POPULATING THE WORLD WITH AN ACOUSTIC GRAPH IN CYBERPUNK 2077

MAREK BIELAWSKI

- 5 years at CD PROJEKT RED
- Sound system profiling and optimization
- Acoustics graph

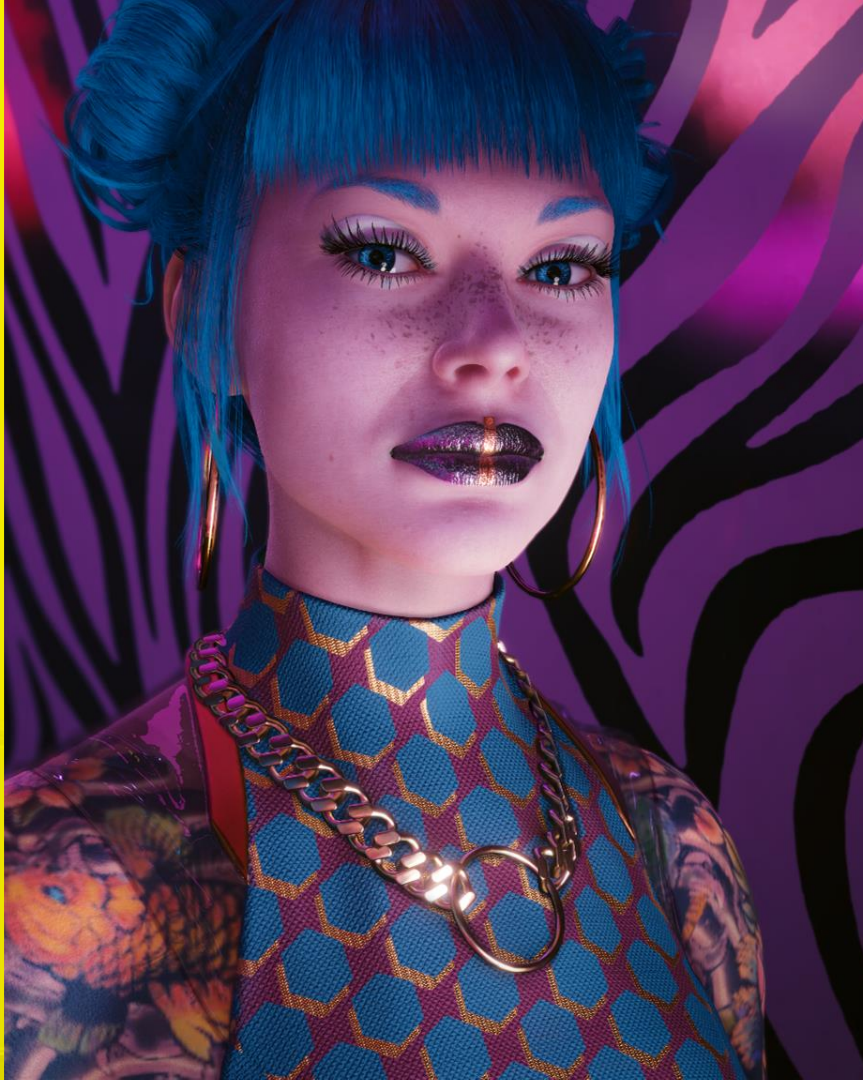
CD PROJEKT RED

CREDITS

- Colin Walder – lead (Colin's talk at GDC: Friday 13:30)
- Marek Bielawski - audio code - acoustics
- Giuseppe Marano - audio code
- Mateusz Ptasiński - audio code
- Daniel Murray - audio code expert
- Engine code (Charles's talk at GDC: tomorrow 11:30)

AGENDA

1. Motivations behind the acoustics graph Why?
2. The journey and the juicy stuff What?
3. Can those gang members hear me? Oh Really?
4. Acoustics from the depths What Else?
5. Go and build acoustics So what?



MOTIVATIONS BEHIND THE ACOUSTICS GRAPH

HOW DID IT ALL BEGIN?

- Witcher 3 tech: reverbs, raycast occlusion
- Next game needs to push us forward
- CP is vertical and dense – Not enough^tm
- Checking out some different tech, papers, presentations

OBSTRUCTION

- Raycast from the listener
- Muffle / lowpass the sound

OCCLUSION

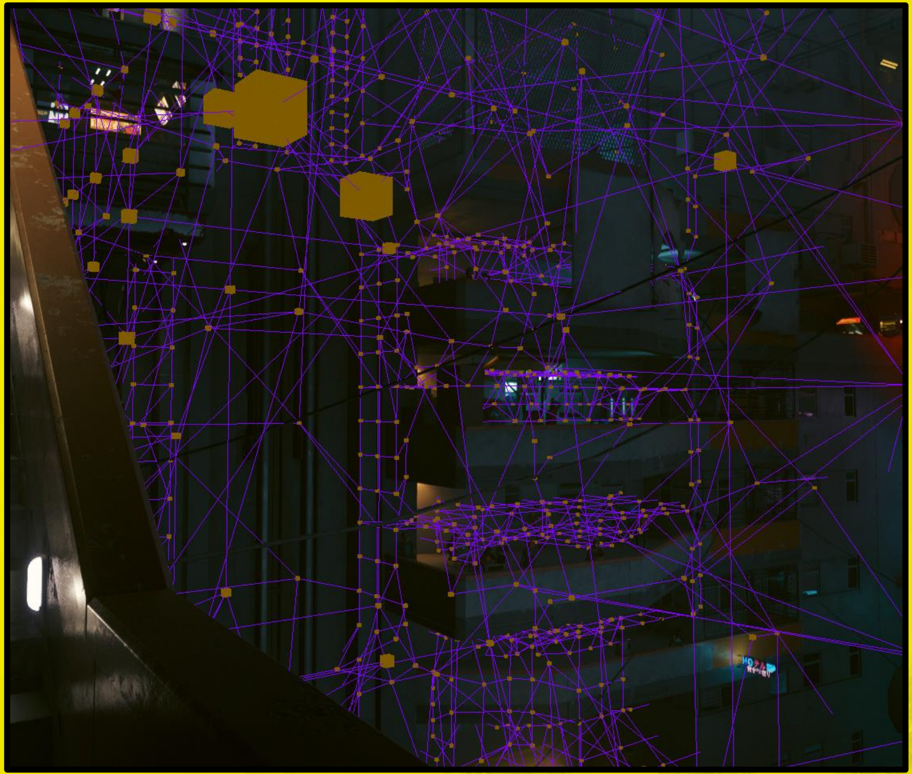
- Tells us about the room presence
- Pathfinding
- Affects the reverb

RESULT SOUND

- Exposing a minimum value of both
- Sound designer chooses how they affect the sound

CAN WE USE NAVMESH FOR PATHFINDING?

- Glued to the floor
- CP is a layered vertical game: balconies, staircases
- Not enough[tm]



FAST FORWARD TO SOME RESULTS

THE JOURNEY AND THE JUICY STUFF

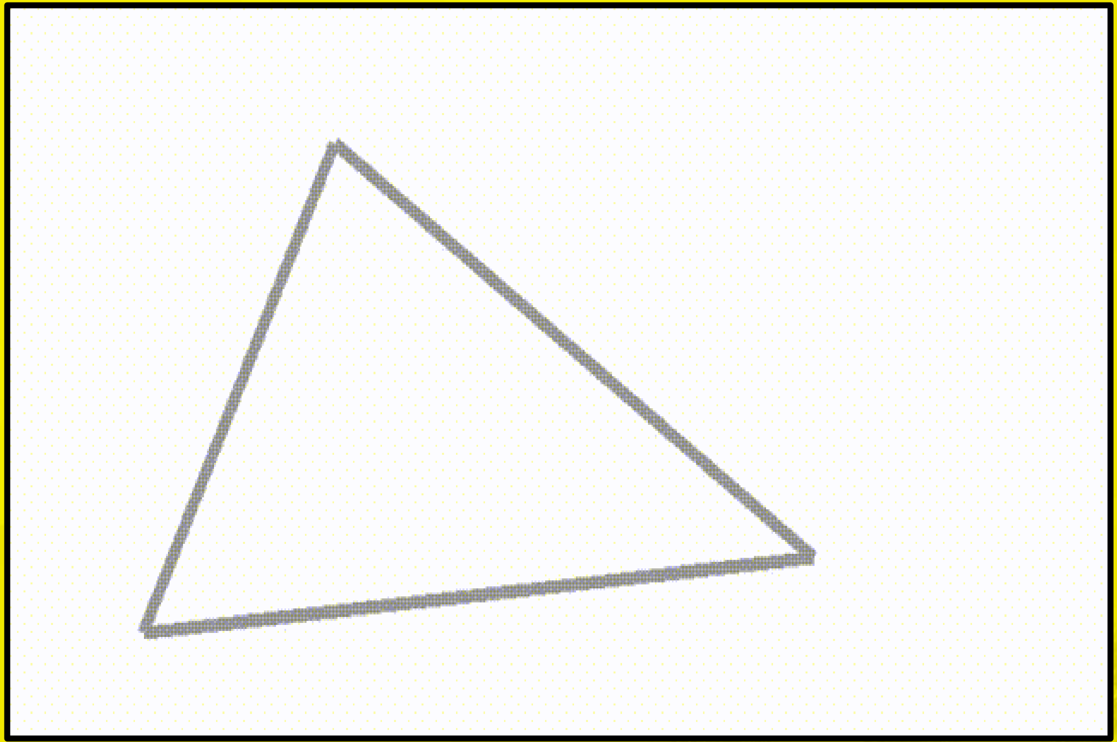
WHAT'S GOING ON

- Generating the graph – offline voxelization, node placement, serializing
- Runtime – streaming
- Pathfinding – algorithm, influence on occlusion

GENERATING THE GRAPH — VOXELIZE

- Iterate over each meaningful mesh
- Get the bounding box of a mesh to build an output buffer
- Iterate over each triangle

GENERATING THE GRAPH — VOXELIZE



GENERATING THE GRAPH – INTCOORDS

- 1 unit = 0.5m
- Moving from discrete space to float vector back and forth

```
struct IntCoords
{
    explicit IntCoords( const math::Vector3& floatPos );
    math::Vector3 ToVector3() const;
    math::Vector4 AsPoint() const;
    math::Vector4 AsDirection() const;

    union
    {
        struct
        {
            Int32 m_x;
            Int32 m_y;
            Int32 m_z;
        };
        Int32 m_array[ 3 ];
    };
};
```

```
RED_INLINE math::Vector3 IntCoords::ToVector3() const
{
    return Vector3(
        0.5f * static_cast< Float >( m_x ),
        0.5f * static_cast< Float >( m_y ),
        0.5f * static_cast< Float >( m_z )
    );
}
```

```
RED_INLINE IntCoords::IntCoords( const math::Vector3& floatPos )
    : m_x( static_cast< Int32 >( math::Floor( 2.0f * floatPos.X ) ) )
    , m_y( static_cast< Int32 >( math::Floor( 2.0f * floatPos.Y ) ) )
    , m_z( static_cast< Int32 >( math::Floor( 2.0f * floatPos.Z ) ) )
{
}
```

GENERATING THE GRAPH — VOXELIZE

- One patch is a 32 meter-wide cube
- 64 by 64 by 64 bits = 4K uint64 = 32KB
- Using full uint64 to represent a row

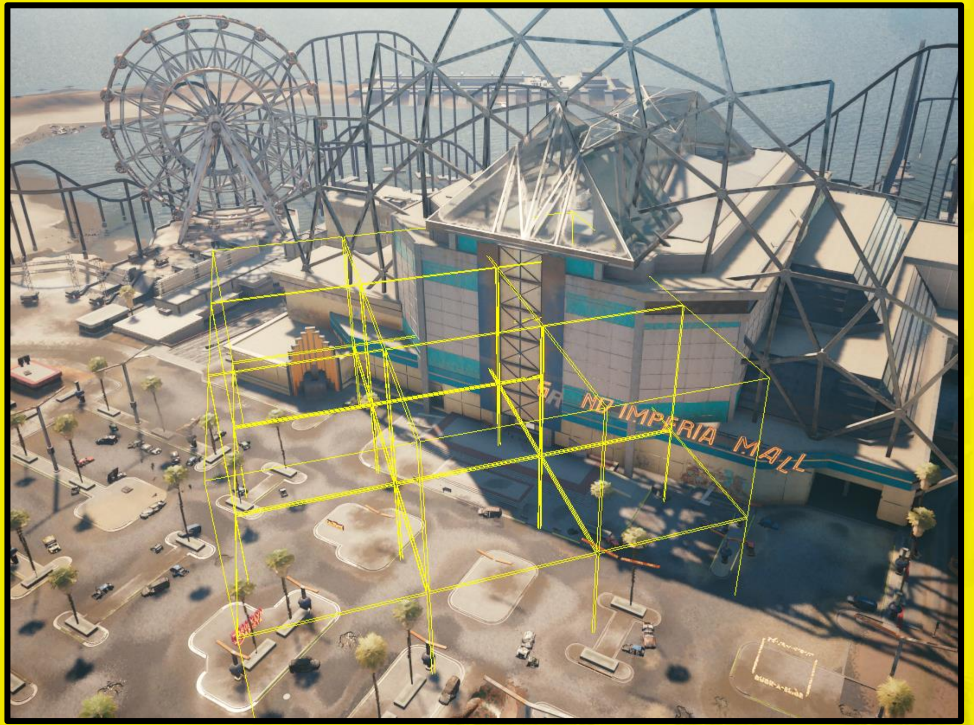
```
class VoxelizationPatch64
{
/// ...
    void Set( const UintCoords& coords );
    Bool Get( const UintCoords& coords ) const;

private:
    Uint64 m_data[ 4096 ];
};
```

```
void VoxelizationPatch64::Set( const UintCoords& coords )
{
    m_data[ ( coords.m_z << 6 ) + coords.m_y ] |= ( 1ull << coords.m_x );
}
```

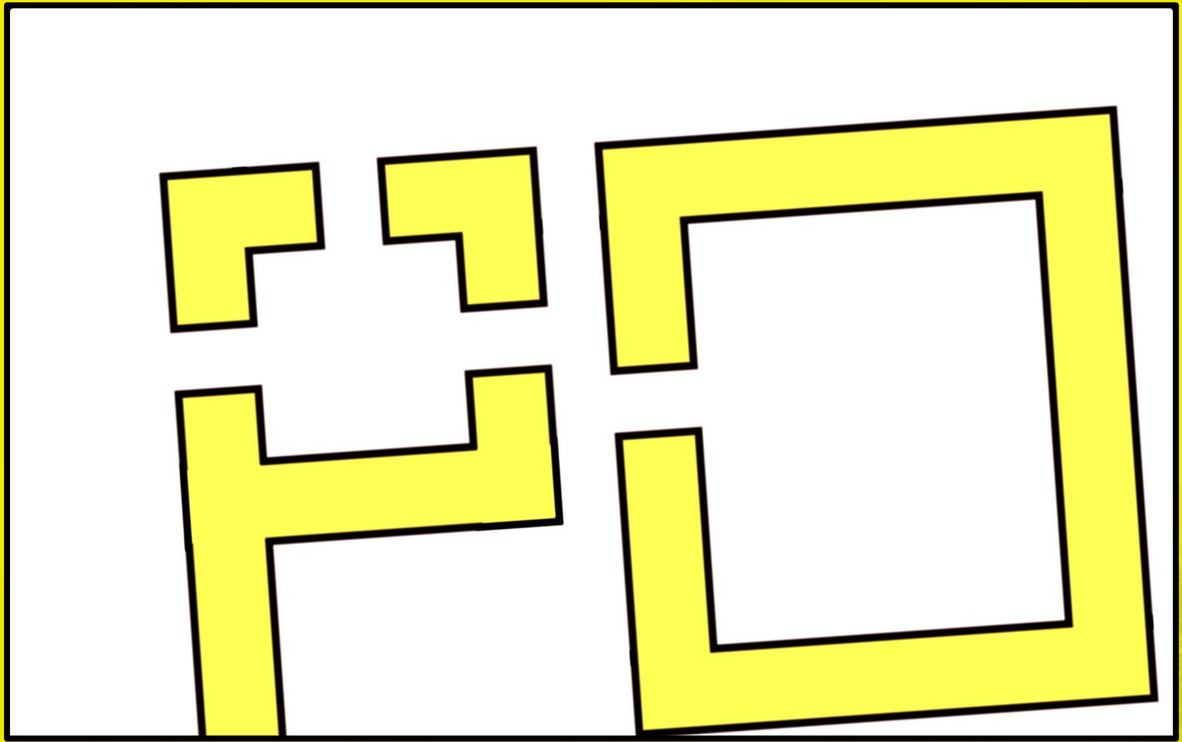
```
Bool VoxelizationPatch64::Get( const UintCoords& coords ) const
{
    Uint64 xRow = m_data[ ( coords.m_z << 6 ) + coords.m_y ];
    return ( ( xRow >> coords.m_x ) & 1 ) != 0;
}
```

GENERATING THE GRAPH — VOXELIZE

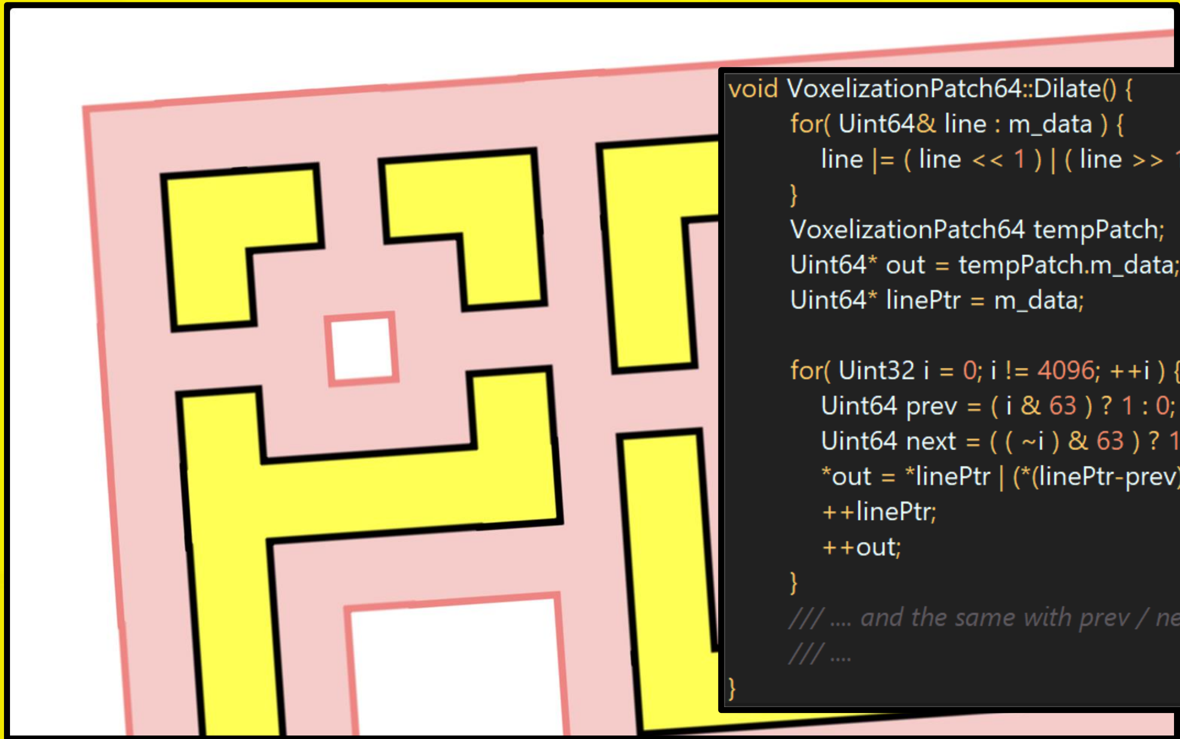
- One patch is a 32 meter-wide cube
- 64 by 64 by 64 bits = 4K uint64 = 32KB
- Using full uint64 to represent a row



GENERATING THE GRAPH — PLACE NODES

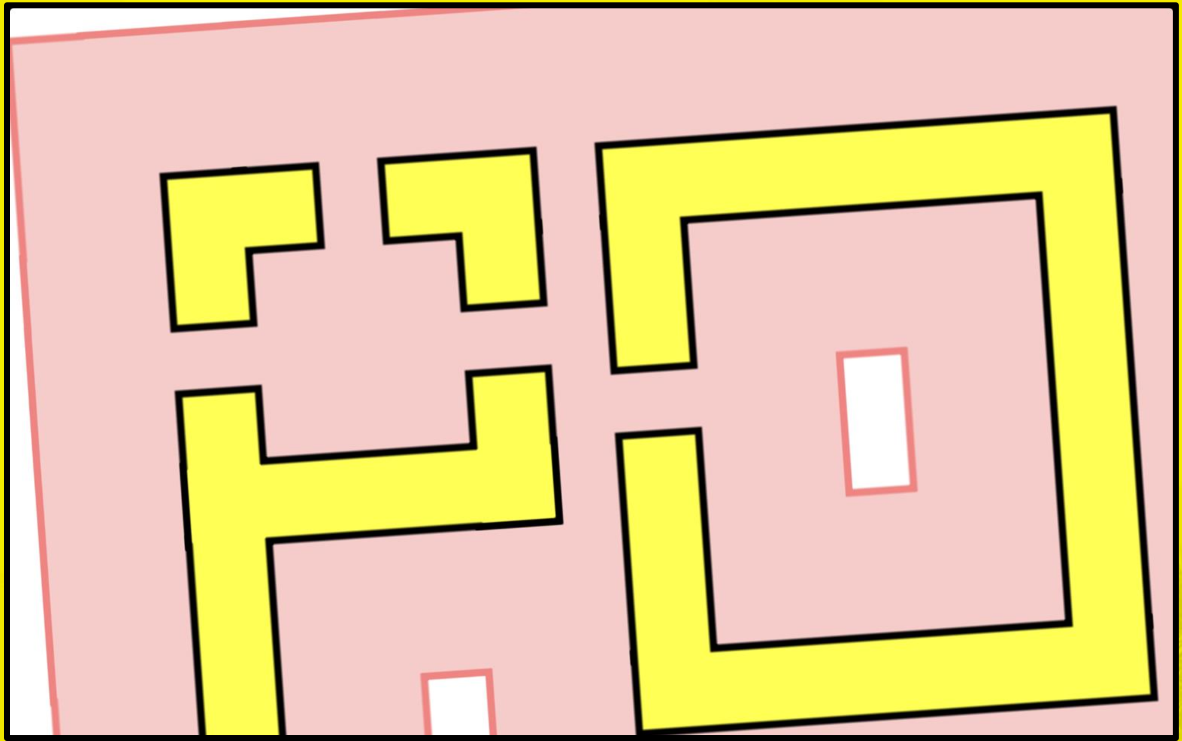


GENERATING THE GRAPH — PLACE NODES

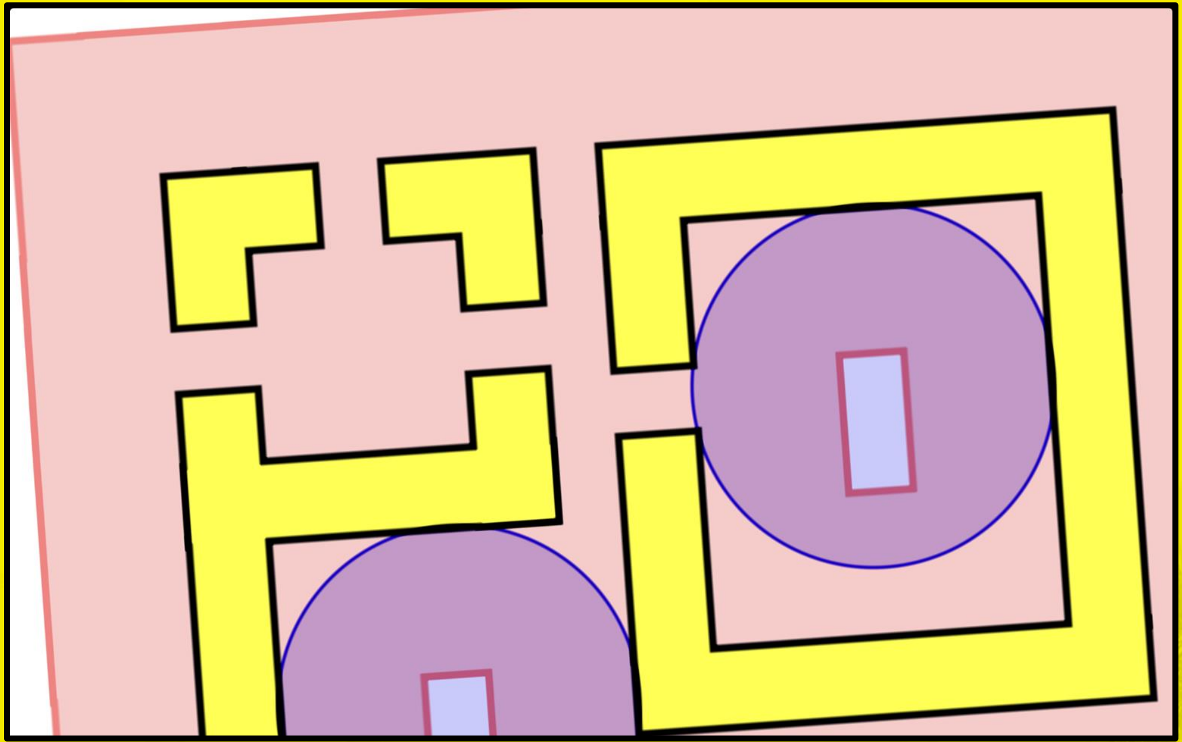


```
void VoxelizationPatch64::Dilate() {
    for( Uint64& line : m_data ) {
        line |= ( line << 1 ) | ( line >> 1 );
    }
    VoxelizationPatch64 tempPatch;
    Uint64* out = tempPatch.m_data;
    Uint64* linePtr = m_data;

    for( Uint32 i = 0; i != 4096; ++i ) {
        Uint64 prev = ( i & 63 ) ? 1 : 0;
        Uint64 next = ( ( ~i ) & 63 ) ? 1 : 0;
        *out = *linePtr | (*(linePtr-prev)) | (*linePtr+next);
        ++linePtr;
        ++out;
    }
    /// .... and the same with prev / next = 64 == tempPatch -> m_data
    /// ....
}
```

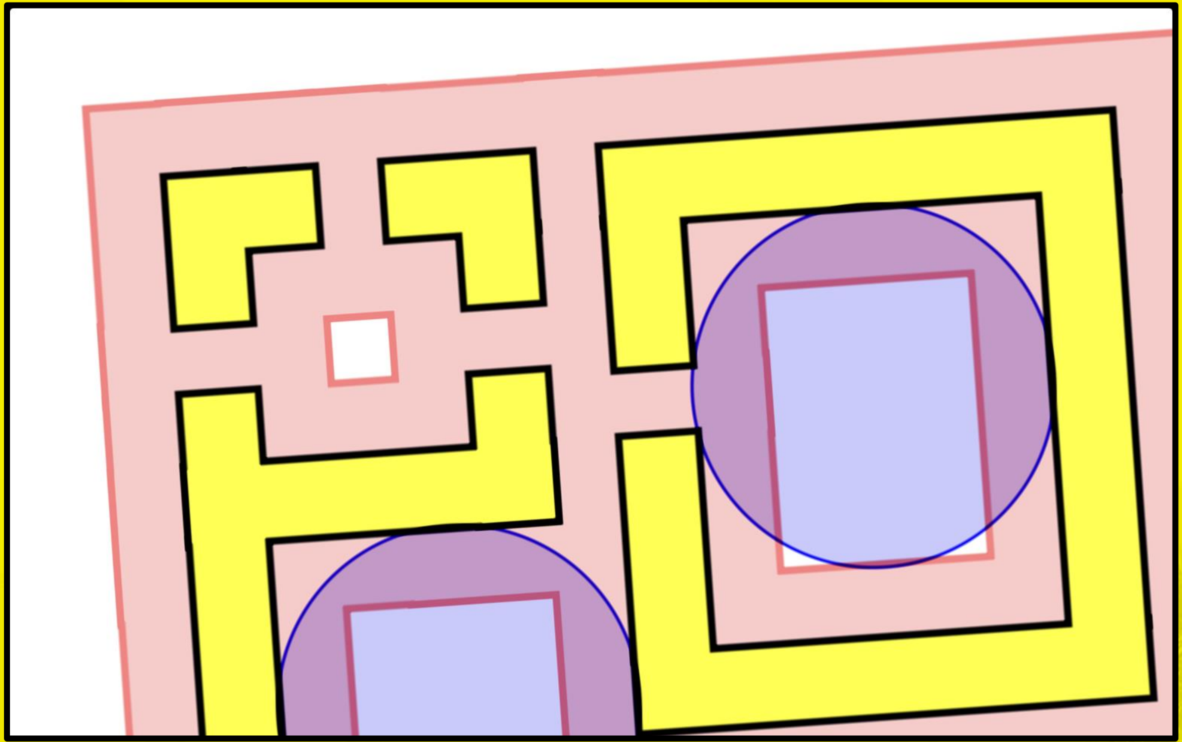
GENERATING THE GRAPH — PLACE NODES



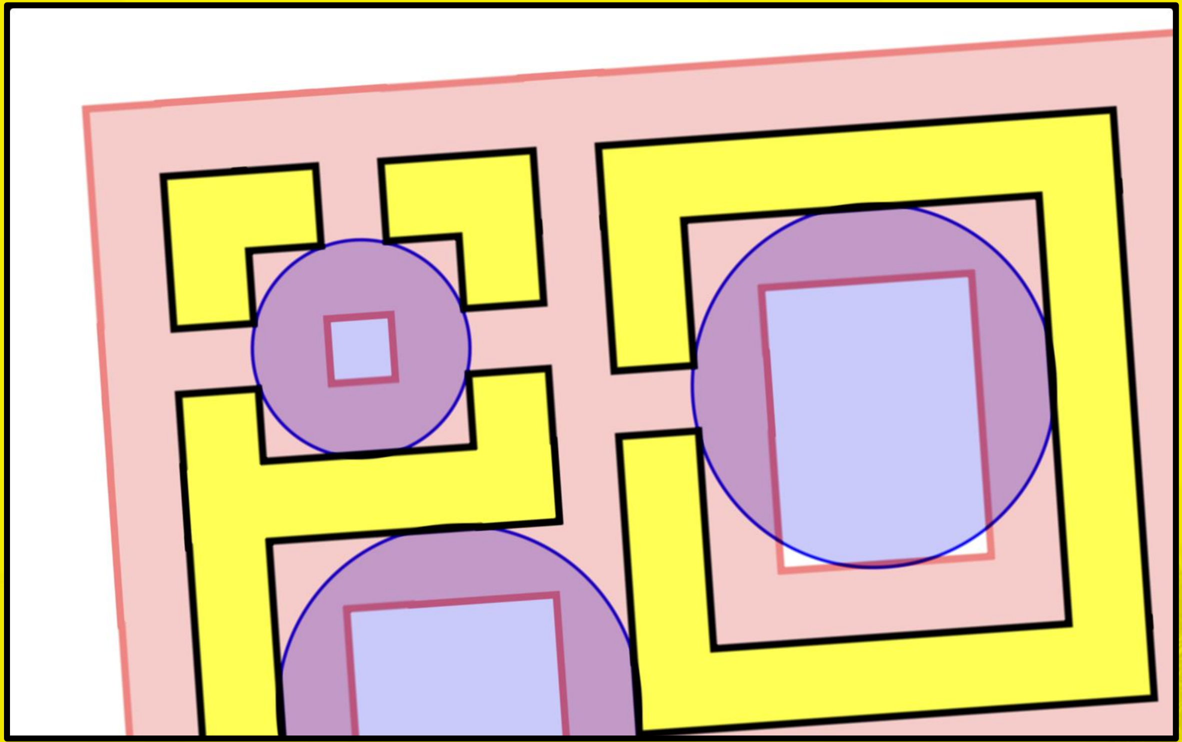
GENERATING THE GRAPH — PLACE NODES



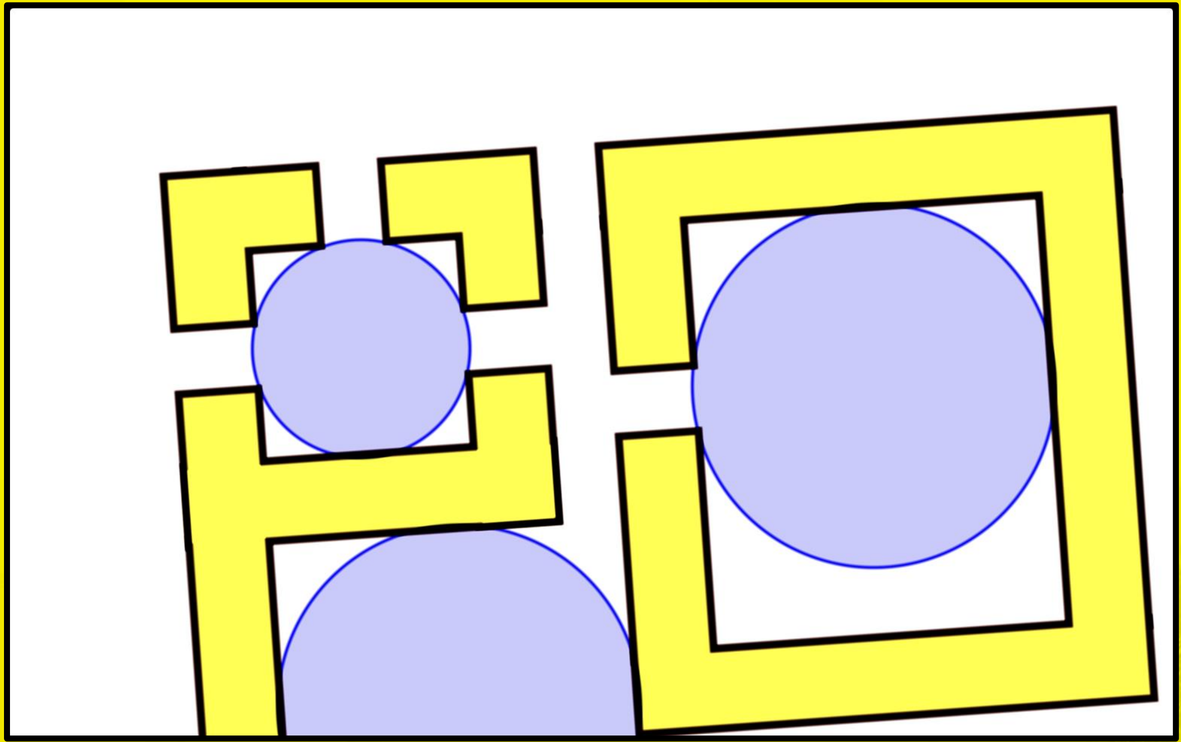
GENERATING THE GRAPH – PLACE NODES



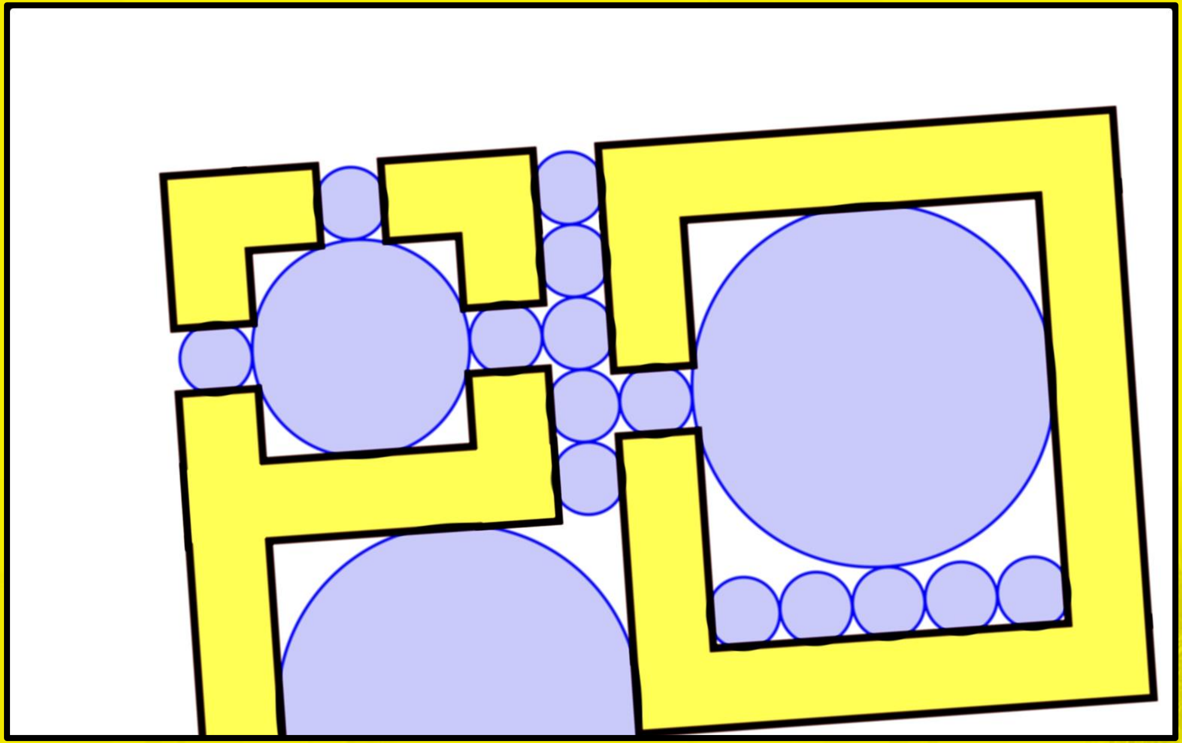
GENERATING THE GRAPH — PLACE NODES



GENERATING THE GRAPH — PLACE NODES

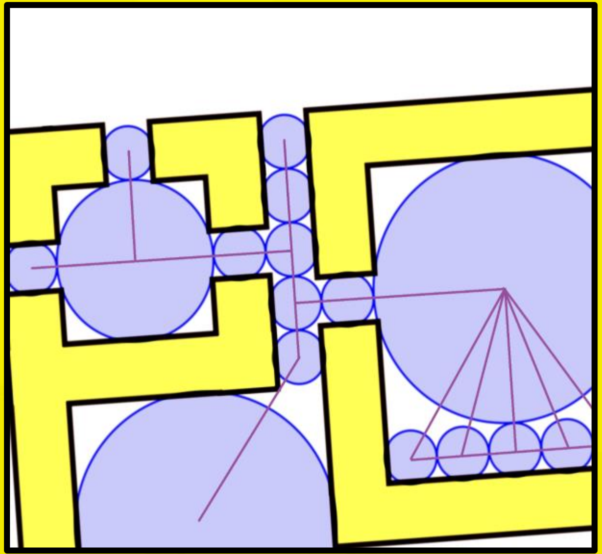


GENERATING THE GRAPH — PLACE NODES

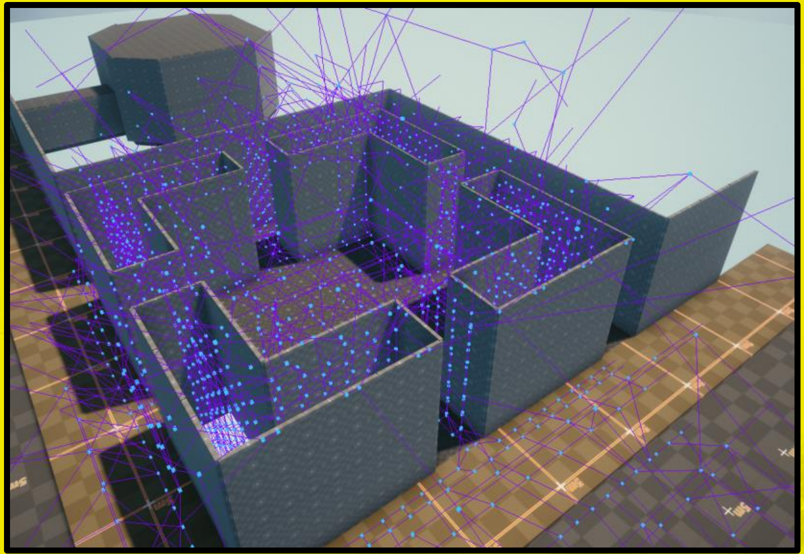


GENERATING THE GRAPH — PLACE NODES

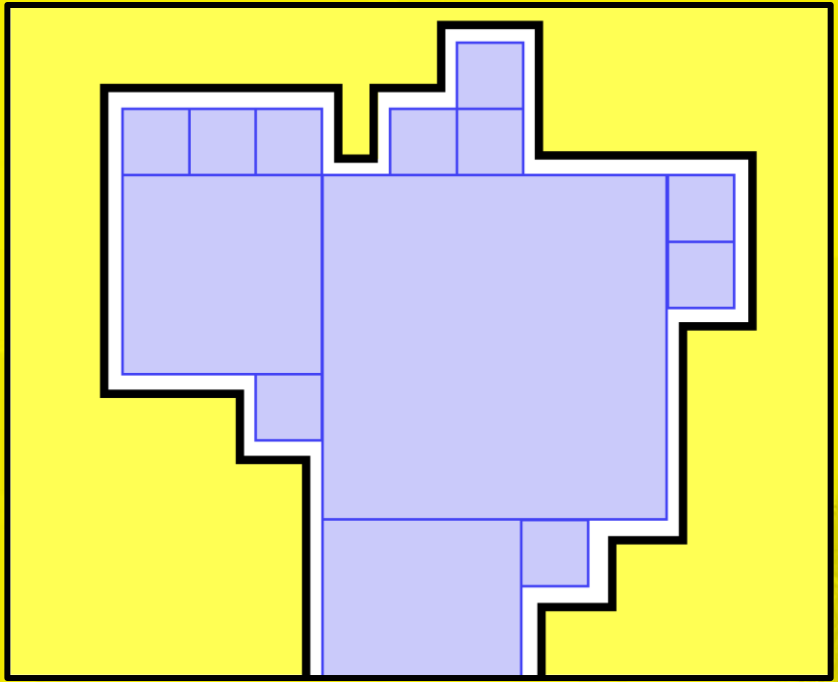
DESIRED RESULT



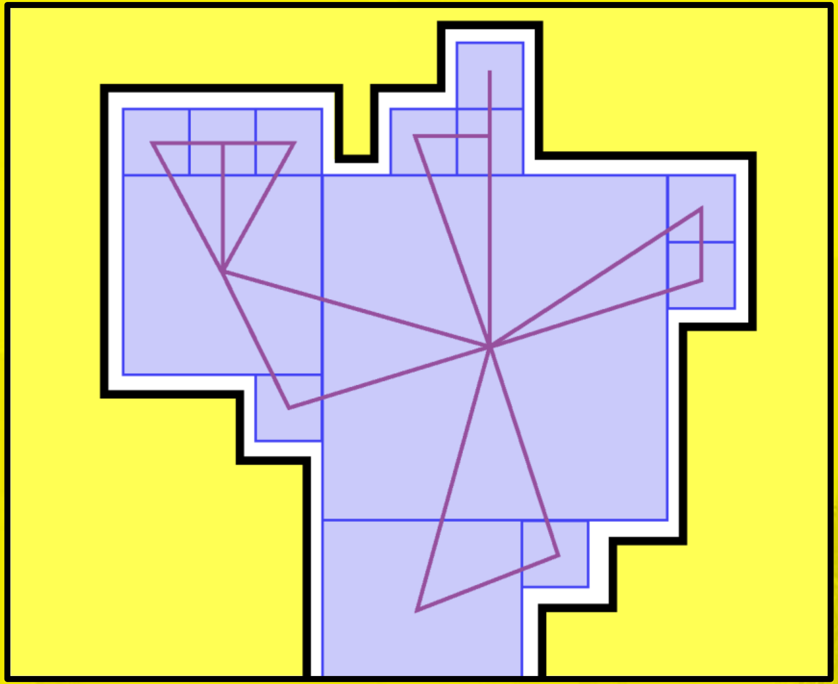
ACHIEVED RESULT



GENERATING THE GRAPH — DETERMINE CONNECTIONS

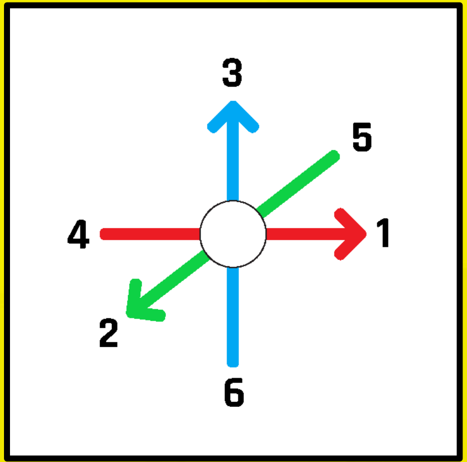


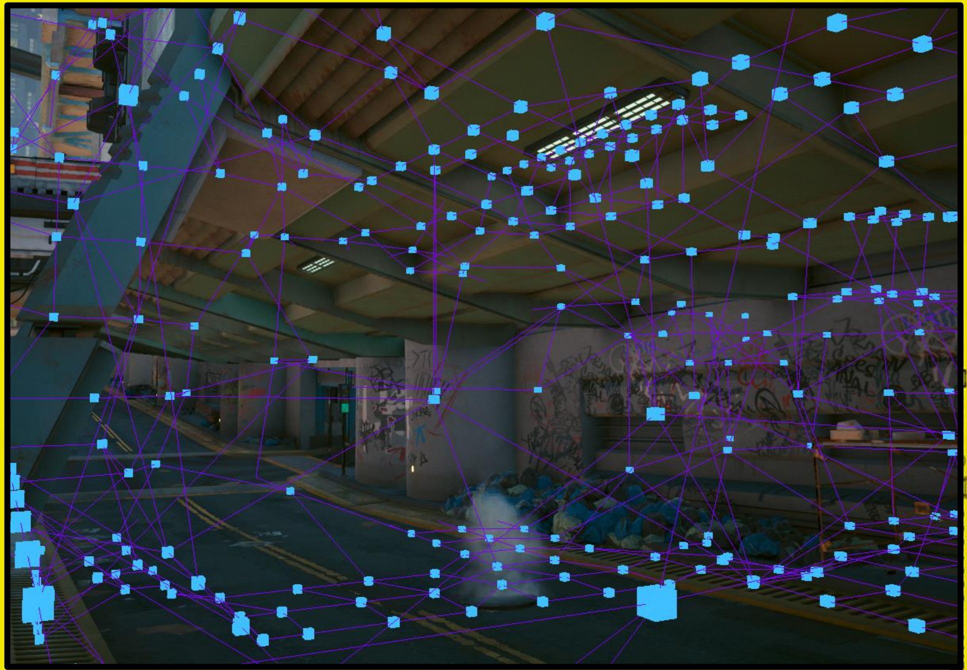
GENERATING THE GRAPH — DETERMINE CONNECTIONS



GENERATING THE GRAPH — DETERMINE CONNECTIONS

- max 6 neighbors
- pick the best node for each of 6 directions
- 2-way connections → we find only 3 directions and the negative ones will come automatically

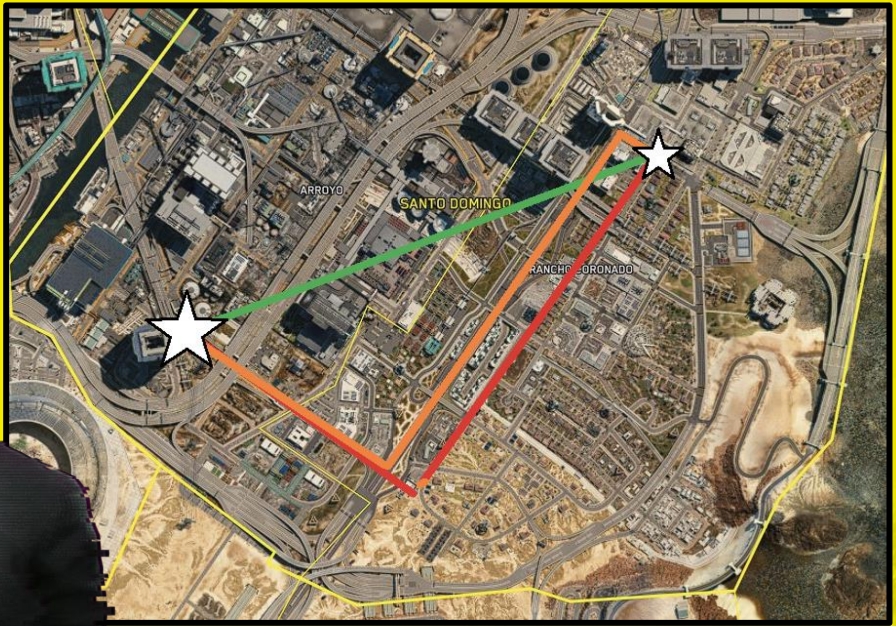




GENERATING THE GRAPH — DETERMINE CONNECTIONS

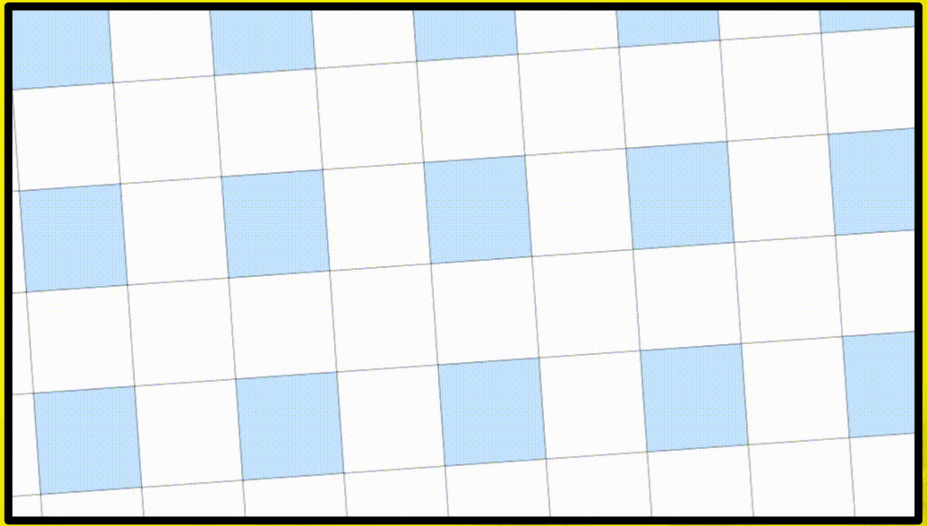
- The smaller the taxi distance is the better the connection is
  dist = |dx| + |dy| + |dz|
- Taxi distance favors axis aligned connections

- Connecting computations are heavy
- Heavy use of SSE



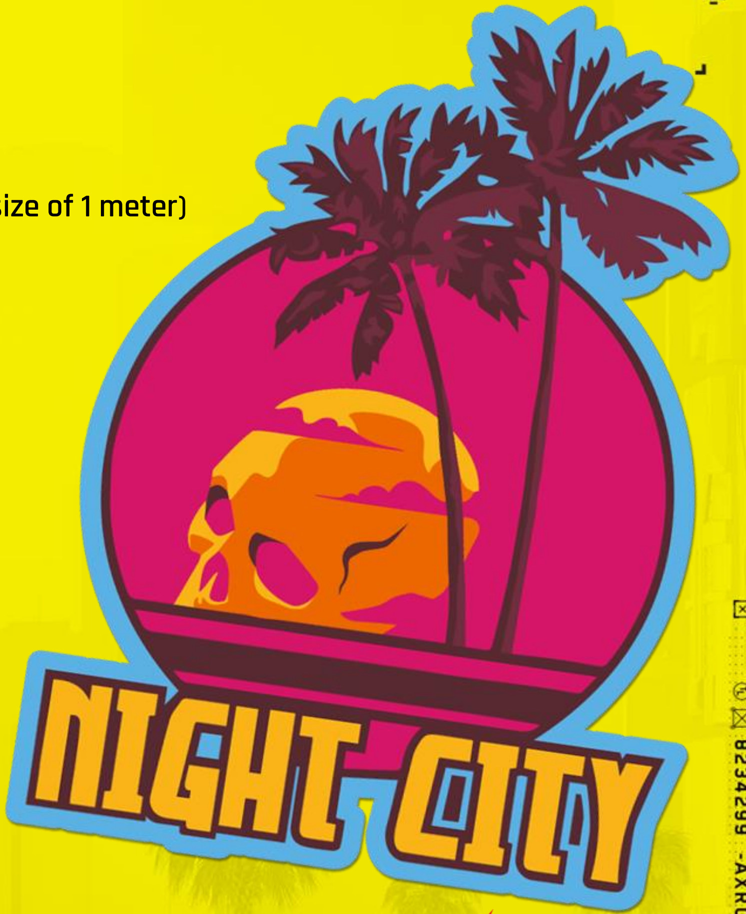
GENERATING THE GRAPH — DETERMINE CONNECTIONS

- Connection sectors in Parallel
- No neighboring sectors at the same time → otherwise:
  - Race condition (never crashed)
  - Indeterministic result
  - Broken incremental build



GENERATING THE GRAPH – PACKING

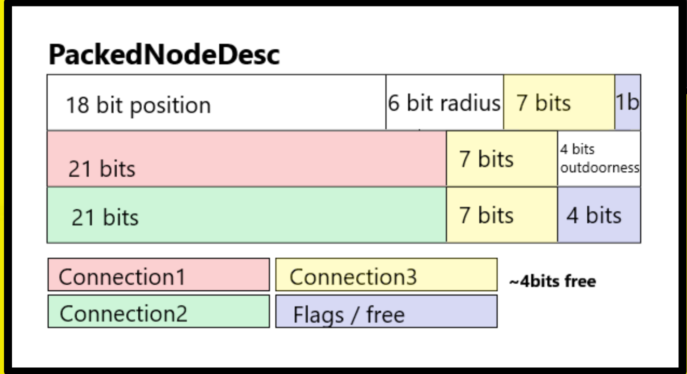
- Overall size was 8GB when I first measured it (with the grid size of 1 meter)
- The budget was ~1GB
- Streaming some large quantities of data



GENERATING THE GRAPH — PACK

- Using sectors aligned with 64x64x64 units
- 12 bytes per node

```cpp
class SOUND_API PackedNodeDescriptor
{
public:

    static const Uint32 c_roomFlag =   RED_FLAG( 28 ); // 0x10000000;
    static const Uint32 c_portalFlag = RED_FLAG( 29 ); // 0x20000000;
    static const Uint32 c_windowFlag = RED_FLAG( 30 ); // 0x40000000;
    static const Uint32 c_centerFlag = RED_FLAG( 31 ); // 0x80000000;


    Uint32 m_coordsAndRadius;
    Uint32 m_connectionCoords_1;
    Uint32 m_connectionCoords_2;
    IntCoords GetConnectionCoords( Uint32 index ) const;
    void SetConnectionCoords( Uint32 index, const IntCoords& coords );
    void ClearConnectionCoords();
```

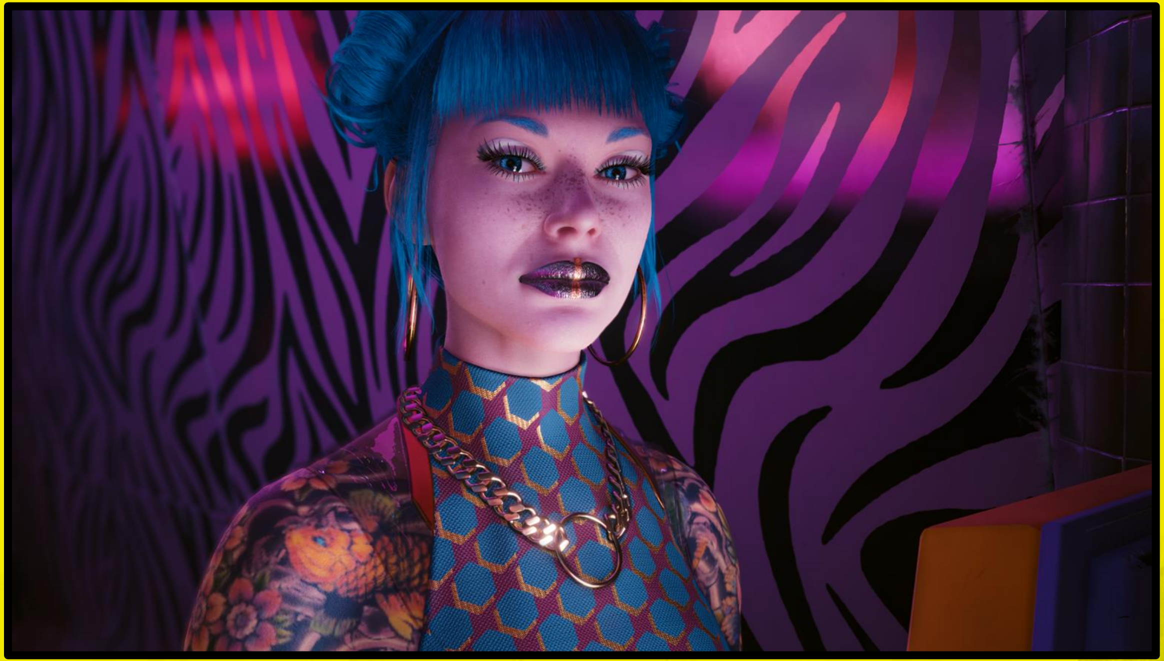


GENERATING THE GRAPH — PACK

```cpp
void SetConnectionCoords( Uint32 index, const IntCoords& coords, Uint32& outBuffer0, Uint32& outBuffer1, Uint32& outBuffer2 )
{
  if( index == 0 )
  {
    outBuffer1 &= ~0x1FFFFF;
    outBuffer1 |= static_cast< Uint32 >( coords.m_x ) & 127;
    outBuffer1 |= ( static_cast< Uint32 >( coords.m_y ) & 127 ) << 7;
    outBuffer1 |= ( static_cast< Uint32 >( coords.m_z ) & 127 ) << 14;
  }
  else if( index == 1 )
  {
    outBuffer2 &= ~0x1FFFFF;
    outBuffer2 |= static_cast< Uint32 >( coords.m_x ) & 127;
    outBuffer2 |= ( static_cast< Uint32 >( coords.m_y ) & 127 ) << 7;
    outBuffer2 |= ( static_cast< Uint32 >( coords.m_z ) & 127 ) << 14;
  }
  else if( index == 2 )
  {
    outBuffer0 &= ~0x7F000000;
    outBuffer1 &= ~0xFE00000;
    outBuffer2 &= ~0xFE00000;
    outBuffer0 |= ( static_cast< Uint32 >( coords.m_x ) & 127 ) << 24;
    outBuffer1 |= ( static_cast< Uint32 >( coords.m_y ) & 127 ) << 21;
    outBuffer2 |= ( static_cast< Uint32 >( coords.m_z ) & 127 ) << 21;
  }
}
```

- using raw buffers (the less rtti fluff the better)
- automatic cracken compression
- One file is 64 sectors (4x4x4)

GENERATING THE GRAPH — PACK



- using raw buffers (the less rtti fluff the better)
- automatic cracken compression
- One file is 64 sectors (4x4x4)
- 814 MB night city

GENERATING THE GRAPH – SUMMARY

- Voxelize meshes, add them up to sectors
- Iterate over voxelized and dilated buffers, with the most dilated first
- Limit Connections to 6 per node, use taxi distance to favor the most aligned ones
- Compress

RUNTIME — STREAMING BEHAVIOR

- Same size as the voxel patch (32 meters)
- Usually we are interested in fewer than 27 sectors (center of focus + neighbors)
- Sector holds a vector of nodes

RUNTIME — NODE

36 bytes per runtime node

- Old pending connection data for reconnecting
- Global world position
- Adding some runtime state like auto room generation

```
class SOUND_API Node : red::NonCopyable
{

    Uint32 m_connectionSetId = 0;
    Uint32 m_pendingConnectionData_0; // could be reduced to 8 bytes instead of 12
    Uint32 m_pendingConnectionData_1;
    Uint32 m_pendingConnectionData_2;
    IntCoords m_position;
    Uint16 m_roomId = 0;
    Uint16 m_tagId;
    Uint8 m_radius;
    Uint8 m_outdoornessAndZoneSpreading;
    Uint8 m_flags;
    Uint8 m_zoneSpreadingCache;

};
```

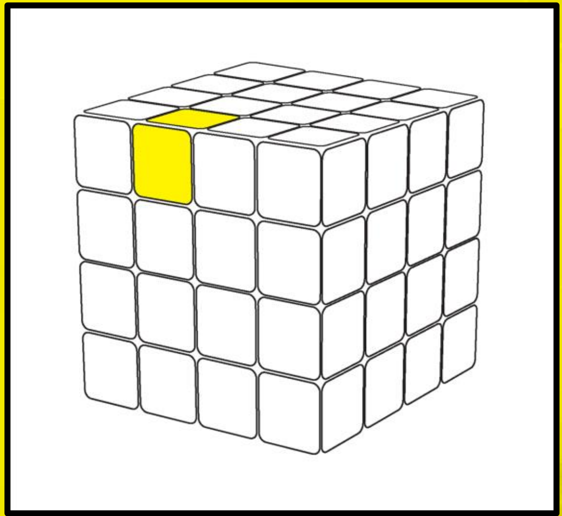
RUNTIME — GLOBAL NODE ID

- 1 byte + 2 bytes
  – still less than 8 byte ptr
- Small sector id 0-216 /
  1 byte
- we never deal with sector
  apart (though there was a
  bug once)

	3	4	5	0	1	2	3
5x6	33	34	35	30	31	32	33
0x6	3	4	5	0	1	2	3
[illegible]	9	10	11	6	7 Player	8	9
2x6	15	16	17	12	13	14	15
3x6	21	22	23	18	19	20	21
4x6	27	28	29	24	25	26	27
5x6	33	34	35	30	31	31	33

RUNTIME – STREAMING BEHAVIOR

- The sector is actually referring to a larger asset that covers 4 by 4 by 4 sectors
- Optimal approach on the hdd drives
- Loading only the packed representation
- Largest asset is ~2MB, while the mediate is about 120KB



RUNTIME – STREAMING BEHAVIOR

- Unpacking nodes – fast
- filling octree – slow
- connecting nodes – slow
- 0.5ms budget in a frame for filling octree and the same for connecting nodes
- using stop watch
- unpacking can take several frames

```
Bool SectorStreamEntry::Tick( Float dt, const IntCoords& focusPosition )
{
    if( m_dataPending )
    {
        if( m_data->PutDataToSoundSystem( m_worldPosition, m_internalCoords, 0, m_data ) )
        {
            GSoundSystem->GetAcousticsSystem()->ResetAreaFunctions();
            m_dataPending = false;
        }
    }
    else if( m_postLoadingStage == audio::acoustics::PostLoadingStage::FillingOctree )
    {
        GSoundSystem->GetAcousticsSystem()->FillOctree( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
    }
    else if( m_postLoadingStage != audio::acoustics::PostLoadingStage::Done )
    {
        GSoundSystem->GetAcousticsSystem()->ConnectNodesBySector( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
        if( m_postLoadingStage == audio::acoustics::PostLoadingStage::Done )
        {
            m_data->ReleaseDeferredBuffer( m_internalCoords );
            GSoundSystem->GetAcousticsSystem()->MarkAsFullyConnected( m_smallSectorId );
        }
    }
    return true;
}
```

```
enum class PostLoadingStage : Uint8
{
    FillingOctree,
    ConnectingCenter,
    ConnectingLeft,
    ConnectingBack,
    ConnectingDown,
    Done
};
```

RUNTIME — STREAMING BEHAVIOR

- Unpacking nodes – fast
- filling octree – slow
- connecting nodes – slow
- 0.5ms budget in a frame for filling octree and the same for connecting nodes
- using stop watch
- unpacking can take several frames

```cpp
Bool SectorStreamEntry::Tick( Float dt, const IntCoords& focusPosition )
{
    if( m_dataPending )
    {
        if( m_data->PutDataToSoundSystem( m_worldPosition, m_internalCoords, 0, m_data ) )
        {
            GSoundSystem->GetAcousticsSystem()->ResetAreaFunctions();
            m_dataPending = false;
        }
    }
    else if( m_postLoadingStage == audio::acoustics::PostLoadingStage::FillingOctree )
    {
        GSoundSystem->GetAcousticsSystem()->FillOctree( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
    }
    else if( m_postLoadingStage != audio::acoustics::PostLoadingStage::Done )
    {
        GSoundSystem->GetAcousticsSystem()->ConnectNodesBySector( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
        if( m_postLoadingStage == audio::acoustics::PostLoadingStage::Done )
        {
            m_data->ReleaseDeferredBuffer( m_internalCoords );
            GSoundSystem->GetAcousticsSystem()->MarkAsFullyConnected( m_smallSectorId );
        }
    }
    return true;
```

```cpp
void Init( const PackedNodeDescriptor* desc, const IntCoords& minCoords )
{
    m_position = desc->GetWorldCoords( minCoords );
    m_radius = desc->GetIntRadius();
    m_outdoornessAndZoneSpreading = desc->GetIntOutdoorness();
    m_flags = ( ( desc->m_coordsAndRadius>>24 ) & c_flagIsCenter );
    m_flags |= ( ( desc->m_connectionCoords_2>>24 ) & c_flagsRoomPortalWindow );
    m_pendingConnectionData_0 = desc->m_coordsAndRadius;
    m_pendingConnectionData_1 = desc->m_connectionCoords_1;
    m_pendingConnectionData_2 = desc->m_connectionCoords_2;
}
```

RUNTIME – STREAMING BEHAVIOR

- Unpacking nodes – fast
- filling octree – slow
- connecting nodes – slow
- 0.5ms budget in a frame for filling octree and the same for connecting nodes
- using stop watch
- unpacking can take several frames

```cpp
Bool SectorStreamEntry::Tick( Float dt, const IntCoords& focusPosition )
{
    if( m_dataPending )
    {
        if( m_data->PutDataToSoundSystem( m_worldPosition, m_internalCoords, 0, m_data ) )
        {
            GSoundSystem->GetAcousticsSystem()->ResetAreaFunctions();
            m_dataPending = false;
        }
    }
    else if( m_postLoadingStage == audio::acoustics::PostLoadingStage::FillingOctree )
    {
        GSoundSystem->GetAcousticsSystem()->FillOctree( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
    }
    else if( m_postLoadingStage != audio::acoustics::PostLoadingStage::Done )
    {
        GSoundSystem->GetAcousticsSystem()->ConnectNodesBySector( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
        if( m_postLoadingStage == audio::acoustics::PostLoadingStage::Done )
        {
            m_data->ReleaseDeferredBuffer( m_internalCoords );
            GSoundSystem->GetAcousticsSystem()->MarkAsFullyConnected( m_smallSectorId );
        }
    }
    return true;
}
```

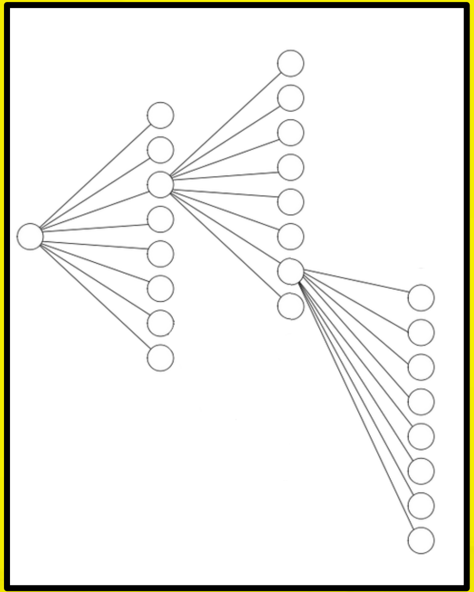
```cpp
red::StopWatch timer;
for( ; nodeCount != end; ++nodeCount )
{
    sector->AddToOctreeRoot( nodeCount );

    if( timer.GetDeltaF() > .0005f )
    {
        ++nodeCount;
        break;
    }
}
```

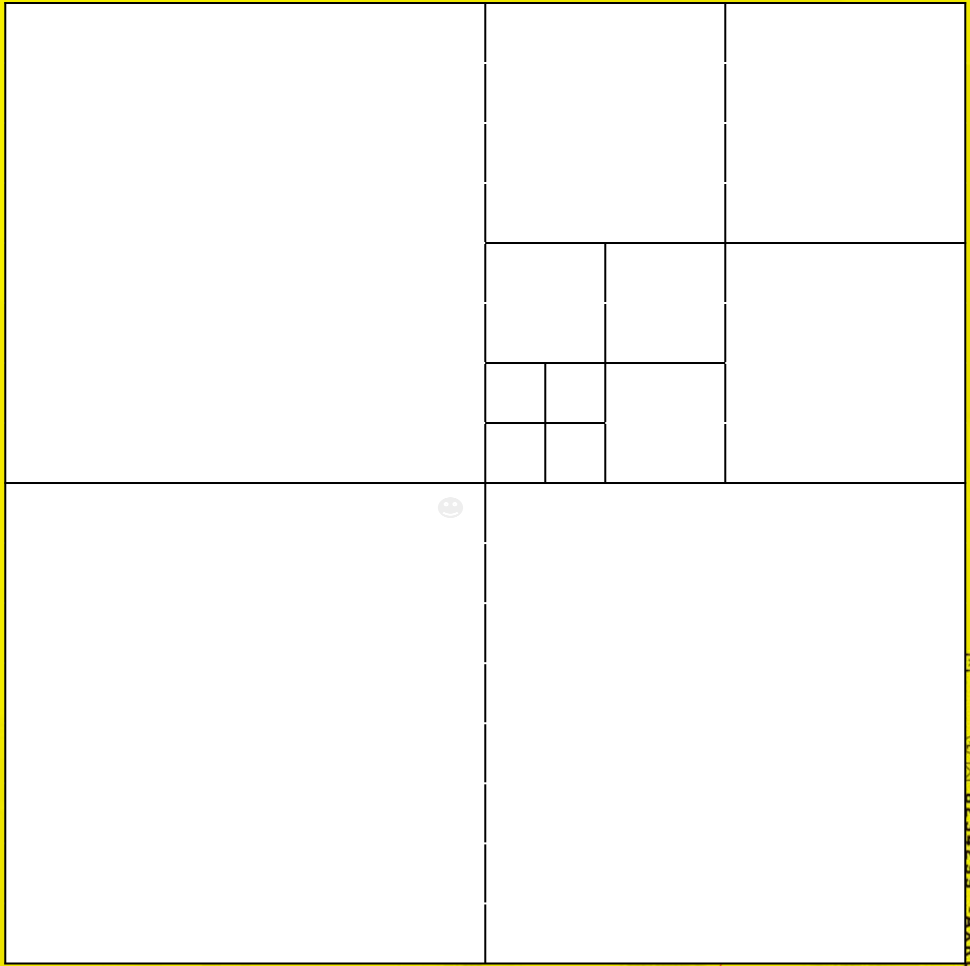
RUNTIME – STREAMING BEHAVIOR

- Unpacking nodes – fast
- filling octree – slow
- connecting nodes – slow
- 0.5ms budget in a frame for filling octree and the same for connecting nodes
- using stop watch
- unpacking can take several frames

```
Bool SectorStreamEntry::Tick( Float dt, const IntCoords& focusPosition )
{
    if( m_dataPending )
    {
        if( m_data->PutDataToSoundSystem( m_worldPosition, m_internalCoords, 0, m_data ) )
        {
            GSoundSystem->GetAcousticsSystem()->ResetAreaFunctions();
            m_dataPending = false;
        }
    }
    else if( m_postLoadingStage == audio::acoustics::PostLoadingStage::FillingOctree )
    {
        GSoundSystem->GetAcousticsSystem()->FillOctree( m_worldPosition, m_processedNodeCount, m_postLoadingStage );
    }
    else if( m_postLoadingStage != audio::acoustics::PostLoadingStage::Done )
    {
        GSoundSystem->GetAcousticsSystem()->ConnectNodesBySector( m_worldPosition, m_processedNodeCount, m_postLoadingStage )
        if( m_postLoadingStage == audio::acoustics::PostLoadingStage::Done )
        {
            m_data->ReleaseDeferredBuffer( m_internalCoords );
            GSoundSystem->GetAcousticsSystem()->MarkAsFullyConnected( m_smallSectorId );
        }
    }
    return true;
}
```

RUNTIME – OCTREE





RUNTIME – OCTREE

- Values of the octree leaf are indices of the nodes in the sector vector
- Using 16 bit numbers
- One node can cover multiple branches

- Oh Wait: We have over 64K octree nodes :)

```
struct OctreeLeaf
{
    Uint16 m_count;
    Uint16 m_values[ 7 ];
};

struct OctreeBranch
{
    Uint16 m_childIndices[ 8 ];
};

struct OctreeeNode
{
    union
    {
        OctreeBranch m_branch;
        OctreeLeaf m_leaf;
    };
    Bool m_isLeaf;
```

```
red::DynArray< OctreeeNode > m_octreeLookup = {PoolAudioAcousticNodeOctrees()};
```

RUNTIME – OCTREE

- Even more custom octree ;)
- 18 bit index -> 256K range

```
void SetBranchIndex( SectorGrid::OctreeeNode* node, Uint32 value, Uint8 i )
{
    node->m_branch.m_indexBuffer[ i ] = (Uint16)value & 0xffff;
    node->m_branch.m_higherValues &= ~( 3 << ( i << 1 ) );
    node->m_branch.m_higherValues |= ( value >> 16 ) << ( i << 1 );
}
Uint32 GetBranchIndex( const SectorGrid::OctreeeNode* node, Uint8 i )
{
    Uint32 retVal = node->m_branch.m_indexBuffer[ i ];
    Uint32 higherValues = ( ( Uint32 )node->m_branch.m_higherValues ) & 0xffff;
    retVal &= 0xffff;
    retVal |= ( ( higherValues >> ( i << 1 ) ) & 3 ) << 16;
    return retVal;
}
```

```
struct OctreeLeaf
{
    Uint16 m_count;
    Uint16 m_values[ 8 ];
};

struct OctreeBranch
{
    Uint16 m_indexBuffer[ 8 ];
    Uint16 m_higherValues;

};

struct OctreeeNode
{
    union
    {
        OctreeBranch m_branch;
        OctreeLeaf m_leaf;
    };
    Bool m_isLeaf;
};
```

RUNTIME – CONNECTING NODES

- Find neighboring nodes based on coords from packed representation
  neighbourPos = nodePos + connectionDelta
- Use octree lookup for that
- We don't want to place the vector of neighbors in the node

RUNTIME — CONNECTION SYSTEM

- SLAB like allocator
- Each node count holds its own free list allocator. When we add a connection we move the set to a different array.

```
class SOUND_API ConnectionSystem
{
    using FreeId = Uint32;

    // access  methods

private:

    union Entry
    {
        struct
        {
            Uint16 m_nodeId;
            Uint8 m_sectorId;
            Uint8 m_packedDistance;
        };
        FreeId m_freeIndex;
        Uint32 m_data; // alias used for copying chunks etc.
    };

    red::StaticArray< red::DynArray< Entry >, c_maxConnectionsPerNode > m_entries;
    red::StaticArray< FreeId, c_maxConnectionsPerNode > m_nextFreeId;
    red::StaticArray< Uint32, c_maxConnectionsPerNode > m_freelistCounts;

};
```

RUNTIME – CONNECTION SET ID

- Example – fetch

connection count 1 byte	*inner index* 3 bytes

```
GlobalNodeIndex ConnectionSystem::GetGlobalNodeIndex( ConnectionId connectionSetId, Uint8 inIndex ) const
{
    Uint8 count = connectionSetId >> CONNECION_COUNT_BITSHIFT;
    RED_ASSERT( count );
    Uint32 index = ( connectionSetId & CONNECTION_INDEX_MASK );
    RED_ASSERT( ( index % count ) == 0 );
    index += inIndex;

    auto& entries = m_entries[ count - 1 ];
    return GlobalNodeIndex( entries[ index ].m_sectorId, entries[ index ].m_nodeId );
}
```

RUNTIME – CONNECTION SET ID

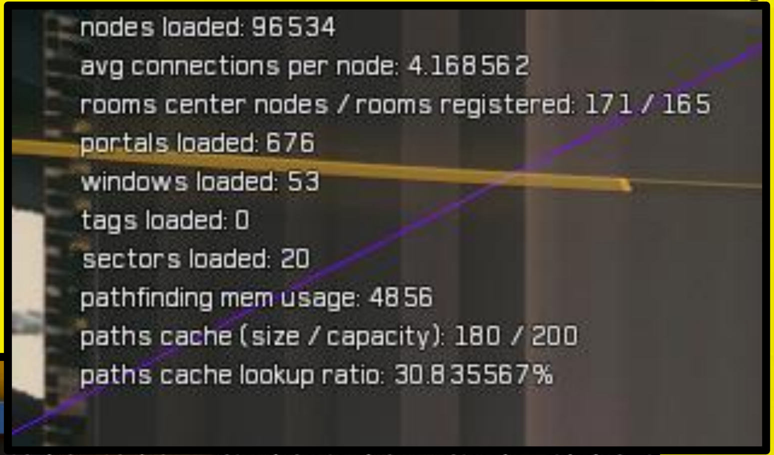
- Example – Add Connection
- passing in out reference to update the current connectionId

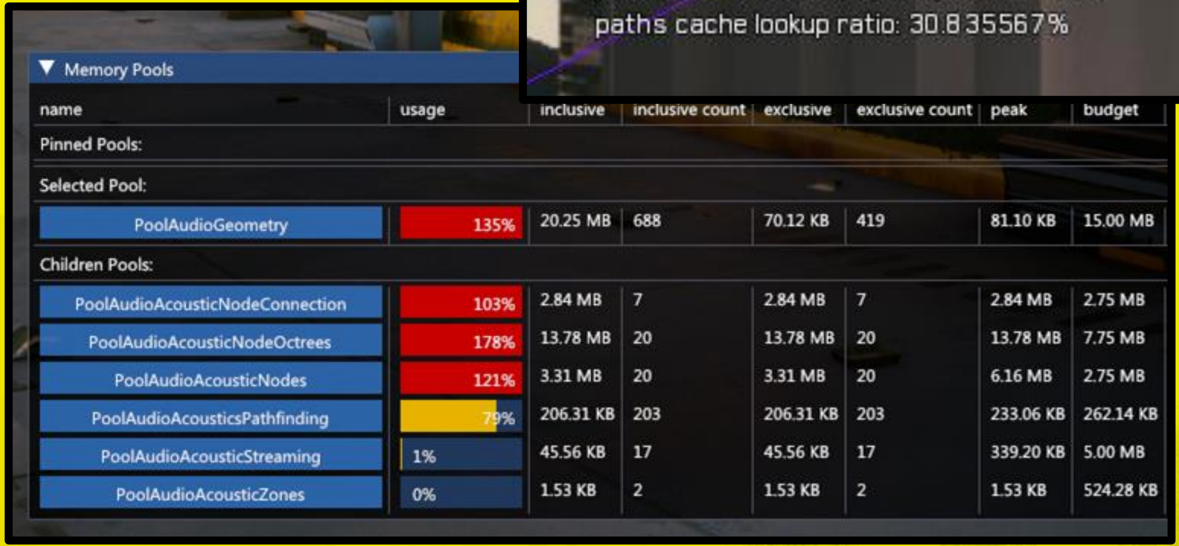
connection count	inner index
1 byte	3 bytes

```
void ConnectionSystem::AddConnection( ConnectionId& connectionSetId, GlobalNodeIndex nodeIndex, Float distance )
{
    Uint8 newCountIndex = connectionSetId >> CONNECION_COUNT_BITSHIFT;

    Uint8 oldCount = newCountIndex;
    Uint8 newCount = oldCount + 1;

    auto& newEntries = m_entries[ newCountIndex ];

    RemoveFormOldArray( oldCount ) ;

    Uint32 newIndex = AddToNewArray()

    newEntries[ newIndex + oldCount ].m_sectorId = nodeIndex.m_smallSectorId;
    newEntries[ newIndex + oldCount ].m_nodeId = nodeIndex.m_nodeId;
    newEntries[ newIndex + oldCount ].m_packedDistance = ConnectionDistanceToChar( distance );

    connectionSetId = ( newCount<<24 ) | newIndex;
}
```

MEMORY – HEAVIEST PLACE

- About 20 MB – slightly more than planned
- Most of the memory is in the octree lookups





Memory Pools

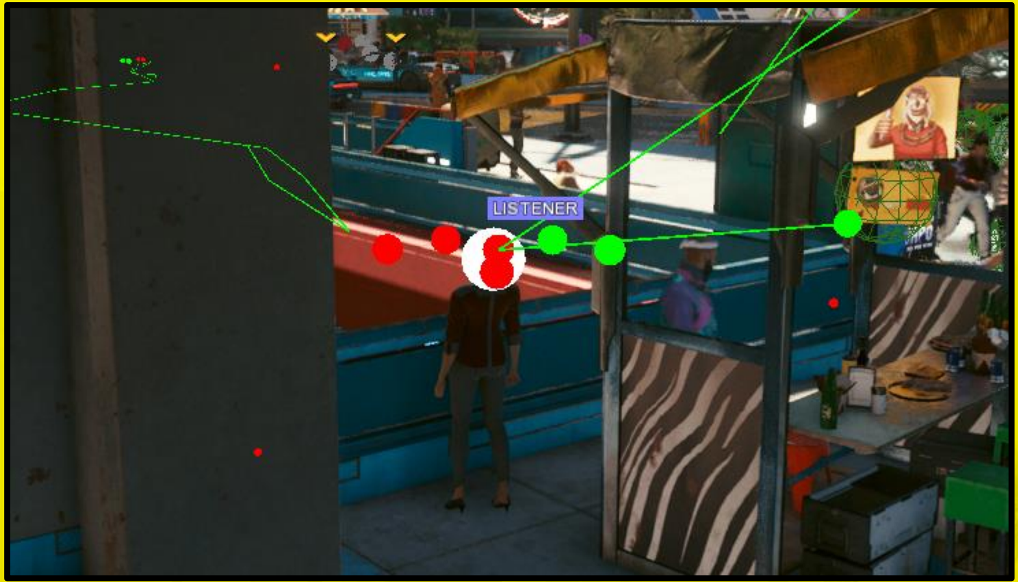
name	usage	inclusive	inclusive count	exclusive	exclusive count	peak	budget
Pinned Pools:							
Selected Pool:							
PoolAudioGeometry	135%	20.25 MB	688	70.12 KB	419	81.10 KB	15.00 MB
Children Pools:							
PoolAudioAcousticNodeConnection	103%	2.84 MB	7	2.84 MB	7	2.84 MB	2.75 MB
PoolAudioAcousticNodeOctrees	178%	13.78 MB	20	13.78 MB	20	13.78 MB	7.75 MB
PoolAudioAcousticNodes	121%	3.31 MB	20	3.31 MB	20	6.16 MB	2.75 MB
PoolAudioAcousticsPathfinding	79%	206.31 KB	203	206.31 KB	203	233.06 KB	262.14 KB
PoolAudioAcousticStreaming	1%	45.56 KB	17	45.56 KB	17	339.20 KB	5.00 MB
PoolAudioAcousticZones	0%	1.53 KB	2	1.53 KB	2	1.53 KB	524.28 KB

RUNTIME – SUMMARY

- Streaming files of 4x4x4 sectors, parsing only the sectors near the player (max 27)
- Spatial lookup via Octree
- Using a custom allocation method for connections

PATHFINDING

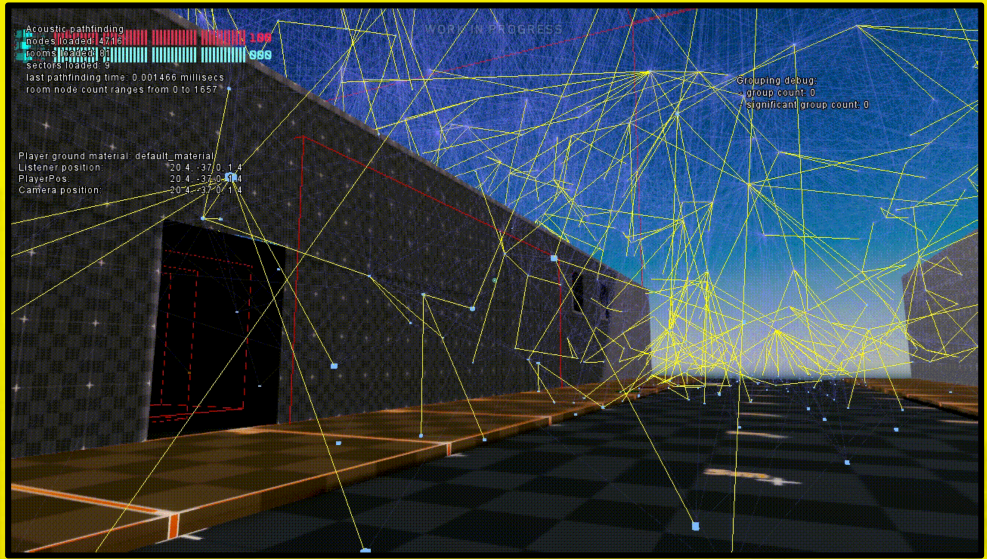
- Finding path from source to the listener
- It handles all playing sounds
- By design there is one center for the paths (didn't apply to ai pathfinding)



PATHFINDING – WHY NOT DIJKSTRA

- It's iterating over unvisited neighbors and picks up the closest path from neighbors distance + dist to neighbor
- Only needs to be updated anytime listener moves
- Can be done across multiple frames

... but it's still slow as hell!



PATHFINDING A*

- Depth first search over the graph with picking the most promising nodes (with cartesian distance)

- Similar to dijkstra
- Once we reach the target we are done (it might miss the best path)

PATHFINDING A* — LIMITS

- Flexible limits based on circumstances but overall max 100 size of closed set, and number of iterations
- Limit reached → path not found
- Path through closed doors → path not found

PATHFINDING A* – PATHS CACHE

- Many sounds play from the same position
- Many sounds don't move during the playback

PATHFINDING A* – PATHS CACHE

- Calculate the hash
- Do we have the path matching this cache?
  - confirm and return or,
- Perform a pathfinding
- Add results in the cache

```
Uint64 GetIntCoordsPairHash( const IntCoords& start, const IntCoords& end )
{
    return
            static_cast< Uint64 >( start.m_x )
        + ( static_cast< Uint64 >( start.m_y ) << 10 )
        + ( static_cast< Uint64 >( start.m_z ) << 20 )
        + ( static_cast< Uint64 >( end.m_x ) << 30 )
        + ( static_cast< Uint64 >( end.m_y ) << 40 )
        + ( static_cast< Uint64 >( end.m_z ) << 50 );
}
```

```
Bool PathsCache::QueryPath(
    const IntCoords& startPosition,
    const IntCoords& endPosition,
    PathfindingResult& result )
{
    lock(lock);

    CachedPath* foundPath =
        m_cacheLookup.FindPtr(
            GetIntCoordsPairHash( startPosition, endPosition )
        );

    // checking if path got outdated
    if( !foundPath || foundPath->second <= .0f )
    {

        return false;
    }

    //checking if hash didn't fool us
    if( foundPath->first.m_start != startPosition
        || foundPath->first.m_end != endPosition )
    {
        return false;
    }

    result = foundPath->first;
    return true;

}
```

PATHFINDING A* – PATHS CACHE

- c_pathCacheTime = 0.5;
- c_maximumCacheSize = 200;

- Hit rate over 50%

```
void PathsCache::Tick( float deltaTime )
{
    //...

    // second for the value in the lookup is float ttl
    for( auto elem : m_cacheLookup )
    {
        elem.Value().second -= deltaTime;

        if( elem.Value().second <= .0f && !toRemoveList.Full() )
        {
            toRemoveList.PushBack( elem.Key() );
        }
    }

    //...
}
```

PATHFINDING: INTERPRETING PATH

Possible approaches:

- Compare path length with cartesian distance
- Use average deviation from the direct straight line connecting start and end
- Use max deviation from the direct straight line connecting start and end
- Increase occlusion when walking through small nodes
- Increase occlusion by walking through the doors

Trial and error

PATHFINDING: INTERPRETING PATH

Used approaches:

- ~~Compare path length with cartesian distance~~
- ~~Use average deviation from the direct straight line connecting start and end~~
- Use max deviation from the direct straight line connecting start and end
- ~~Increase occlusion when walking through small nodes~~
- Increase occlusion by walking through the doors

Multiply inverse occlusion

PATHFINDING: SUMMARY

- Using A* with max 100 iterations
- Caching 200 paths, with path TTL = 500 ms
- Occlusion = MaxDeviation, additionally halving the opacity when passing through doorway

Cyberpunk 2077

CAN THOSE GANG MEMBERS HEAR ME?

ACOUSTICS FOR THE NPCS

- It all started with an internal highlights feed...

What if we used your acoustics graph, check the occlusion from NPC to some event and we'd know if the NPC can hear that explosion, or player sneaking, or a soda machine deliberately broken?

ACOUSTICS FOR THE NPCS – BENEFITS

- Additional QA
- Contribution to the actual gameplay

ACOUSTICS FOR THE NPCS

RAYCAST + GRAPH COMBO

- Using the most optimistic results (to solve some tricky edge cases)…
- … unless the distance is too big (and the physics is not streamed in).

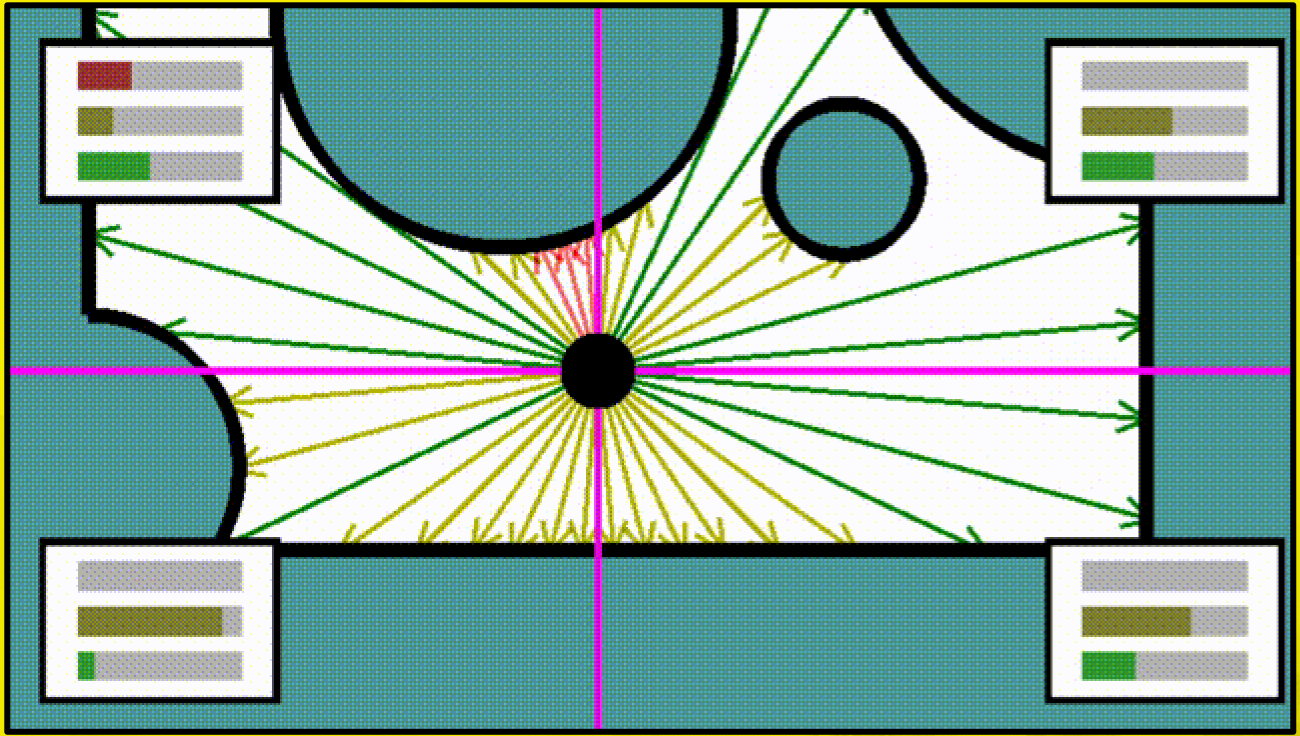
```
//...
if( GetDecisionFromPFResult( *data.m_pendingQuery->GetResult(), maxPathLength, distance ) )
{
    decision = EffectObjectFilterDecision::Accept;
}
else if( maxPathLength > distance || distance > s_fallbackRaycastDistanceTreshold )
{
    decision = EffectObjectFilterDecision::Reject;
}
else
{
    data.m_pathfindingFailed = true;
    data.m_raycastToken = RequestRaycast( commonContext, position, filteringContext );
    // keep processing
}
//...
```

ACOUSTICS FROM THE DEPTHS

HEDGEHOG

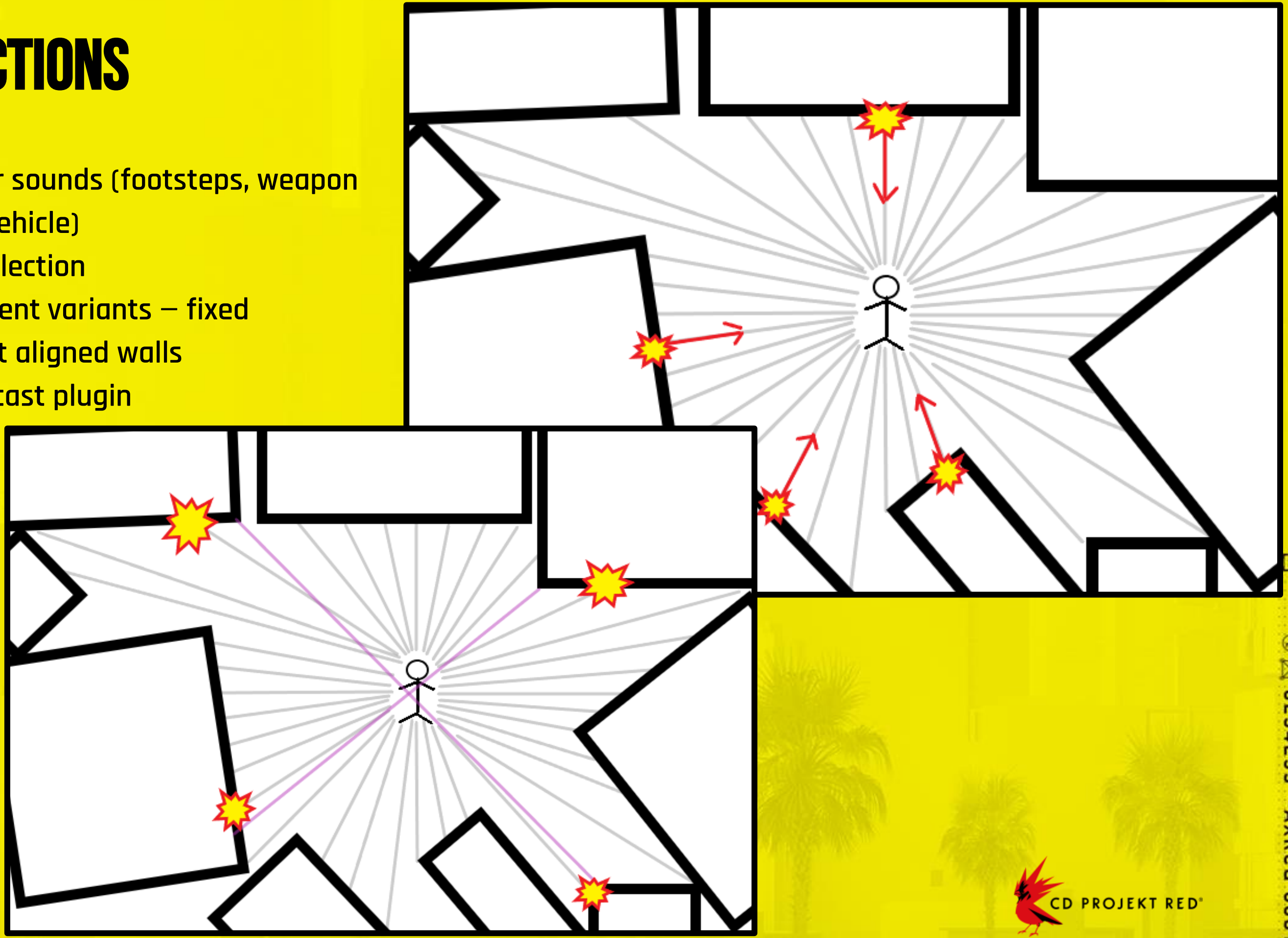
- 36 horizontal directions
- 7 vertical angles
- 36 raycasts per frame – one pitch direction

HEDGEHOG VS. DYNAMIC REVERB



EARLY REFLECTIONS

- Mainly for player sounds (footsteps, weapon sounds, player vehicle)
- Just the first reflection
- We tested different variants – fixed directions / most aligned walls
- Using the broadcast plugin



WEAPON TAILS

- Picking prebaked tails
- Using statistics:
  - ceiling distance,
  - avg horizontal distance,
  - average elevated distance
  - outdoorness factor

Newly calculated stats contributed to environmental sounds as well

Cyberpunk 2077

GO AND BUILD ACOUSTICS

FINAL THOUGHTS

- Every bit of memory counts
- Automated testing
- Talk to people, ask for ideas
- There is always room for improvement (Not enough^{tm})

QUESTIONS:

VOXELIZATION?
A*?
THAT'S BULL#%^T!
MEMORY?
SECTOR EDGE?
TESTING SUCKS!
RAYCAST?
OCTREE?
FREELIST?
SSE?

CD PROJEKT RED

USER ID_99877 LENS DRVR 3.8.83
STATUS: ONLINE SERIAL>34 73.32

8234299 -AXRCG 0001