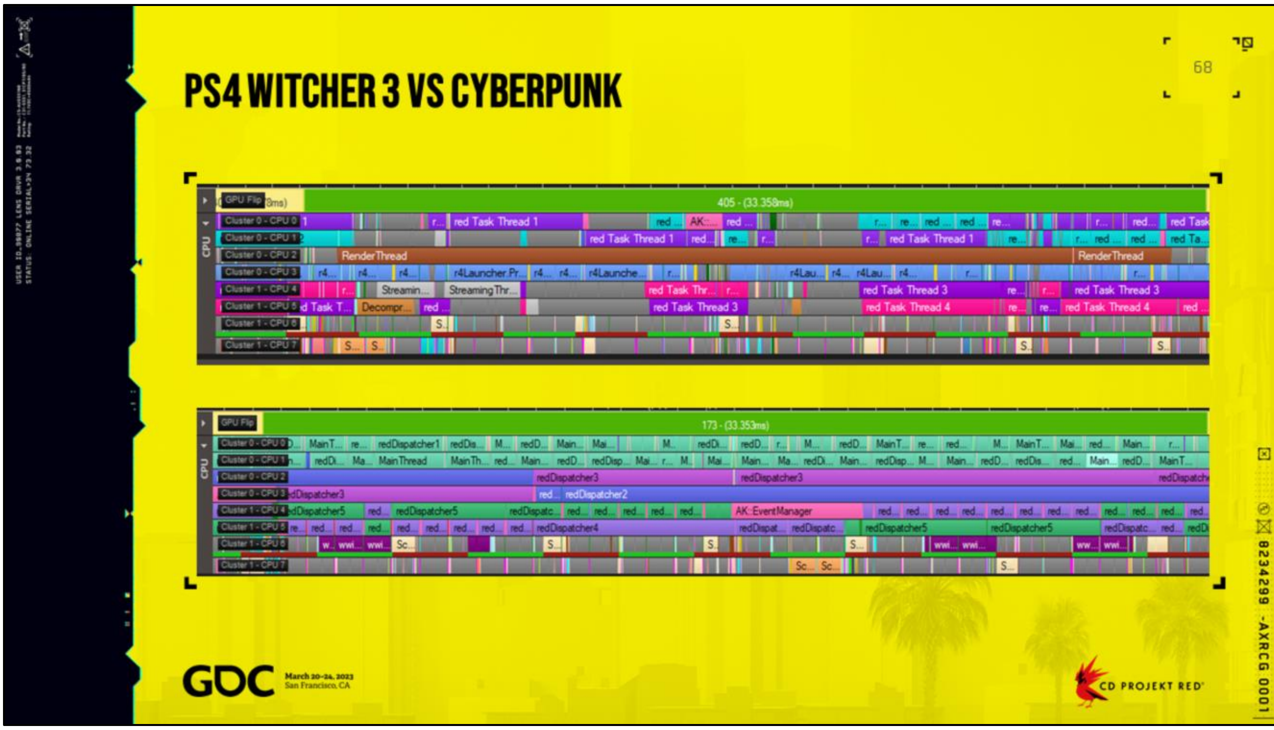


AI — LIVING ON THE EDGE

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

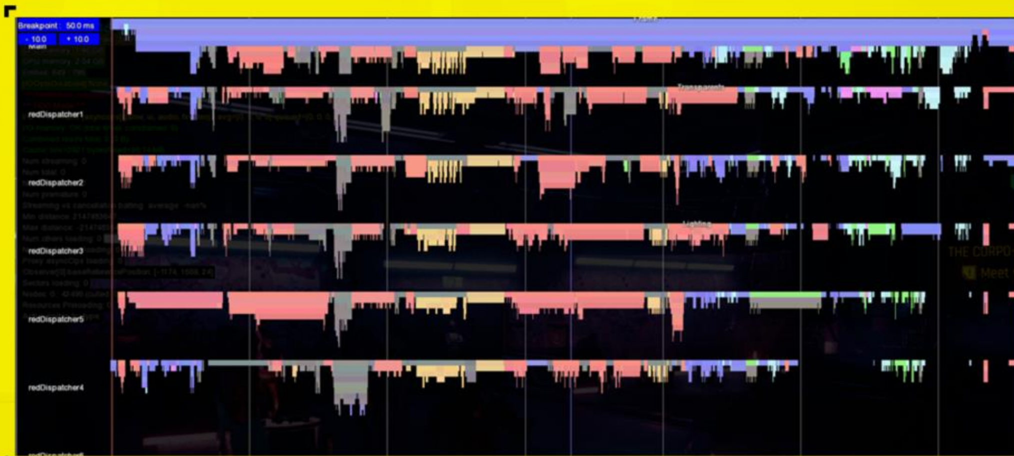
CD PROJEKT RED

You can change the img on the right

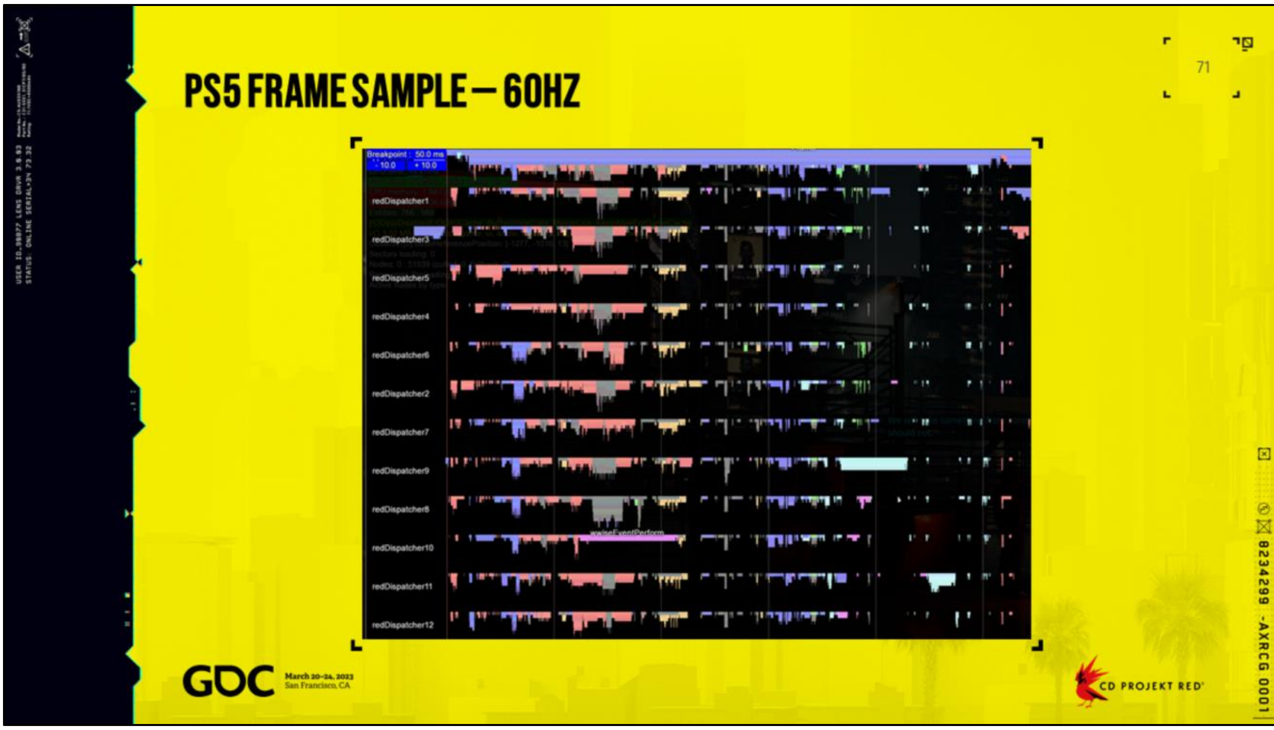


PS4, Witcher3 on top, Cyberpunk bottom one

PS4 FRAME SAMPLE

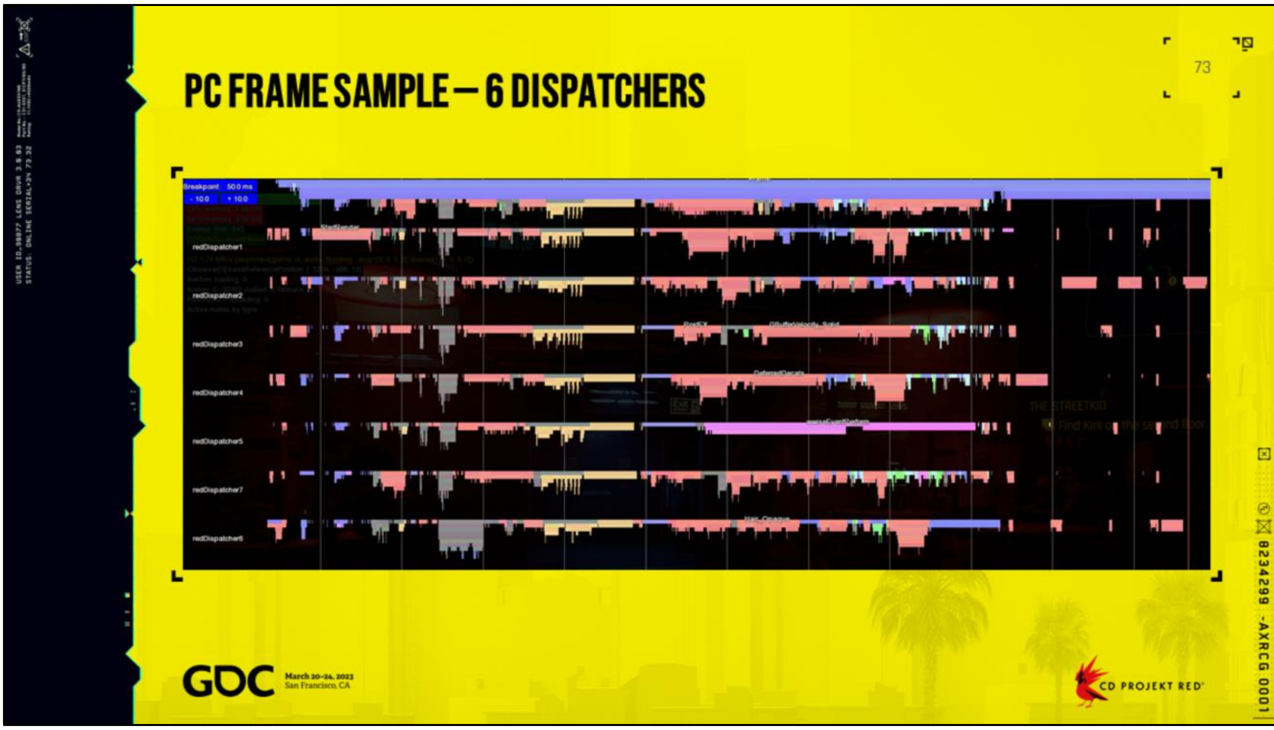


PS4 MULTIPLE FRAME SAMPLE

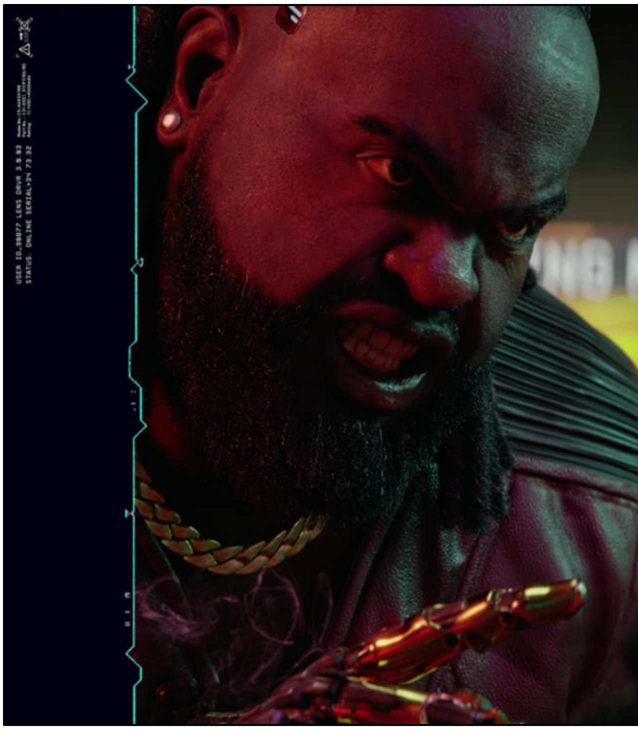


More or less 12ms on CPU

PS5 MULTIPLE FRAME SAMPLE – 60HZ



Force 6 dispatcher,
i9 - 7980XE @2.6ghz

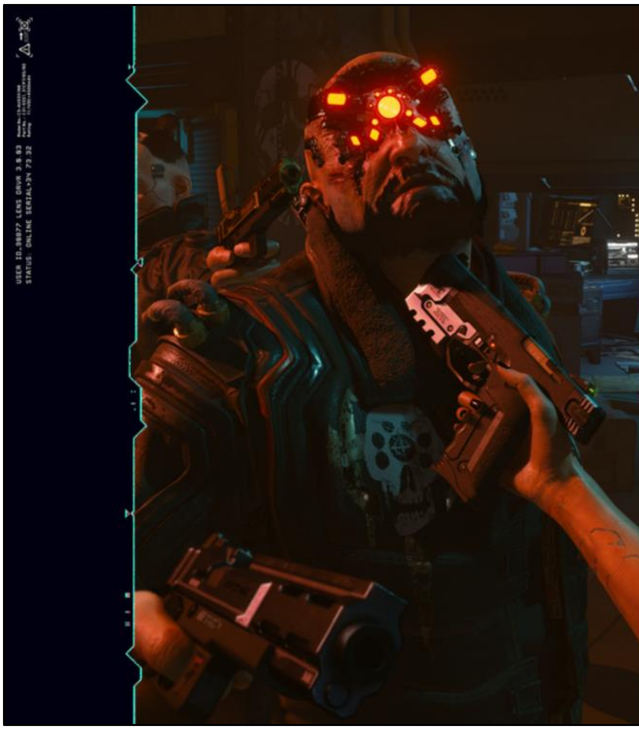


WHAT ABOUT THE MAIN THREAD?

- 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

CD PROJEKT RED

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

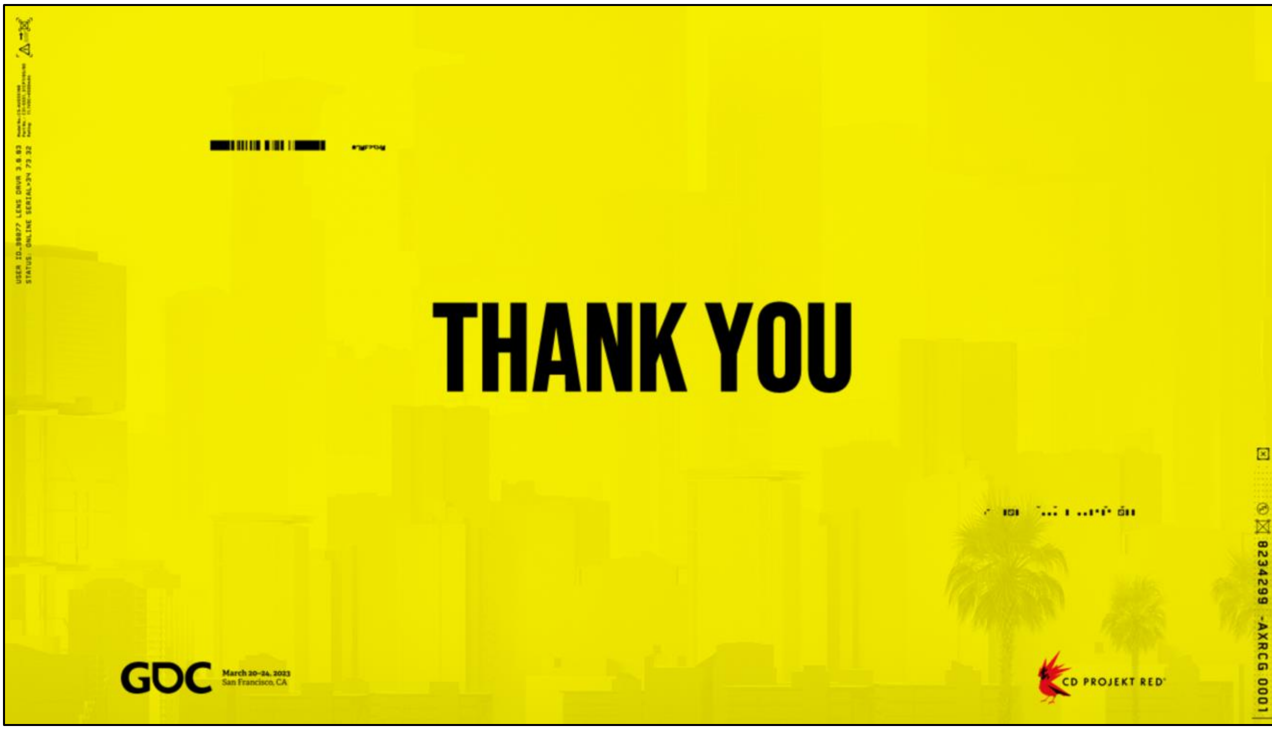
CD PROJEKT RED

WHAT WENT RIGHT ?

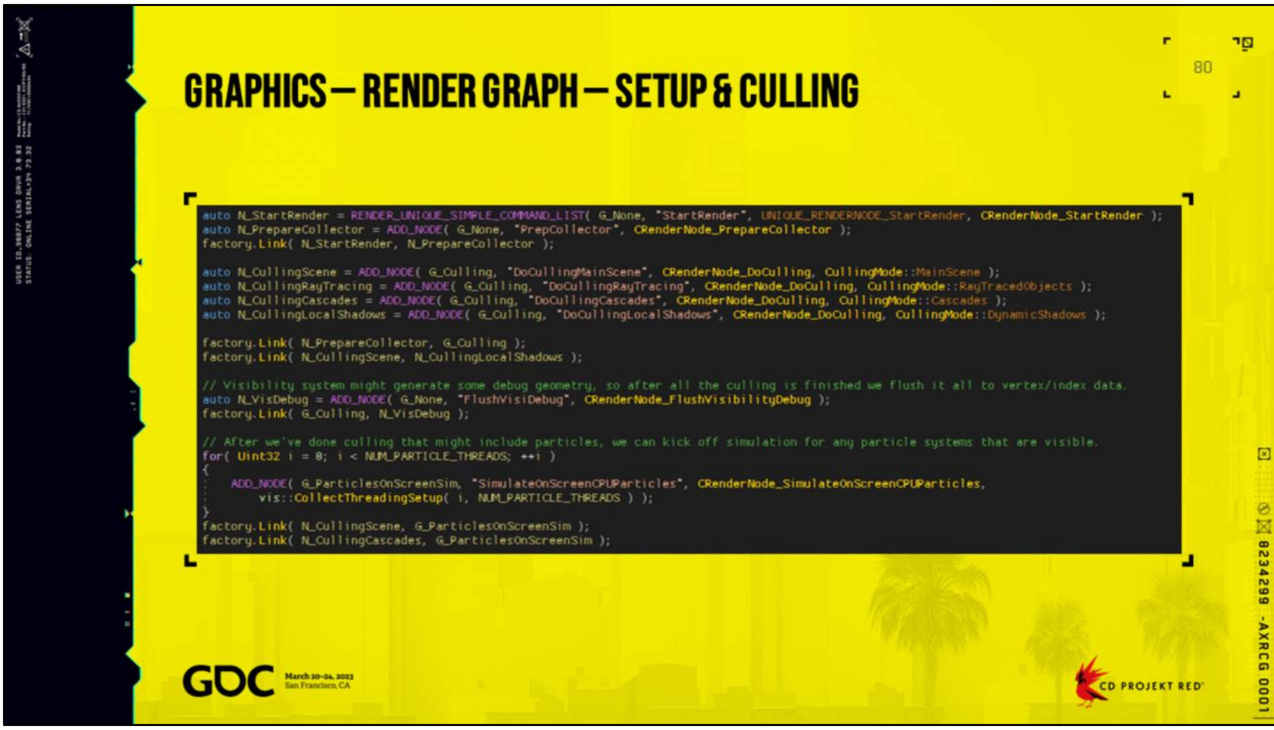
- The technological leap from The Witcher 3 to Cyberpunk was crazy
- PS5 & XSX/XSS port went smoothly
- No more main & render thread limitation
- Multithreaded gameplay & script
- Great scalability

CD PROJEKT RED

8234299 -AXRCG 0001


THANK YOU
GDC
March 20-24, 2023
San Francisco, CA
CD PROJEKT RED
8234299 -AXRCG 0001

SAMPLES



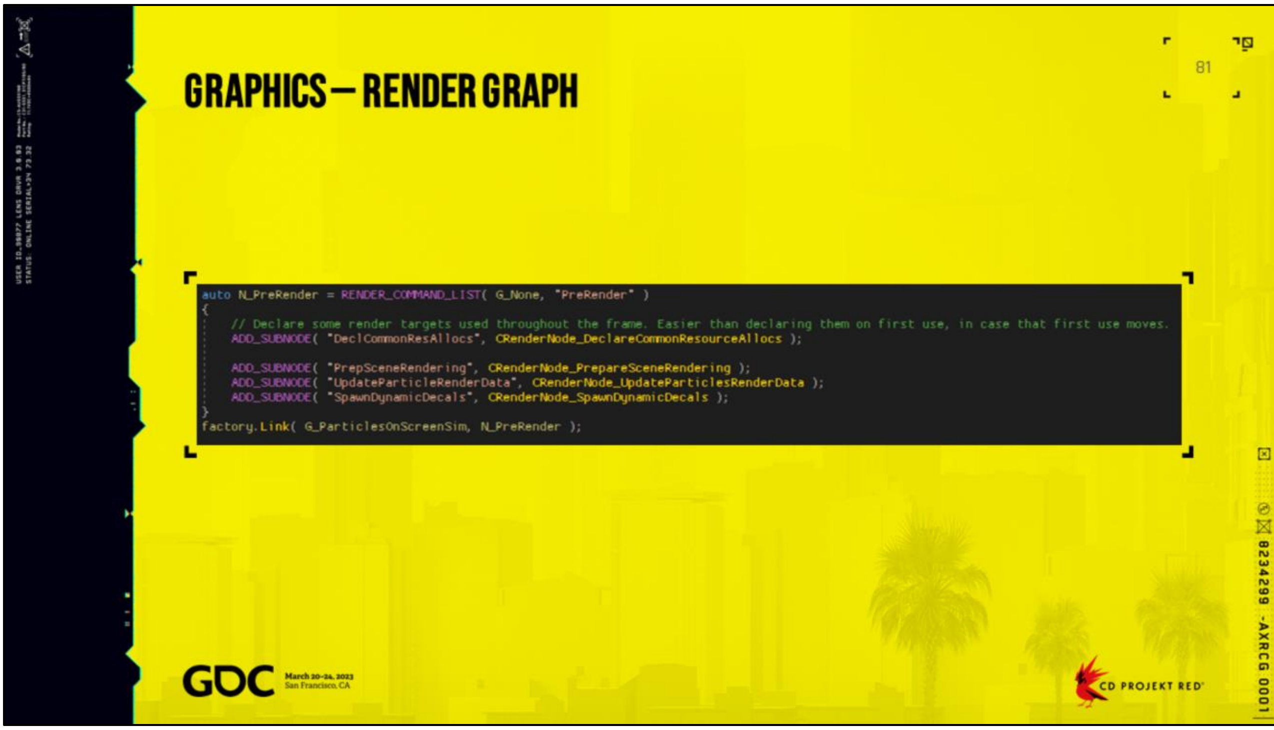
Set up for render, and kick off culling
"Unique" nodes are for special points in the frame. Not too important in this example, but used when merging multiple graphs -- unique stay unique (all unique from all subgraphs are de-duplicated), regular nodes are copied over.

"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).

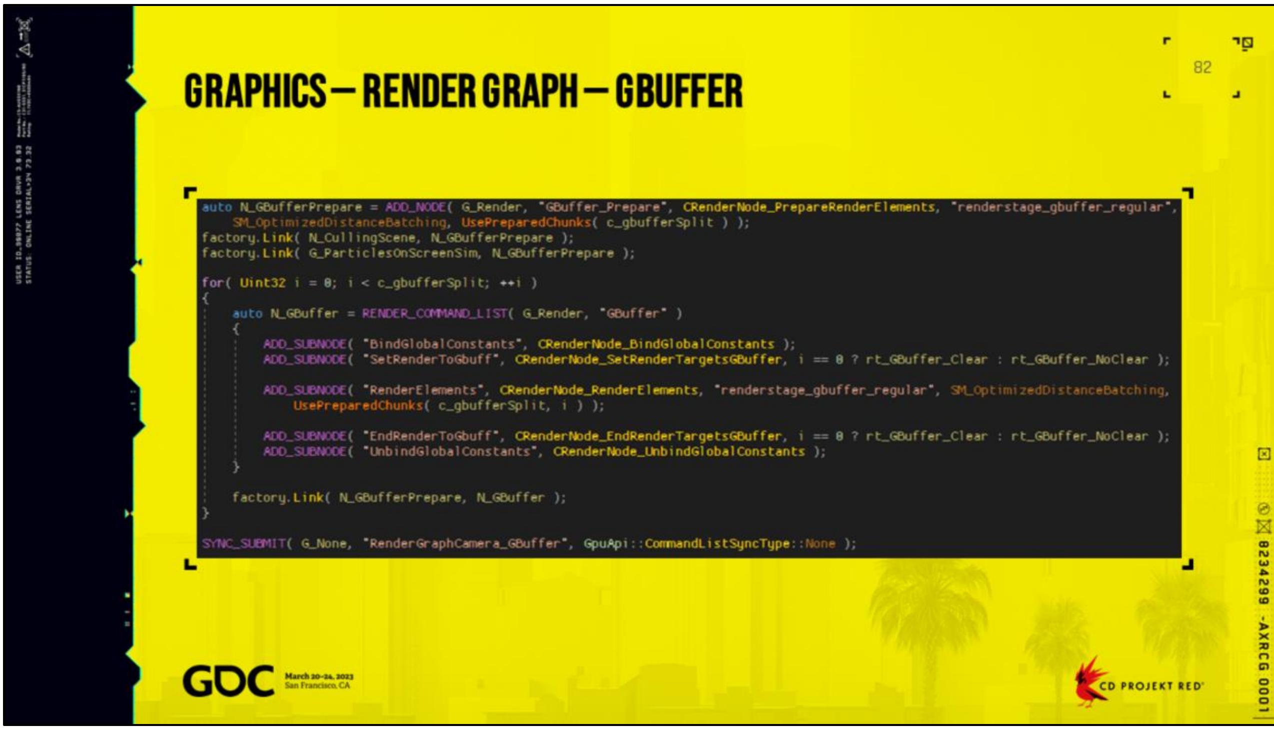
All culling needs to wait for PrepareCollector to finish.
In addition, LocalShadows needs to wait for main scene culling, in order to know what lights are visible.



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.



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.

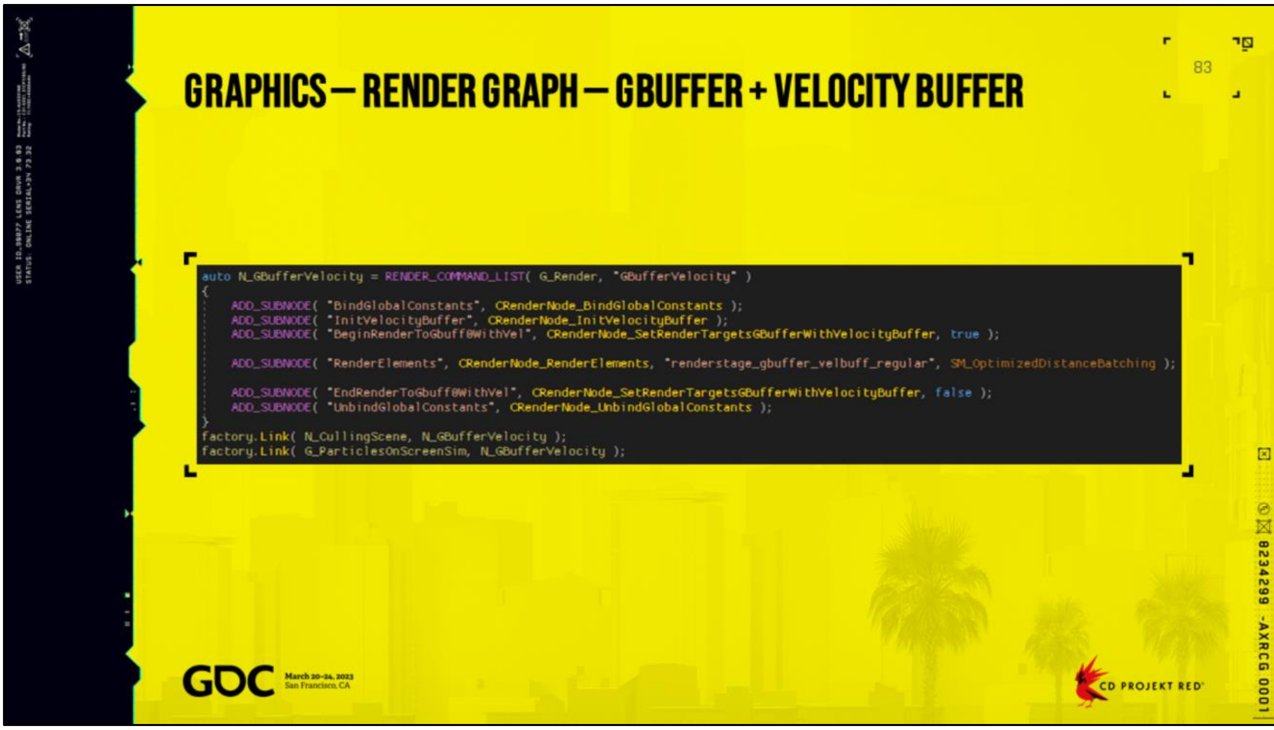
First a CPU-only node to build and sort the list of objects to be drawn. This needs the results of main scene culling, as well as on-screen particles.

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.

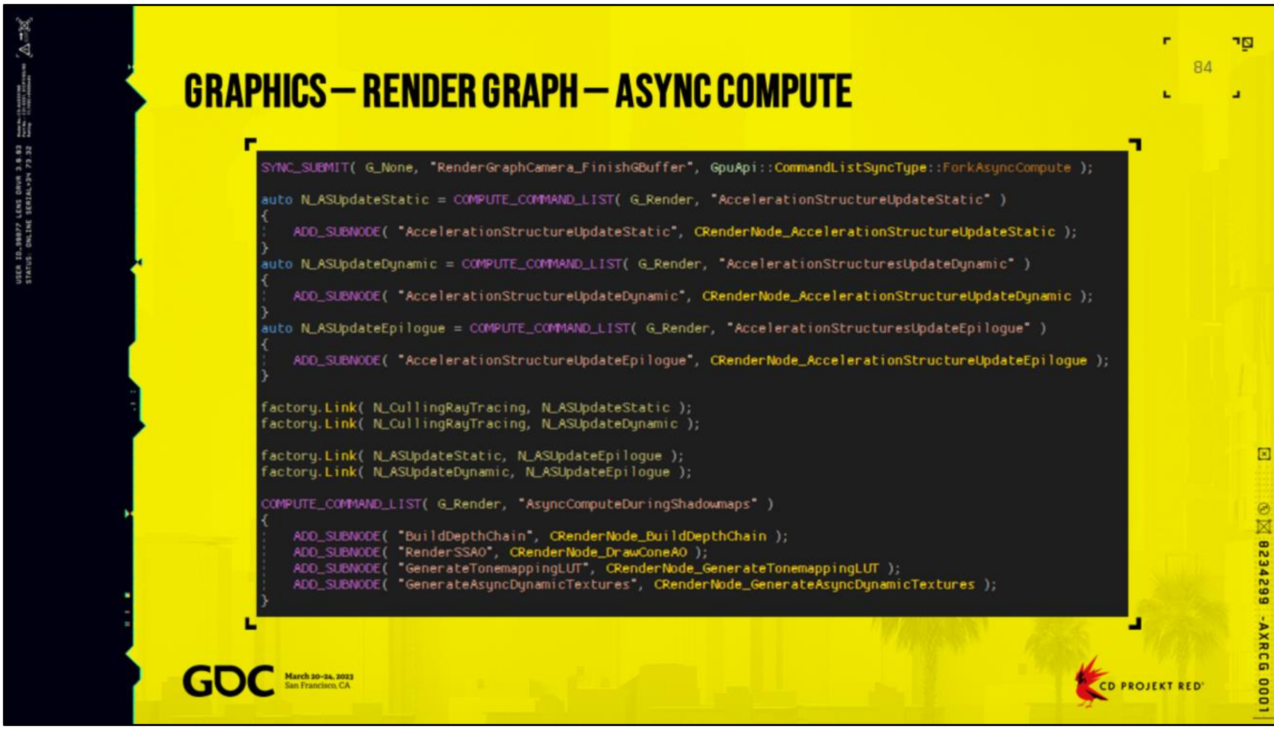
Since we have a pretty hefty amount of work built up with the GBuffer, we might want to submit it to the GPU already, so that it can keep busy with that while we prepare more.

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.



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.



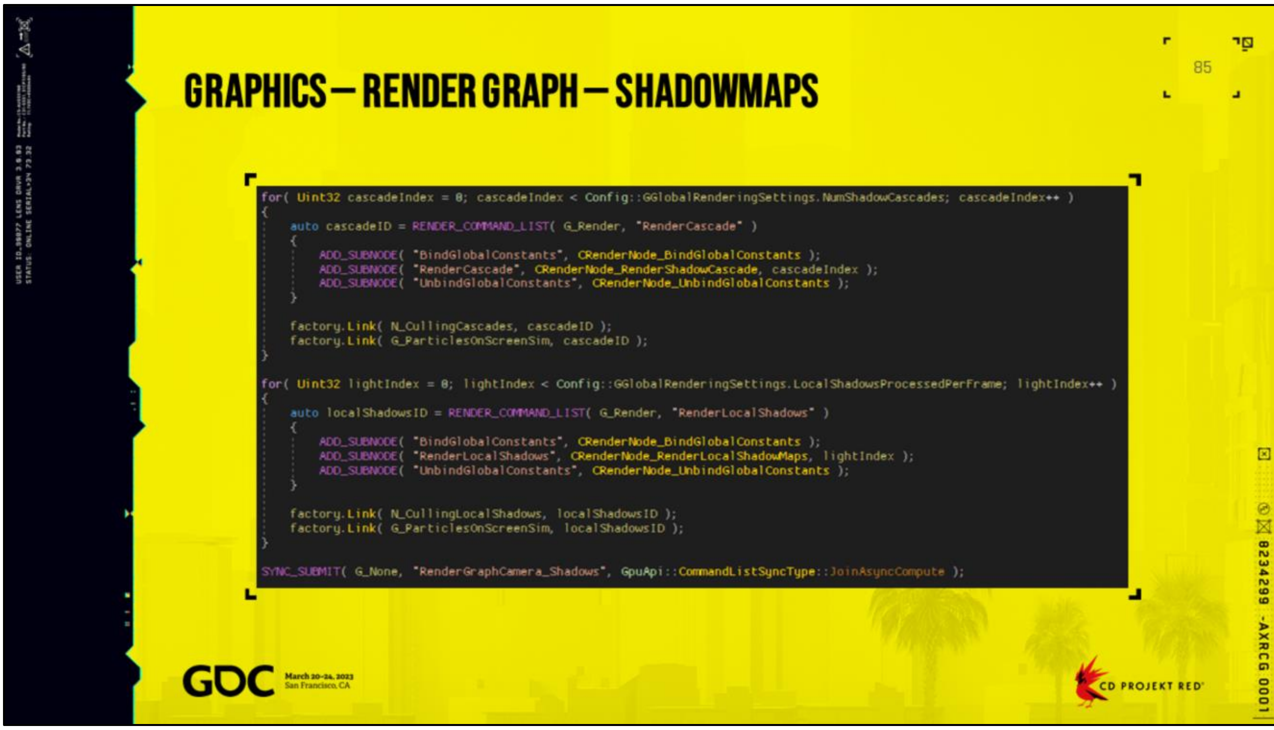
Async Compute work during shadowmap rendering

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.

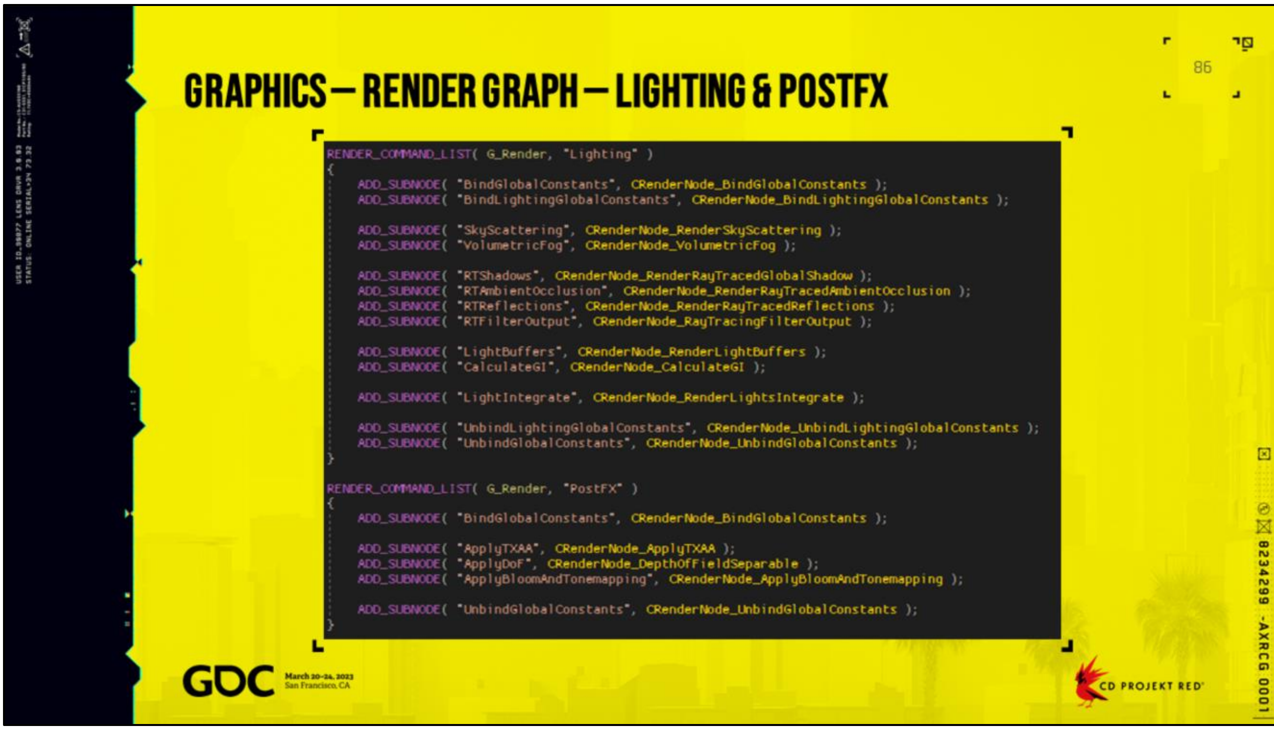
We also have another compute command lists for other compute workloads, there aren't any CPU dependencies so we don't need to link it to anything.



Shadowmaps

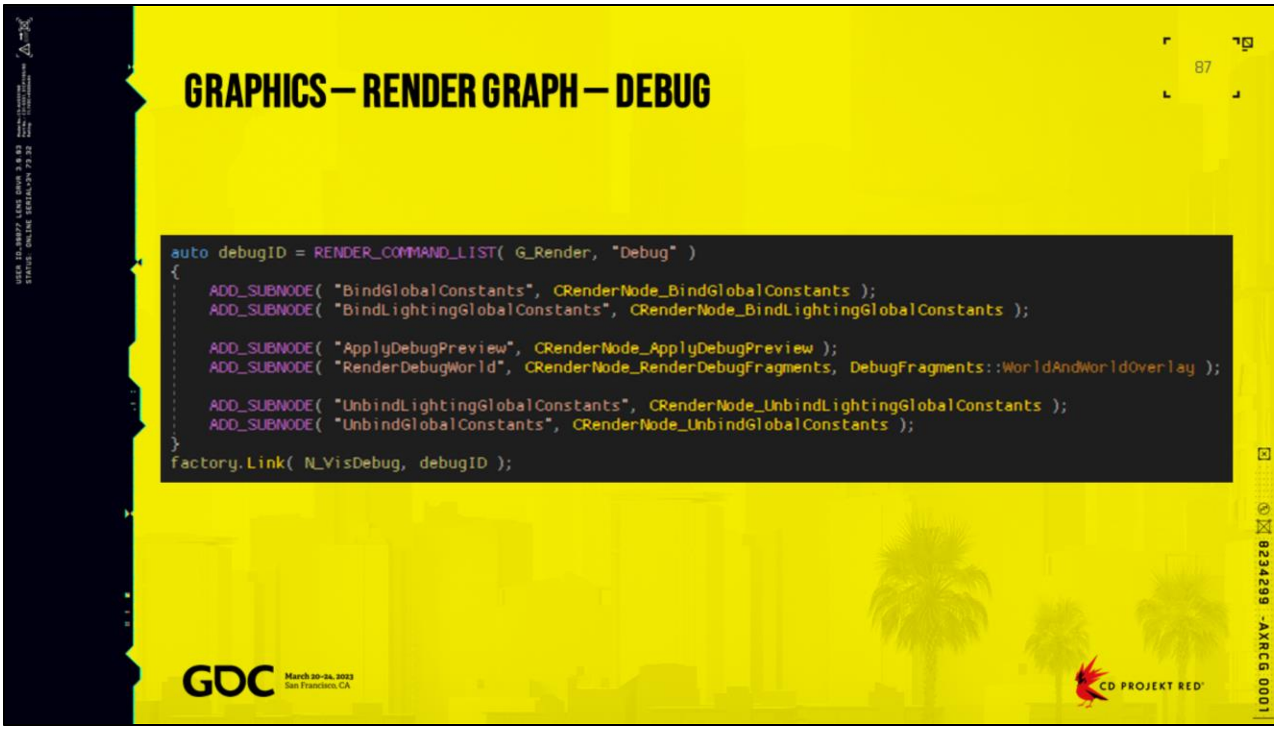
With the async compute work defined, we can do the shadowmap rendering.
There's a command list for each cascade, and for each local light we plan to update.
We need to wait for the appropriate culling to finish, as well as particle sim, in case there were some
shadow-casting mesh particles.

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.



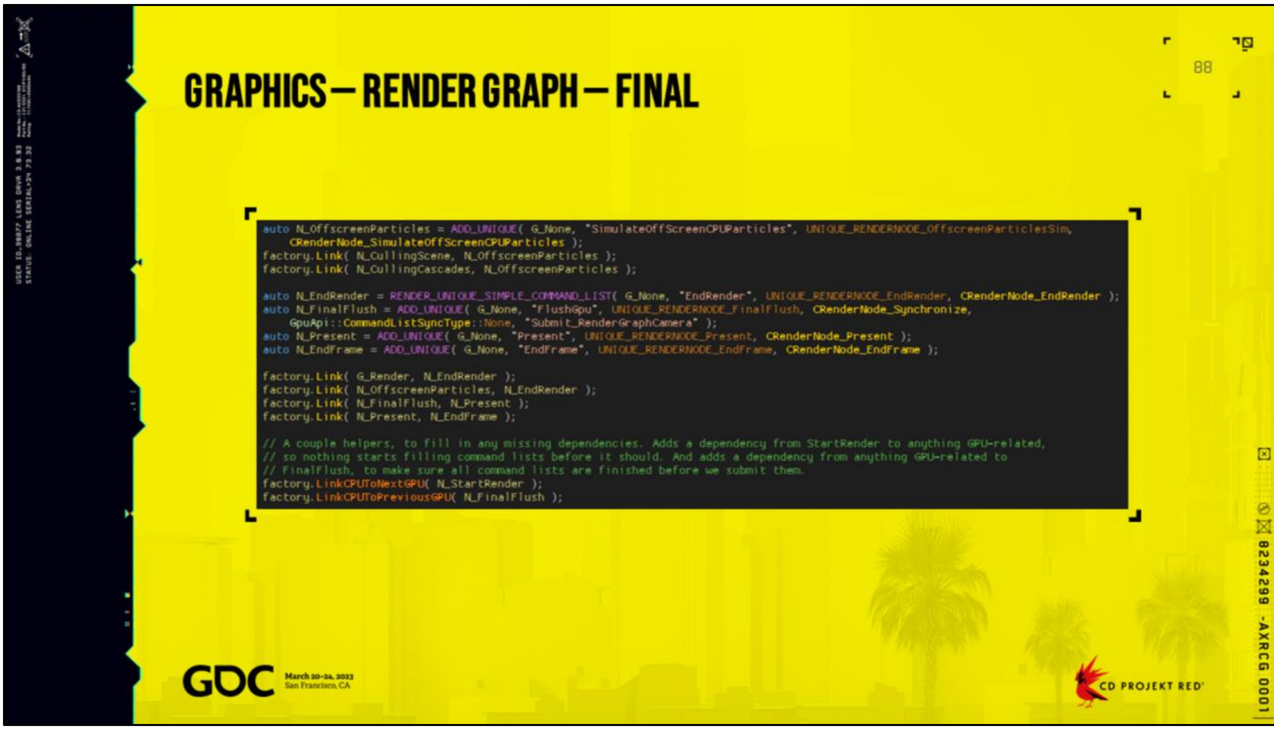
Lighting, PostFX

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.



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.



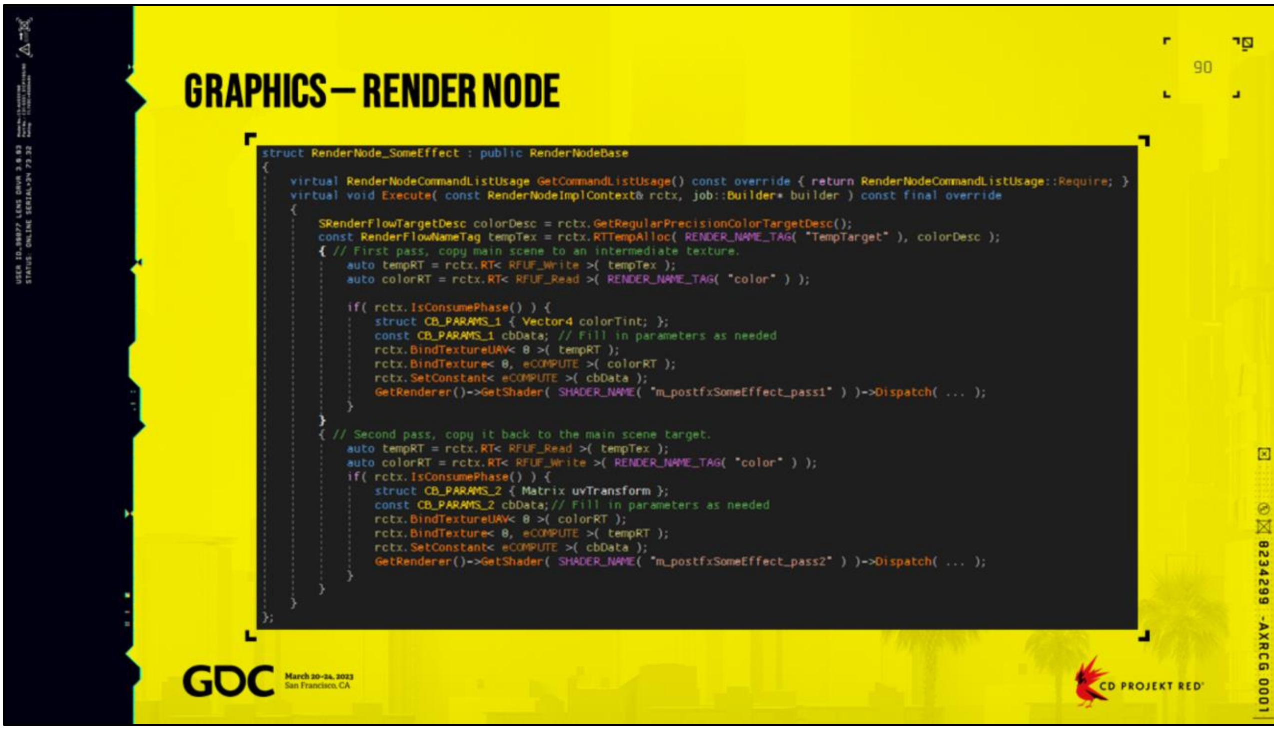
Main rendering is done, so we just have some assorted finalization.

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



First example, some sort of post process

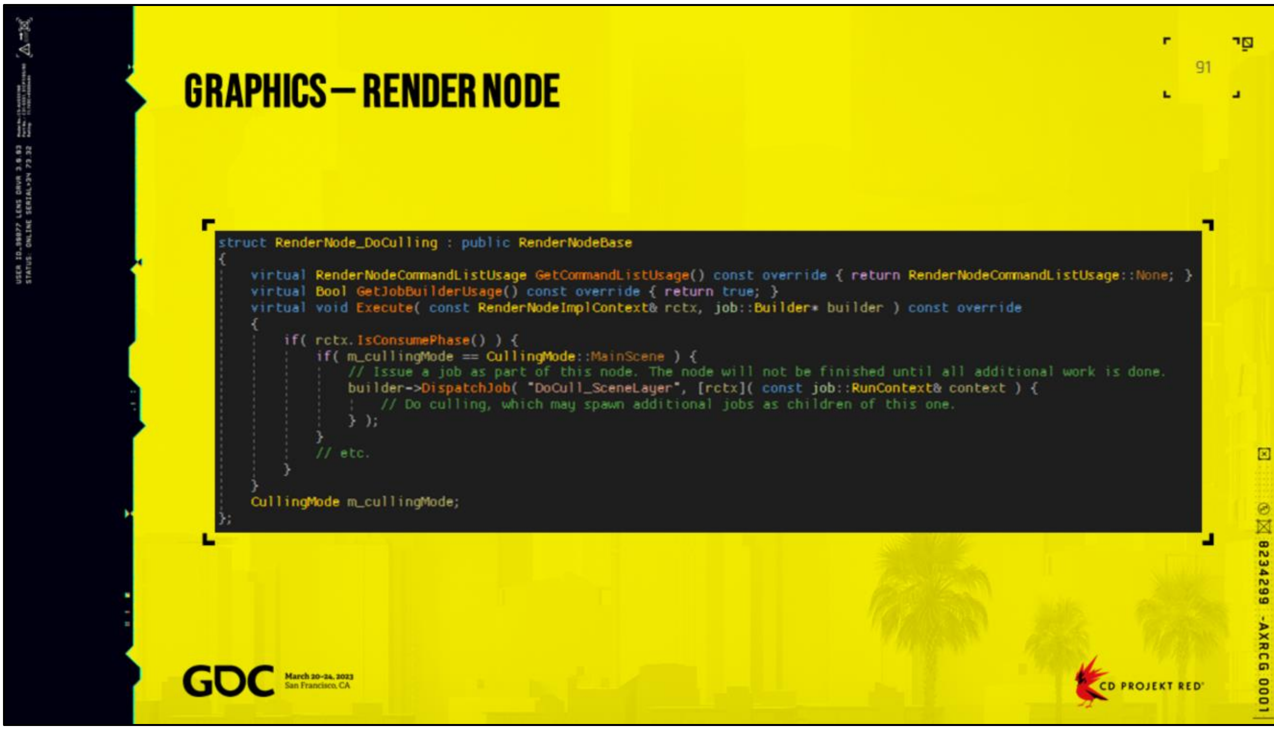
This is using a temporary texture to hold some intermediate results.

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 descs), 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.

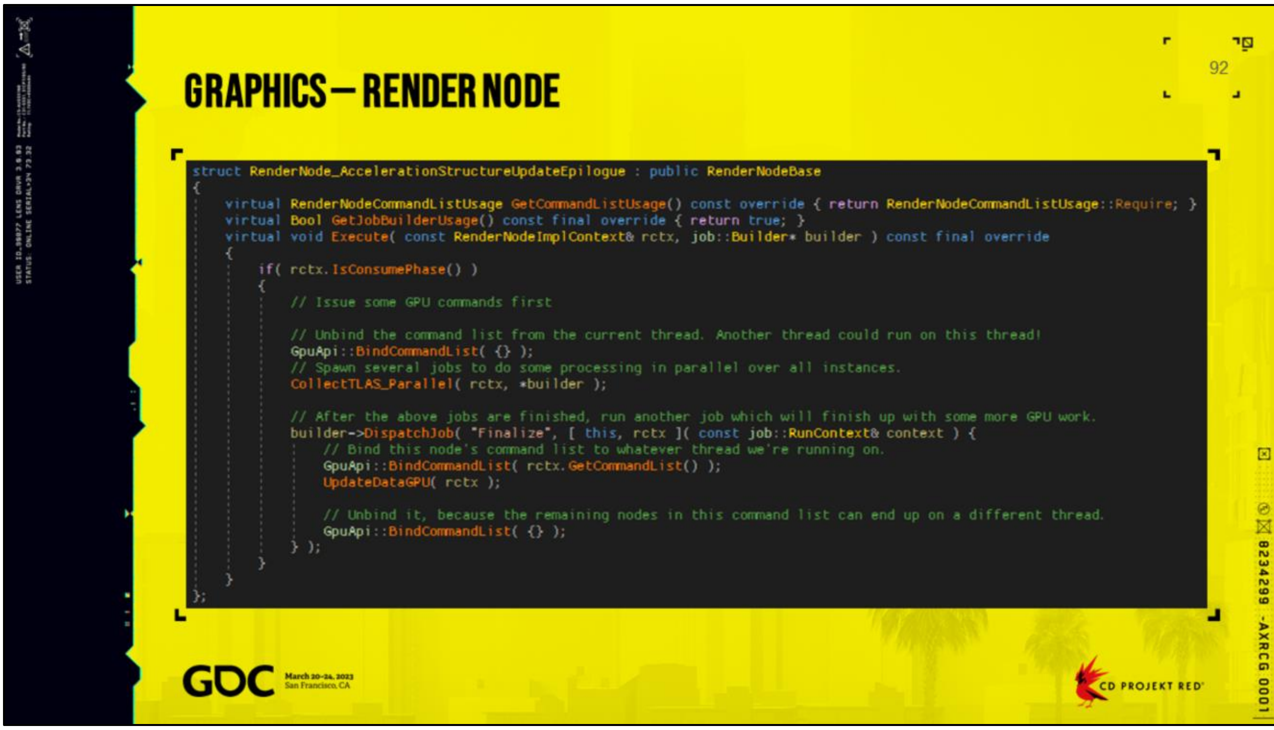
The node declares that it requires a Command List, which allows it to issue GPU commands during the Consume phase.



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.



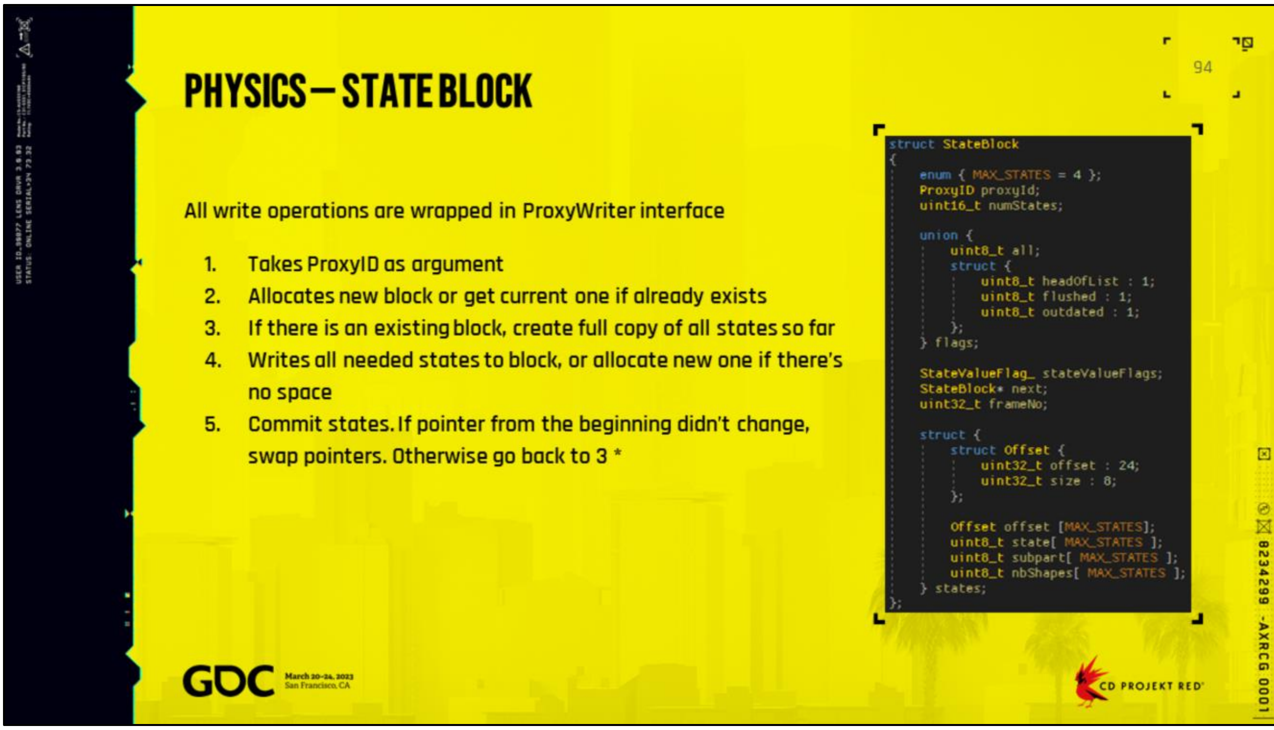
Finally we have a GPU node, which also can spawn additional parallel work.

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.

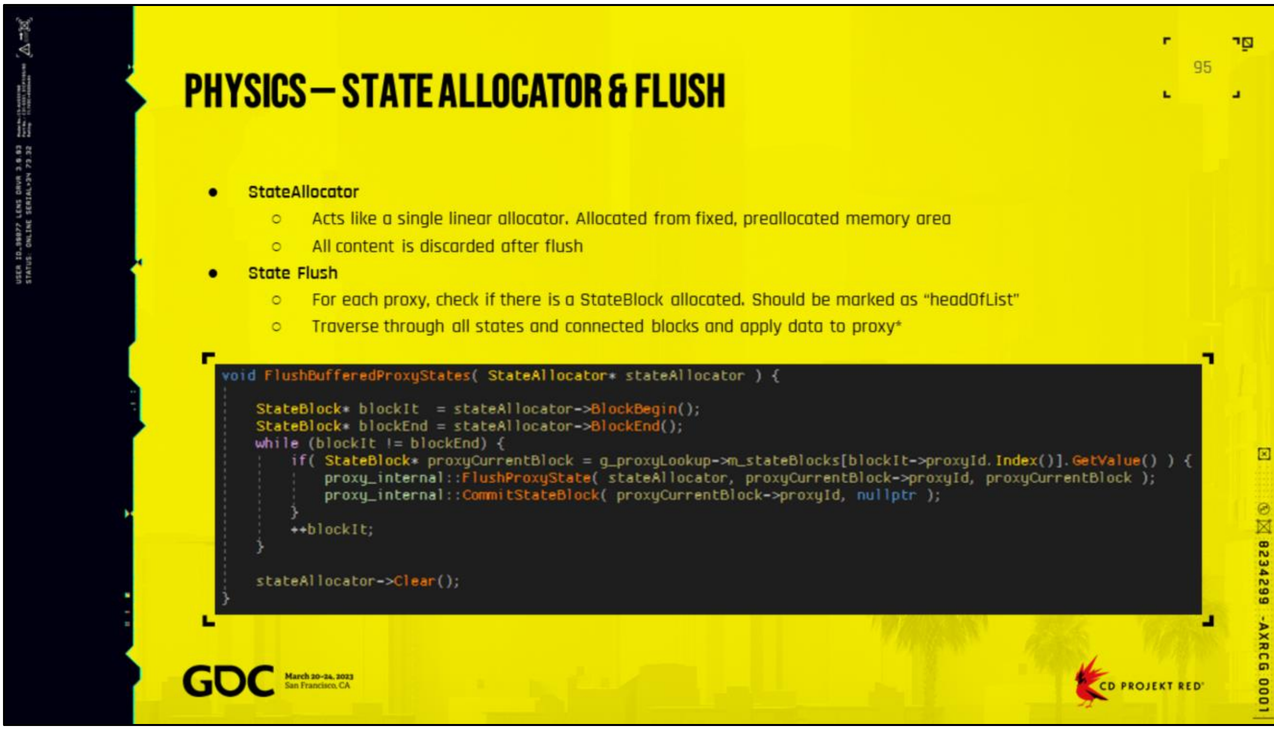
PHYSICS – STATE BUFFERING

- **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



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.



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