MARCH 18-22, 2024 #GDC2024

ECS in practice
The case board of Alan Wake 2

ALEXANDER BALAKSHIN, SENIOR GAMEPLAY PROGRAMMER
03–21–2024

DISCLAIMER

THIS TALK IS NOT ABOUT

- Effective memory access and all allocation shenanigans
- Parallelization (though we’ll still cover that)
- ECS as a silver performance bullet in general

THIS TALK IS ABOUT

- An example of how to apply ECS for non-common gameplay programmer tasks
- How a gameplay programmer's mindset is changed from "working with objects" to "working with data"

WARNING

- The presentation contains some C++ code snippets
- But I promise to explain :))

INTRODUCTION

Hello, my name is Alexander 'Sasha' Balakshin and I'm a gameplay programmer!

- In the game development industry since 2014

- Worked on Rainbow Six: Siege, Halo: Infinite, Atomic Heart, and the most recent – Alan Wake 2

- I had a chance to work with multiple game engines and frameworks both publicly accessible (Unreal Engine, Unity) and proprietary ones.

- Most of the gameplay frameworks I worked with were object-oriented in some kind of way

REMEDY ENTERTAINMENT

- Finnish video game developer based in Espoo
- Founded in 1995 (we’re turning 30 next year!)
- Author of such games as Max Payne 1&2, Quantum Break, Control and Alan Wake 1&2

MARCH 18–22, 2024 #GDC2024

REMEDY ENTERTAINMENT PLC.

FOUNDED IN AUGUST
1995

HEADQUARTERS IN ESPOO
FINLAND

STUDIO IN STOCKHOLM
SWEDEN

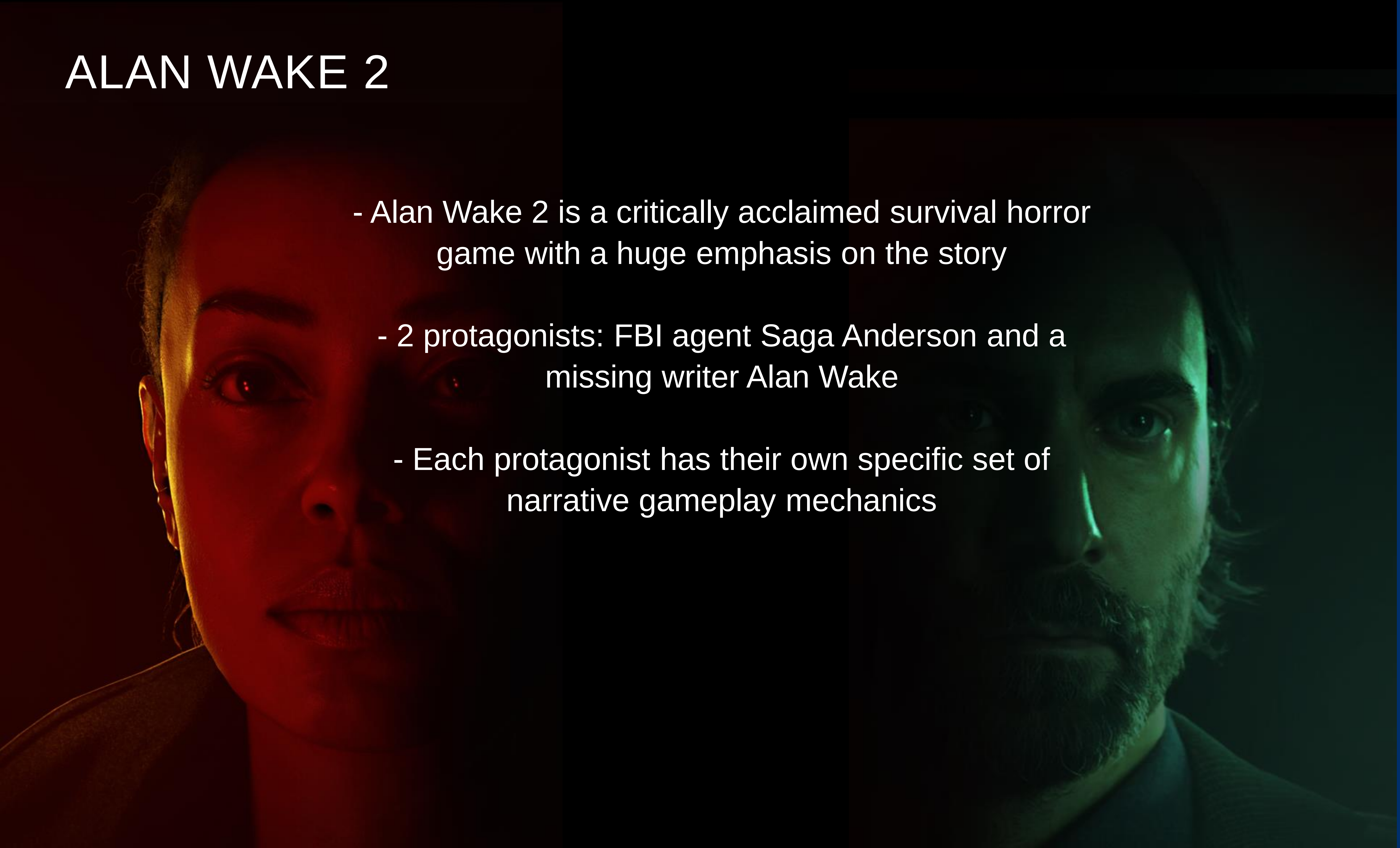
LISTED AS PUBLIC COMPANY IN
2017
(NASDAQ FIRST NORTH)

TRANSFERRED TO MAIN LIST
2022
(NASDAQ HELSINKI)

APPROXIMATELY
380
EMPLOYEES

33
NATIONALITIES

ALAN WAKE 2



- Alan Wake 2 is a critically acclaimed survival horror game with a huge emphasis on the story

- 2 protagonists: FBI agent Saga Anderson and a missing writer Alan Wake

- Each protagonist has their own specific set of narrative gameplay mechanics

ECS FRAMEWORK IN NL

NORTHLIGHT

Northlight is a focused, state-of-the-art game engine and toolset developed by Remedy Entertainment, empowering our games now and in the future.

Constantly evolving tech for the creation of industry-leading game visuals:
- Top-tier facial animations for Quantum Break
- The first hardware ray-tracing in Control)

For Alan Wake 2 Northlight has even more improvements:
- GPU driven pipeline
- SDF-base wind system
- foliage rigs.
- …
- more on https://www.remedygames.com/article/how-northlight-makes-alan-wake-2-shine

northlight

REMEDY'S IN-HOUSE
GAME ENGINE

A NEW ECS GAMEPLAY FRAMEWORK IN NL ENGINE

– In 2021 a new ECS ("Entity Component System") object model was taken into use in NL.

– It replaced the previously used heavy OOP entity-component framework with pre-defined functions and event graphs connecting different components between entities.

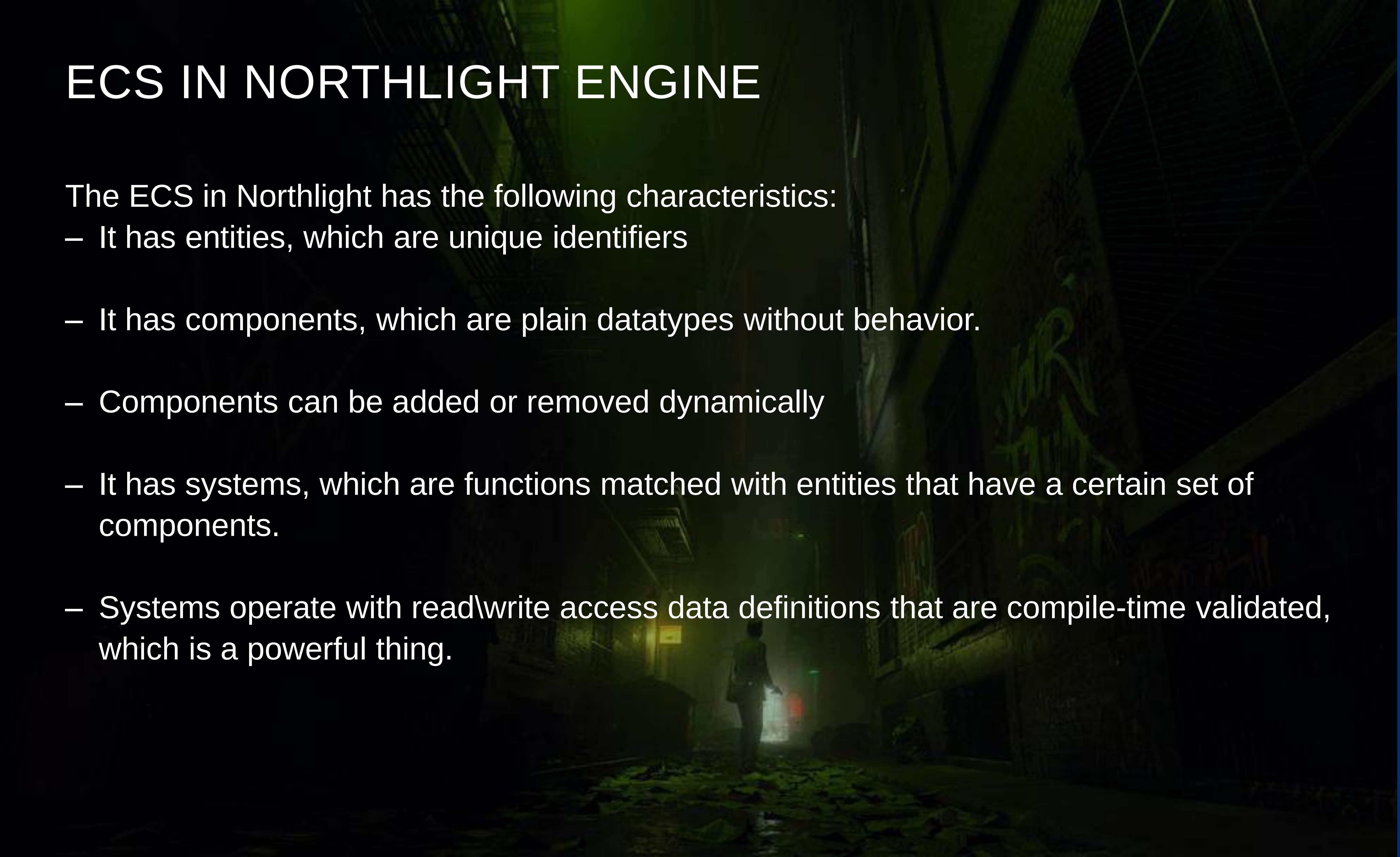
– ECS is a software design approach that promotes code reusability by separating data from behavior. Data is stored in a cache-friendly way, which benefits performance.

– A cache-friendly data layout and easiness of parallelization of NL ECS framework benefits performance.

– What is more important, it makes it easier for gameplay programmers to understand the program flow and write a game's logic

northlight
REMEDY'S IN-HOUSE
GAME ENGINE

MARCH 18–22, 2024 #GDC2024



ECS IN NORTHLIGHT ENGINE

The ECS in Northlight has the following characteristics:

- It has entities, which are unique identifiers

- It has components, which are plain datatypes without behavior.

- Components can be added or removed dynamically

- It has systems, which are functions matched with entities that have a certain set of components.

- Systems operate with read\write access data definitions that are compile-time validated, which is a powerful thing.

ECS FRAMEWORK OVERVIEW: COMPONENTS

– Components are any plain data structures that are meant to represent a state
– They should not have any behavior, only state.
– If you need to provide a public API to process the component data in some way, make a free function accepting this component as a parameter instead of a method

```
struct CurrentHealth
{
    float m_value; // << small and trivial
};

// some public routines
void takeDamage( CurrentHealth& health, float damageValue ) // you encapsulate the logic and require the caller to
have a proper access to component
{
    health.m_value -= damageValue;
}

struct CollisionShape
{
    r::ScopedPtr< ShapeData > m_data;
}

struct ActiveAvatar {}; // changed relatively rarely, allows you to efficiently group entities for further processing
```

ECS FRAMEWORK OVERVIEW: ENVIRONMENTS

- Environments keep data that should be accessed globally in the game
- Reside directly in the memory, and they are not tied to any entity.
- Only a single instance of any environment type may exist in the game.

```
namespace env
{
    struct GameCursor
    {
        m::Vector2 m_position;
        bool m_isEnabled;
    }
}

void updateCursorPosition( GameCursor& gameCursor )
{
    gameCursor.m_position = input::analog2d( input::game_and_menu(), input::actions::GAME_CURSOR_POSITION );
}

void drawCursor( const GameCursor& gameCursor )
{
    if( gameCursor.m_isEnabled )
    {
        showCursorAtPosition( gameCursor.m_position )
    }
}
```

ECS FRAMEWORK OVERVIEW: ENTITIES

- Entity is a **conceptual** object that is composed of data structures called components, each representing certain functionality associated with that object.

- An entity is **defined** by the set of components attached to it. There's no such thing as an "**object**" of an entity.

- An entity is identified by a 32-bit integer number

- Still, there's a way to represent single or multiple entities in the code and get access to its components

ECS FRAMEWORK OVERVIEW: ACCESS DEFINITIONS

- Access definition is a template data structure that allows to define the access to the data

- Required for the ECS to understand your dependencies.

```
using DamagedEnemiesAccess = ecs::Access
    ::Read< CharacterComponent > // Define a read dependency on `CharacterComponent`
    ::Read< TakenDamageComponent > // Define a read dependency on `TakenDamageComponent`
    ::ReadIfExists< EffectsComponent > // Define a potential read dependency on the `EffectsComponent`.
    ::Write< HealthComponent > // Define a write dependency on the `HealthComponent`
    ::WriteIfExists< HitReactionComponent > // Define a potential write dependency on the
                                            // `HitReactionComponent`
    ::With< EnemyComponent > // Defines a filter on `EnemyComponent` - does not allow you to read or
                             // write the component, but affects a query scope, which will only return
                             // entities that has the `Component`
    ::Without< DeadComponent > // Similar to `::With`, but will only return entities that do NOT have
                                // the `DeadComponent`
```

ECS FRAMEWORK OVERVIEW: TYPES OF ACCESSES

The definition chain is usually finished with a specified access scope identifying if need to read a data from a single entity or from a collection

```
using DamagedCharacter = ecs::Access
    ::Read< CharacterComponent >
    ::Write< HealthComponent >

// an access definition is specified with ::As< Accessor > at the end of the chain

using DamagedCharacterEntity = DamagedCharacter::As< ecs::Entity > // Localized access to a single entity with ALL the
components. Used as a "main filter" for a system.

using DamagedCharacterGroup = DamagedCharacter::As< ecs::Group > // Localized access to ALL entities with ALL the components
(but may have more). Also used as a "main filter"

using DamagedCharacterQuery = DamagedCharacter::As< ecs::Query > // Localized access to ALL entities with ALL the components
(but may have more). Allows arbitrary access within the query as well as iteration over multiple groups. Does not affect how
the system will be called, but affects the scheduling, effectively "locking" all groups that match the definition
```

ECS FRAMEWORK OVERVIEW: SYSTEMS

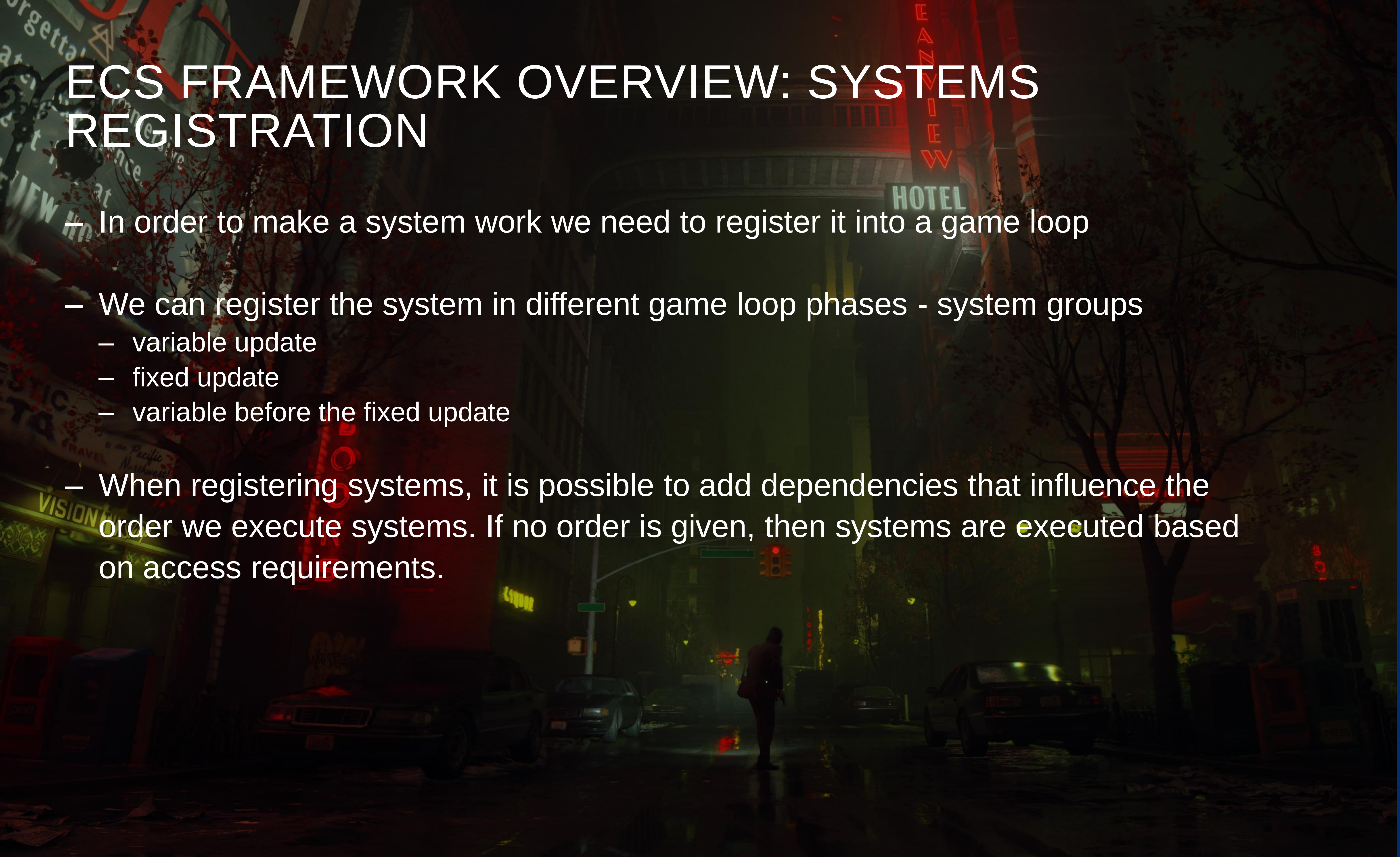
- An ECS system is a free function that can receive some input, does processing, and produces an output to components.

- System may also add or remove components.

- Systems use **accesses** and **environments** as arguments

ECS FRAMEWORK OVERVIEW: SYSTEMS (EXAMPLE)

```
using DamagedCharacterEntity = ecs::Access
    ::Read< TransformComponent >
    ::Write< HealthComponent >
    ::With< CharacterComponent >
    ::As< ecs::Entity >

using DamageSourceQuery = ecs::Access
    ::Read< DamageSourceComponent >
    ::Read< TransformComponent >
    ::As< ecs::Query >

void dealDamageOverTimeSystem( DamagedCharacterEntity entity, DamageSourceQuery query, const env::FixedTime& fixedTime )
{
    const auto& characterTransform = entity.getComponent< TransformComponent >();
    auto& characterHealth = entity.getComponent< HealthComponent >();

    for( auto damageSourceEntity : query )
    {
        auto[ damageSource, damageTransform ] = damageSourceEntity;
        if( transform::distance( characterTransform, damageTransform ) < damageSource.distance )
        {
            characterHealth.m_healthValue -= damageSource.m_damageOverTimeValue * fixedTime.m_deltaTime;
        }
    }
}
```



ECS FRAMEWORK OVERVIEW: SYSTEMS REGISTRATION

- In order to make a system work we need to register it into a game loop

- We can register the system in different game loop phases - system groups
  - variable update
  - fixed update
  - variable before the fixed update

- When registering systems, it is possible to add dependencies that influence the order we execute systems. If no order is given, then systems are executed based on access requirements.

ECS FRAMEWORK OVERVIEW: SYSTEMS REGISTRATION

```
void VariableUpdateSystem( Entity entity, QueryB query, const env::VariableUpdate& variableUpdate ) { /*
... */ }


void FixedUpdateSystemA( Entity entity, QueryA query, const env::FixedUpdate& fixedUpdate ) { /* ... */ }
void FixedUpdateSystemB( Entity entity, QueryB query, const env::FixedUpdate& fixedUpdate ) { /* ... */ }


void registerModule( ECSBuilder& builder )
{
    auto variableUpdate = builder.gameloop()[ sg_svVariableUpdateSystemGroupName ];
    auto fixedUpdate = builder.gameloop()[ sg_svFixedUpdateSystemGroupName ];

    auto variableSystem = varibleUpdate.registerSystem< VariableUpdateSystem >();

    auto systemA = fixedUpdate.registerSystem< FixedUpdateSystemA >();

    auto systemB = fixedUpdate.registerSystem< FixedUpdateSystemB >()
        .order()
        .executeAfter( systemA );
}
```

THE CASE BOARD

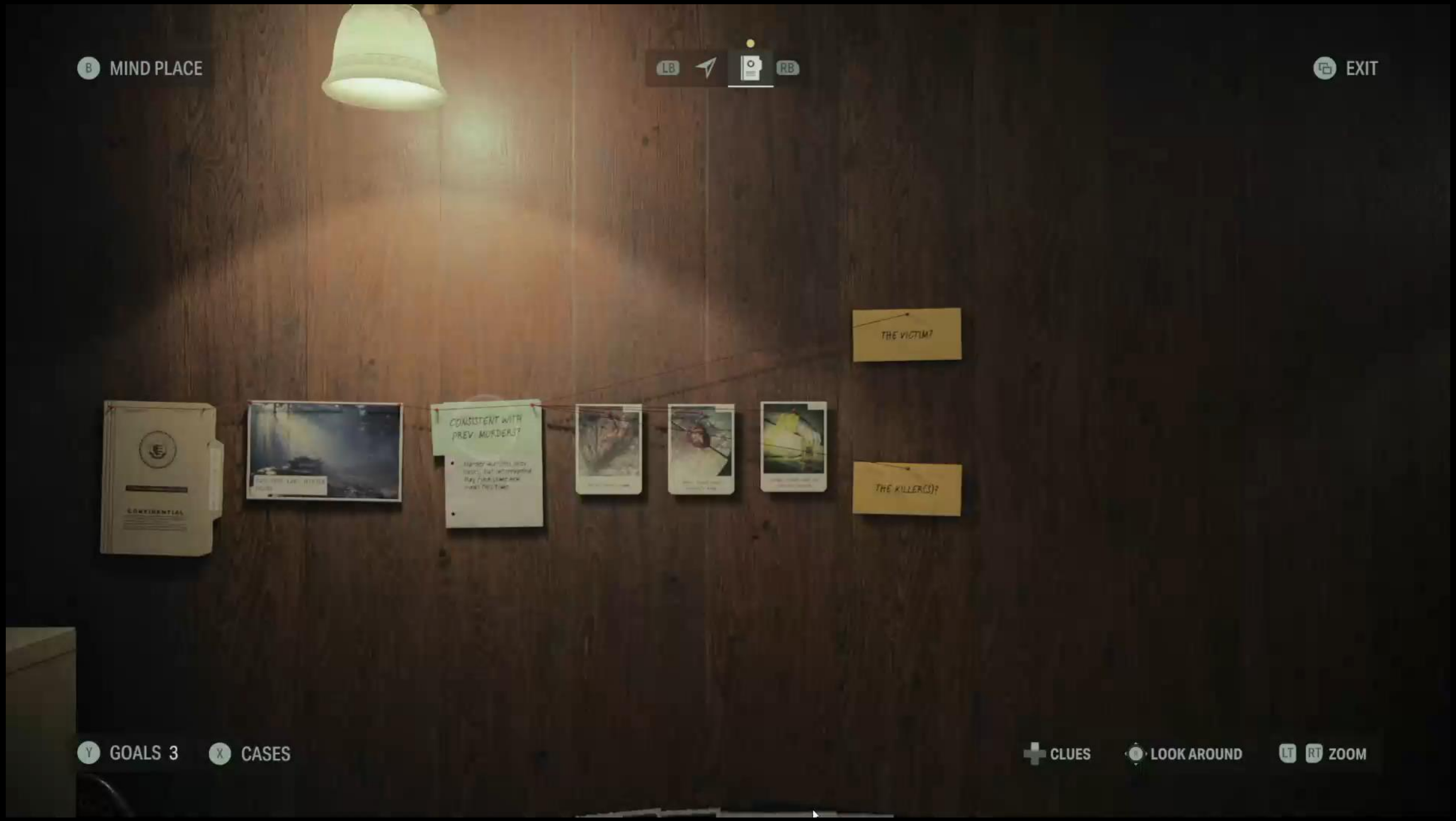
THE CASE BOARD

- Saga's visual storyboard for gathering evidence.
- She can progress investigations by placing clues and combining them with other ones that she finds during the game.
- Works primarily to help a player follow the story and be more involved in that.
- Some events in the game can only be triggered by certain case board combinations.

THE CASE BOARD REQUIREMENTS

– A player collects various clues during the game (interactions, pickups, investigation dioramas, and scripted events)
– https://youtu.be/FKuY8TvV5w4

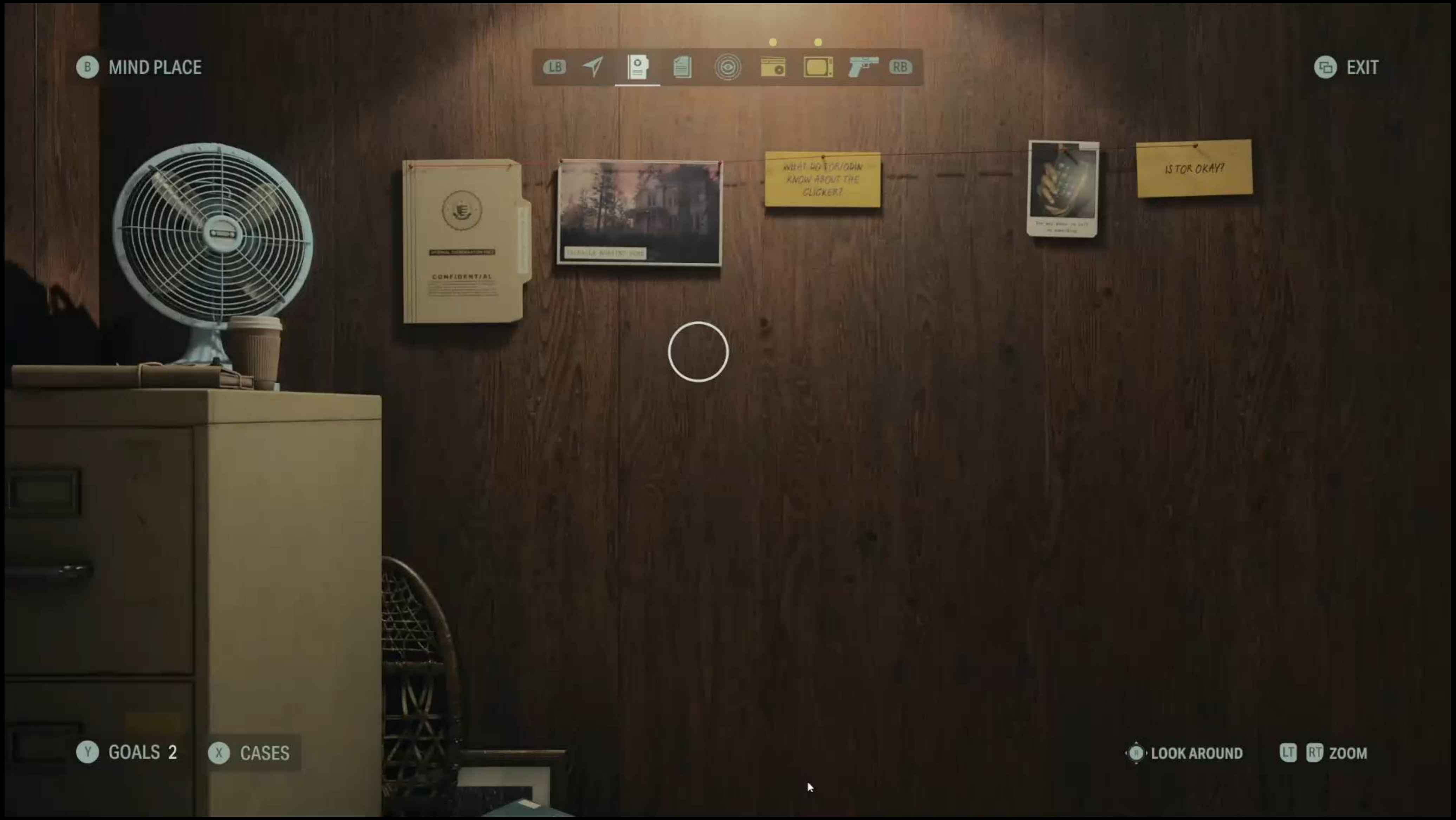
THE CASE BOARD REQUIREMENTS



– A player places clues on a board by dragging them from the hand and combining them with other clues https://youtu.be/V7ts6BcgZ5Q

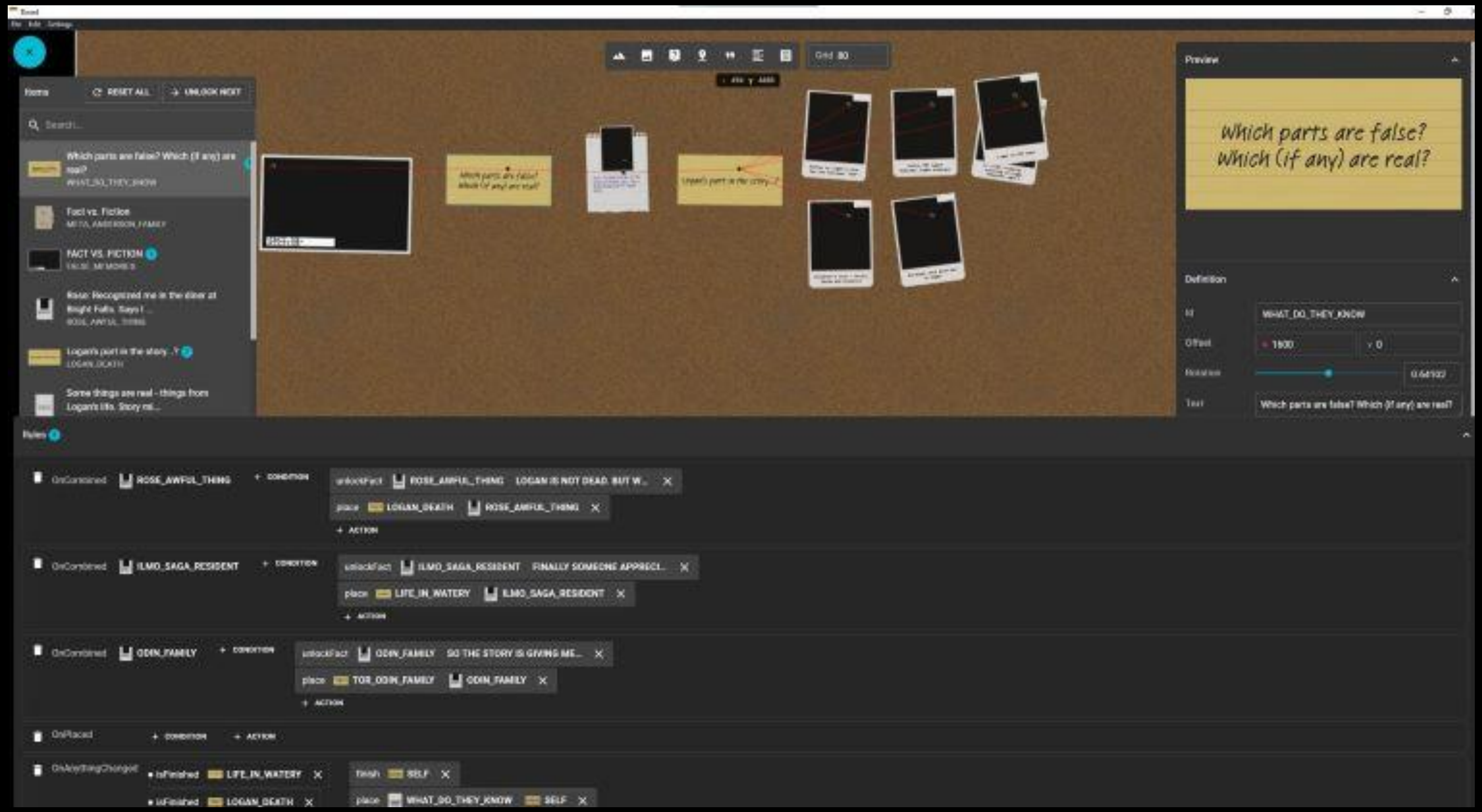
– Some of the clues could be placed\unlocked depending on the case board events itself

THE CASE BOARD REQUIREMENTS: INVESTIGATIONS



– There are multiple investigations during the game and a player can switch between them https://youtu.be/2ezndzY6jNU

THE CASE BOARD REQUIREMENTS: AUTHORING CASES



Which parts are false? Which (if any) are real?
WHAT_DO_THEY_KNOW

Fact vs. Fiction
META_ANDERSON_FAMILY

FACT VS. FICTION
FALSE_MEMORIES

Rose: Recognized me in the diner at Bright Falls. Says I ...
ROSE_AWFUL_THING

Logan's part in the story ..?
LOGAN_DEATH

Some things are real - things from Logan's life. Story mi...

Which parts are false?
Which (if any) are real?

Id: WHAT_DO_THEY_KNOW
Offset: x 1800, y 0
Rotation: 0.64102
Text: Which parts are false? Which (if any) are real?

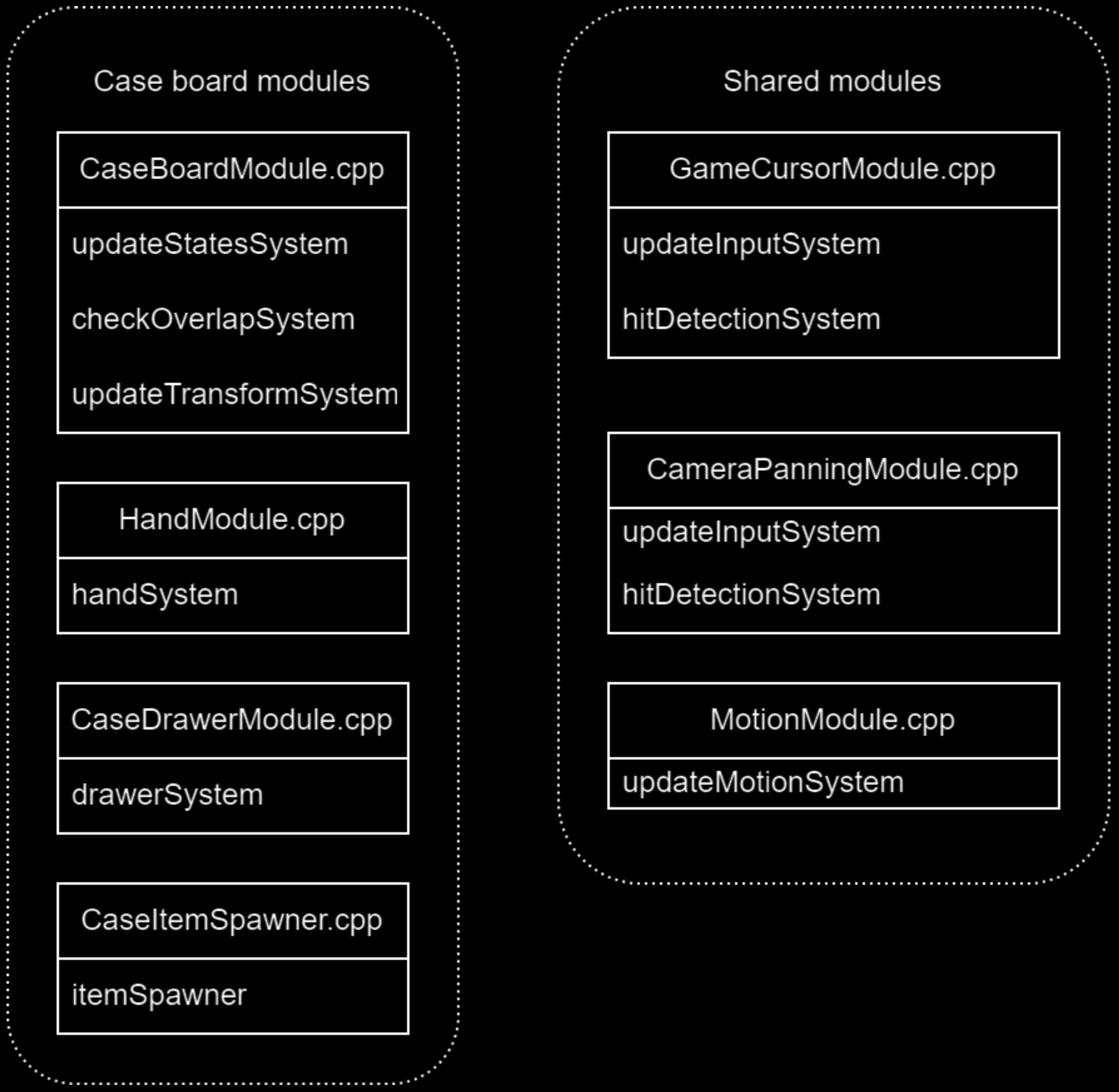
```
OnCombined   ROSE_AWFUL_THING   + CONDITION   unlockFact  ROSE_AWFUL_THING  LOGAN IS NOT DEAD. BUT W...
                                              place  LOGAN_DEATH  ROSE_AWFUL_THING
                                              + ACTION
OnCombined   ILMO_SAGA_RESIDENT + CONDITION   unlockFact  ILMO_SAGA_RESIDENT  FINALLY SOMEONE APPRECI...
                                              place  LIFE_IN_WATERY  ILMO_SAGA_RESIDENT
                                              + ACTION
OnCombined   ODIN_FAMILY   + CONDITION   unlockFact  ODIN_FAMILY  SO THE STORY IS GIVING ME...
                                         place  TOR_ODIN_FAMILY  ODIN_FAMILY
                                         + ACTION
OnPlaced   + CONDITION   + ACTION
OnAnythingChanged   • isFinished  LIFE_IN_WATERY    finish  SELF
                    • isFinished  LOGAN_DEATH       place  WHAT_DO_THEY_KNOW  SELF
```

– The case layout is defined by an editing tool, created by Remedy's lead UI/UX designer Riho Kroll. This allows narrative designers to author cases in a convenient WYSIWYG way.

– The case layout is exported to a configuration file (.lua) and should be used by the game

THE CASE BOARD: CODE

THE CASE BOARD ARCHITECTURE



- Code is structured with a set of module files

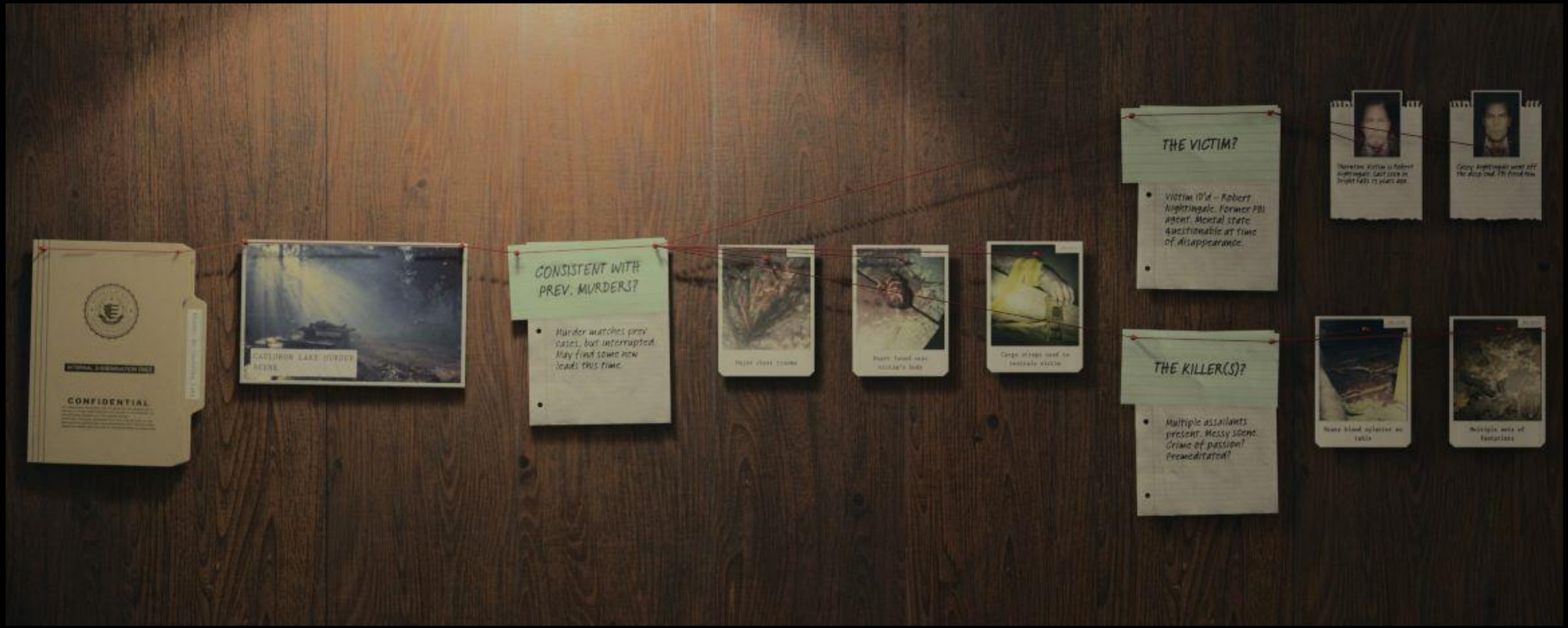
- Module at least have a header and .cpp file

- Each module has one or multiple systems

- For the case board we have a collection of modules for each specific aspect of the feature

- We also have other modules shared with other gameplay systems

CASE BOARD ELEMENTS: ITEMS

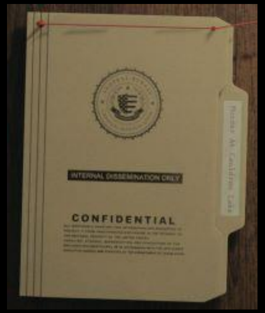

CONFIDENTIAL
CAULDRON LAKE MURDER SCENE
CONSISTENT WITH PREV. MURDERS?
Murder matches prev cases, but interrupted. May find some new leads this time.
THE VICTIM?
Victim ID'd – Robert Nightingale. Former FBI agent. Mental state questionable at time of disappearance.
THE KILLER(S)?
Multiple assailants present. Messy scene. Crime of passion? Premeditated?

- Each case on the board is a "tree" of items
- Items are the main objects the player operates with on the case board
- When some action is done with an item (i.e. placed, unlocked, combined) one or few **rules** can be triggered
- Each **rule** can invoke some other actions (placing an item, unlocking to hand, removing an item, etc.) in case necessary conditions are met (some items placed, unlocked, etc)
- Items can be placed on a board in the following ways
  - By a player from the "hand"
  - By triggering a rule
  - Explicitly from a script

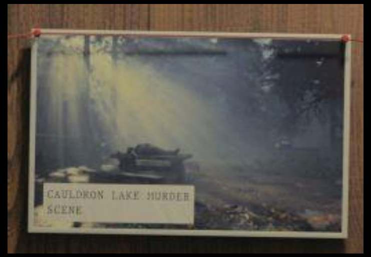
CASE BOARD ELEMENTS: ITEMS

Items can be of the following types:

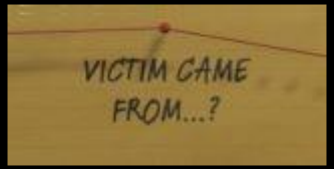
- Case - an investigation folder item. Parent of all items. Placed automatically.



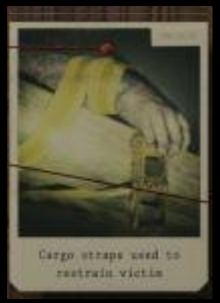
- Photo - starts a new investigation branch. Placed from hand

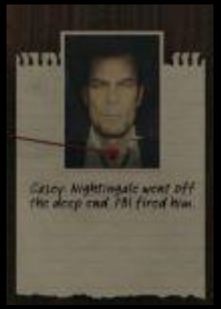


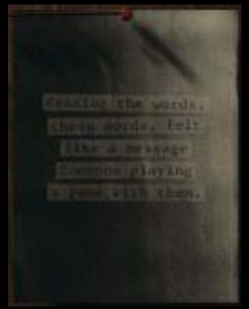
- Question - an investigation problem. Placed automatically (mostly by processing item rules).



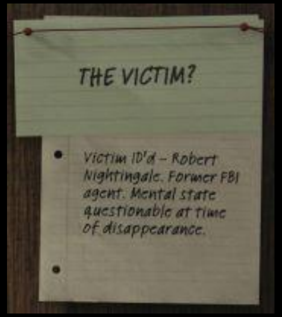
- Clues: polaroids, quotes, manuscript pages. Collected by a player during the game. Placed from a hand. Can be placed on a board only when successfully combined with a question






Reading the words, three words, felt like a message. Someone playing a game with them.

- Deductions - placed automatically under the question when all required clues have been combined with a question


THE VICTIM?

- Victim ID'd – Robert Nightingale. Former FBI agent. Mental state questionable at time of disappearance.

CASE BOARD ENVIRONMENT

- We store all cases (names, collection of items) and item descriptions and states (placed, unlocked, parent) in a single **environment**

- This becomes very useful from the following perspectives:

- It's convenient storage for loading data from configuration files
  - It's a synchronization point. Whenever we want to modify the current case board's logical state, especially from scripts, we work with the case board's environment
  - It's very helpful for saving and loading the case board's state

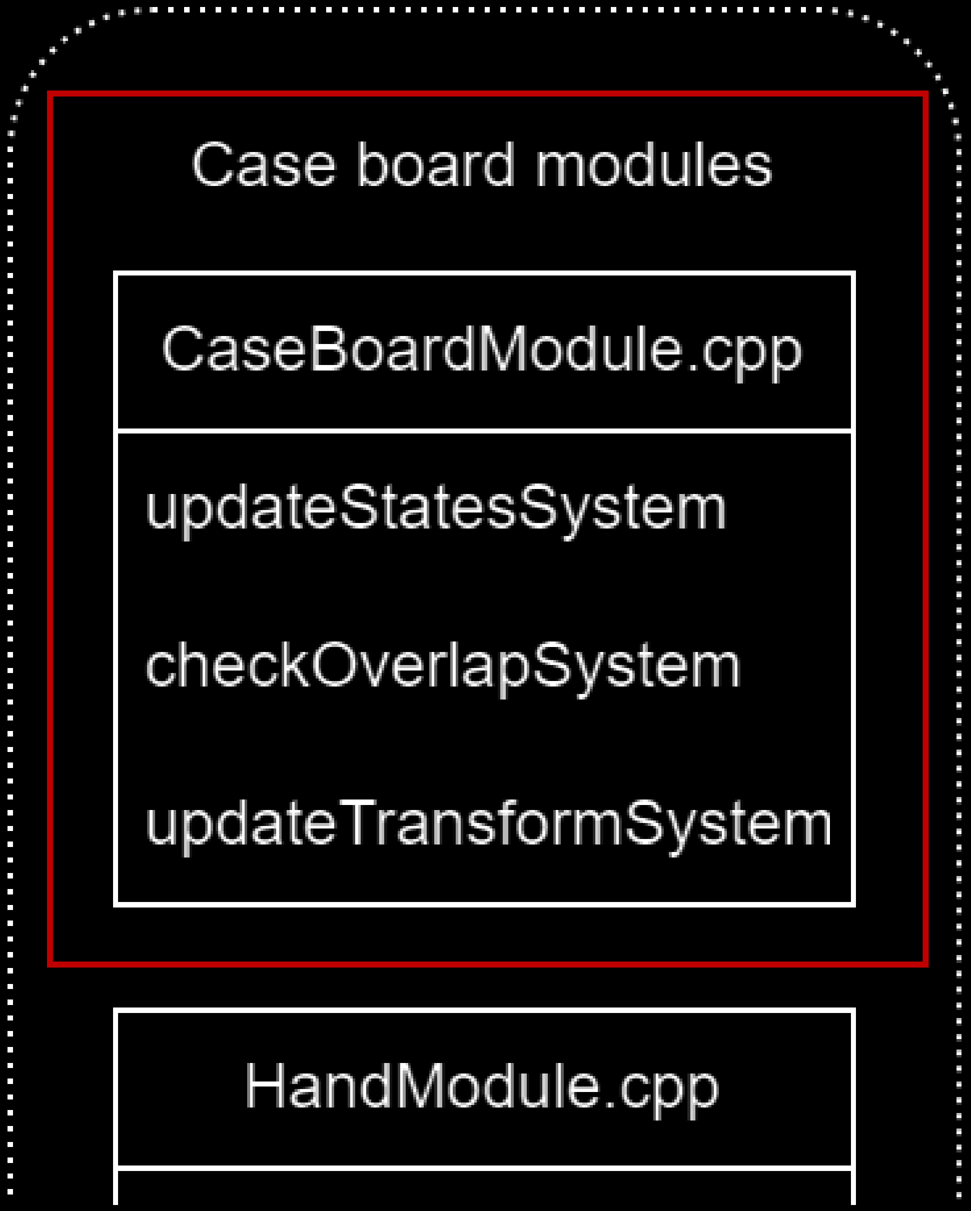
```
struct ItemState
{
    r::String parentId;
    m::Vector2 offset;
    bool isPlaced;
    bool isUnlocked;
}

struct Item
{
    r::String itemId;
    ItemState state;
    r::Vector< Rules > rules;
}

struct Case
{
    r::String caseId;
    r::Vector< Item > items;
}

namespace env
{
    struct CaseBoard
    {
        r::Vector< Case > cases;
        bool isEnabled;
        // … other member-variables …
    }
}
```

CASE BOARD: MAKING ACTION WITH AN ITEM

- With this approach making any action with an item will be implemented like this
- It's a free C++ function that takes itemId and a mutable reference to a case board environment as arguments
- In this function we will do the following steps:
  - Find a mutable pointer to an item in the environment
  - Modify necessary state
  - Do action-specific logic
  - Process action-specific rules

```cpp
void placeItem( const r::String& itemId, env::CaseBoard& caseBoard )
{
    Item* item = findItem( itemId, caseBoard );
    item.state.isPlaced = true;
    // Some item placement logic
    processItemRule( item, caseBoard, Rule::Type::OnPlaced );
}
```

CASE BOARD MODULE

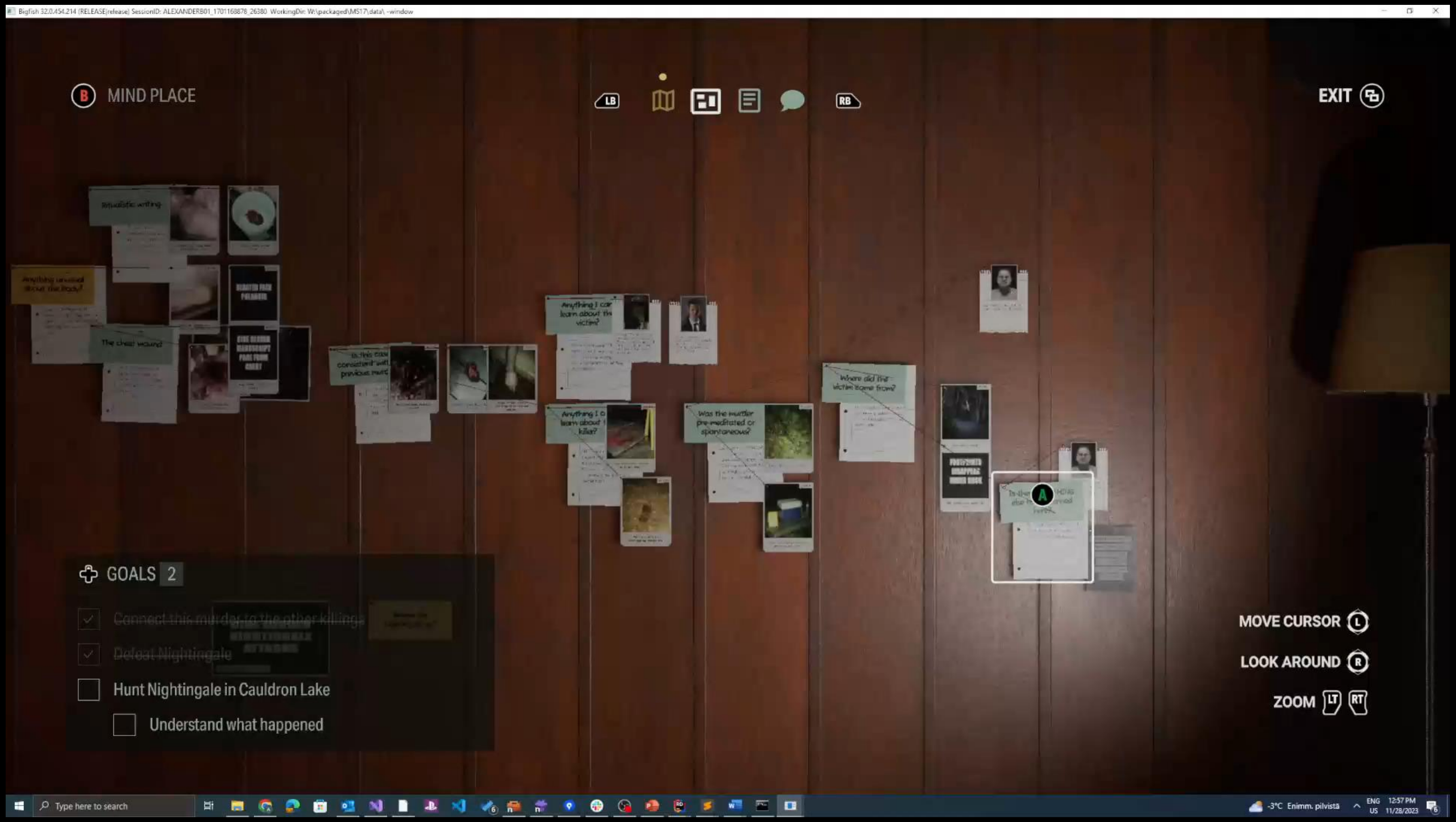


– This module works with "real world" item entities and is mainly responsible for:
  - Updating item entities states
  - Updating transforms of items on a board

– Logic is split between different systems. It’s totally necessary since calculating and applying the final transform of an is **not as easy** as it seems to be.

– It also allowed us
  - Avoid locking unnecessary data and as a result blocking the execution of other gameplay systems (i.e. transforms)
  - Running systems in parallel when possible
  - Making sure that a specified logic is applied to all required entities before we switch to the next step

CASE BOARD MODULE: CALCULATING TRANSFORM



- For quite a long time, items on the board could be freely moved with the cursor.
- When moving an item, all child items should also move with keeping their offsets to parent
- Items can overlap and stack on top of each other.
- Also, their position should be clamped to the board's limits.

CASE BOARD MODULE: TRANSFORM SYTEMS

The whole positioning mechanic is split into multiple systems, being executed one after another:

1. Update the state and offsets, particularly of a “real world” entity from the ones set in the case board **environment**

2. Set items XY position and boundaries based on offsets values and the case board boundaries.

3. Check overlaps for each item and find which items lie underneath each other based on hierarchy level, item type, item index, last interaction, etc.

4. Sort items by depth based on overlap info and compute board-relative Z value

5. Set items final transform (either directly or via animation)

CASE BOARD MODULE

```
void registerModule( ECSBuilder& builder )
{
    auto variableUpdate = builder.gameloop()[ sg_svVariableUpdateSystemGroupName ];

    // Compare states with the ones stored in environment, apply them to real-world entities
    // and do some actions if necessary (change visibility and etc.)
    auto updateItemsStates =  variableUpdate.registerSystem< updateStates >();

    // Apply current items XY position and AABB boundaries in a board's space
    auto updateItemBoundaries =  variableUpdate.registerSystem< updateItemBoundaries >()
        .executeAfter( updateItemStates );

    // Check AABB overlaps and store items that are below of other ones
    auto checkOverlaps = variableUpdate.registerSystem< checkOverlaps >()
        .executeAfter( updateItemBoundaries );

    // Set items Z value based on overlaps info
    auto updateItemDepth = variableUpdate.registerSystem< updateItemDepth >()
        .executeAfter( checkOverlaps );

    // Set an item's final transform
    auto updateItemsTransforms = variableUpdate.registerSystem< updateItemTransforms >()
        .executeAfter( updateItemDepth );
}
```

CASE BOARD MODULE: STATES UPDATE SYSTEM

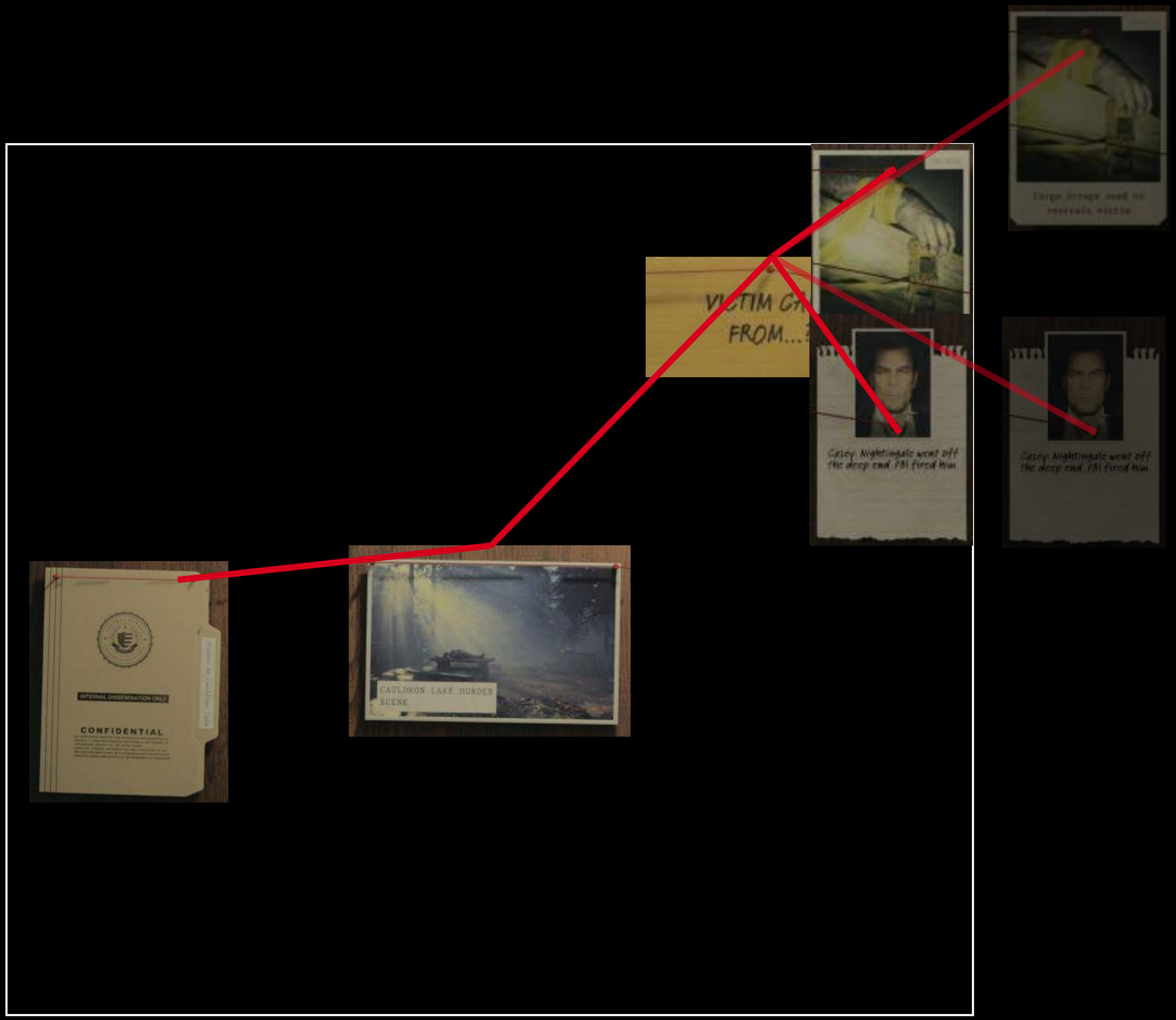
- We define a component struct that will store an item id and an item state
- We compare a component's state with an item's state stored in an environment and modify a component if necessary
- The key access is an item's entity, so the system can be potentially **parallelized** on an entity level: for each entity, a system will run in it’s own thread.

CASE BOARD MODULE: STATES UPDATE SYSTEM

```cpp
namespace component
{
    // According to our internal guideline we put components to their own namespace
    struct Item
    {
        r::String itemId;
        ItemState state;
    }
}

using ItemEntity = ecs::Access
    ::Write< component::Item >
    ::Write< component::Visibility >
    //... some other necessary components
    ::As< ecs::Entity >;

void updateStates( ItemEntity entity, const env::CaseBoard& caseBoard )
{
    auto& itemComponent = entity.getComponent< component::Item >();
    const Item* item = findItem( itemComponent.itemId, caseBoard );
    // ...
    if( itemComponent.state.offset != item->state.offset )
    {
         itemComponent.state.offset = item->state.offset;
    }
    if( itemComponent.state.isUnlocked != item->state.isUnlocked )
    {
        auto& visibilityComponent = entity.getComponent< component::Visibility >();
        visibilityComponent.isVisible = item->state.isUnlocked;
        itemComponent.state.isUnlocked = item->state.isUnlocked;
    }
    // ...
    itemComponent.state = item.state;
}
```

CASE BOARD MODULE: UPDATE BOUNDARIES

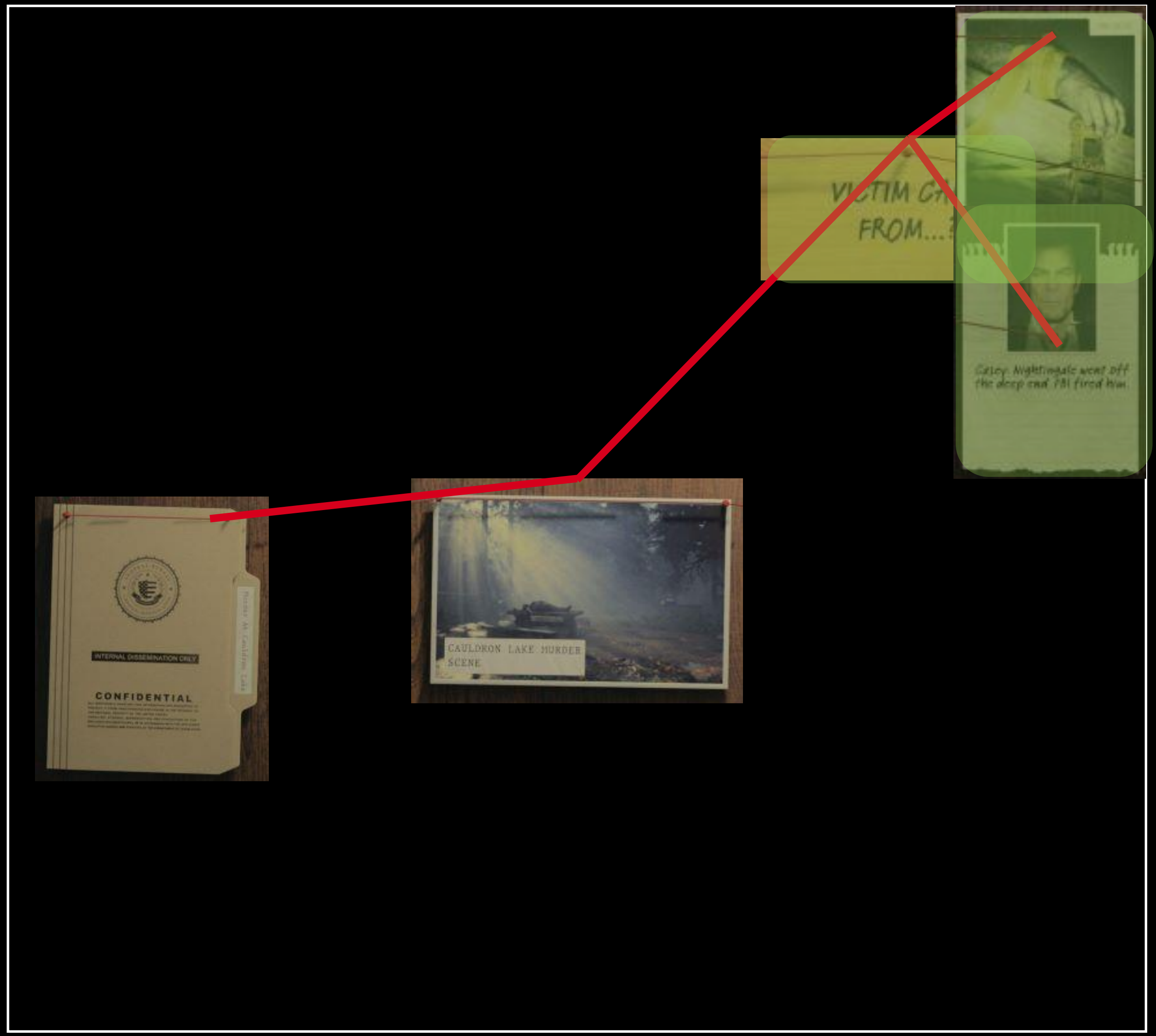
- After we updated an item’s state and read it’s offsets from data, we need to update its board relative XY position and boundaries, considering a board’s limitations.
- For storing these values, we specify a new component for item's XY boundaries
- The system will read item's state from the Item component, get extents from Mesh component and write values to ItemBoundaries component


VICTIM CA
FROM...
CONFIDENTIAL
CAULDRON LAKE MURDER SCENE
Casey Nightingale went off the deep end. FBI fired him.

CASE BOARD MODULE: UPDATE BOUNDARIES

```
namespace component
{
    struct ItemBoundaries
    {
        m::Vector2 center;
        m::Vector2 min;
        m::Vector2 max;
    }
}

using ItemBoundariesEntity = ecs::Access
    ::Read< component::Item >
    ::Read< component::Mesh >
    ::Write< component::ItemBoundaries>
    ::As< ecs::Entity >;

void updateItemBoundaries( ItemBoundariesEntity entity, const env::CaseBoard& caseBoard )
{
    auto[ itemComponent, mesh, boundariesComponent ] = entity;
    m::Vector3 itemExtents = getMeshExtents( mesh );

    boundariesComponent.center = calculateItemCenter( itemComponent, itemExtents, caseBoard );
    boundariesComponent.min = calculateItemMin( itemComponent, itemExtents, caseBoard );
    boundariesComponent.min = calculateItemMax( itemComponent, itemExtents, caseBoard );
}
```

CASE BOARD MODULE: CHECK OVERLAPS


VICTIM C
FROM...
CONFIDENTIAL
CAULDRON LAKE MURDER SCENE
Casey Nightingale went off the deep end. FBI fired him.

- After we update items’ centers and boundaries we can check for items that overlap
- This is required further for updating items’ depth value
- The key access for the system is **entity,** but we also need to **query** all other items separately
- If we want to keep parallelization on an **entity** level we need to make sure that we don't read the same component in a query access that we write in an entity access
- Therefore, it also makes sense to write overlap results in a **separate** component

CASE BOARD MODULE: CHECK OVERLAPS

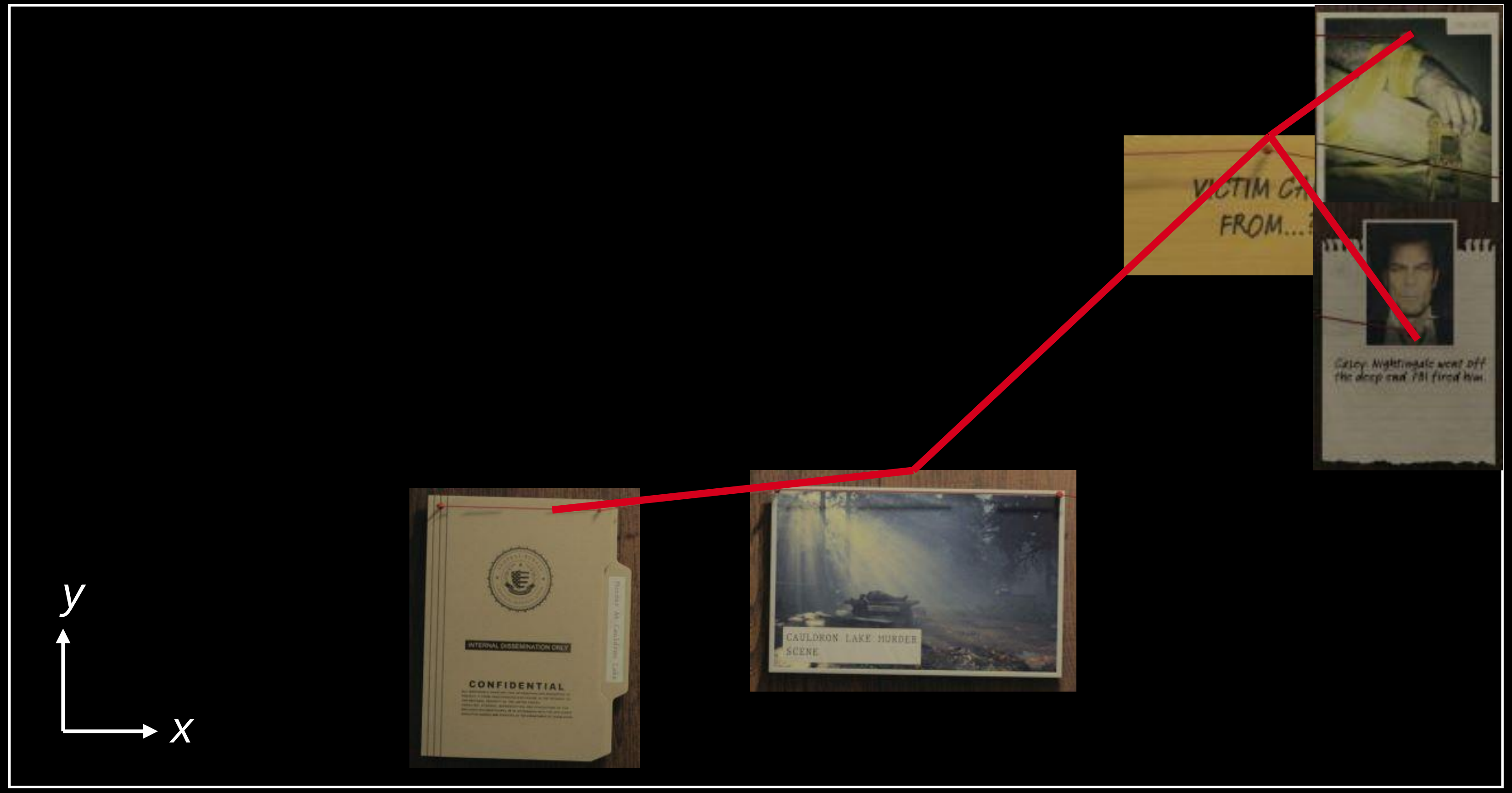
```
namespace components
{
    struct OverlappedItems
    {
        r::Vector< r::String > itemsBelow;
    }
}

using OverlappedItemsEntity = ecs::Access
    ::Read< component::Item >
    ::Read< component::ItemBoundaries >
    ::Write< component::OverlappedItems >
    ::As< ecs::Entity >

using ItemsBoundariesQuery = ecs::Access
    ::Read< component::Item >
    ::Read< component::ItemBoundaries >
    ::As< ecs::Entity >

void checkOverlaps( OverlappedItemsEntity entity, ItemsBoundariesQuery query, const env::CaseBoard& caseBoard )
{
    auto[ itemComponent, itemBoundaries, overlappedItems ] = entity;
    for( auto otherEntity : query )
    {
        auto[ otherItemComponent, otherItemBoundaries ] = otherEntity;
        if( itemComponent.id != otherItemComponent.id // Check that items are not the same
            && itemsHaveOverlap( itemBoundaries, otherItemBoundaries ) // Check that they have overlap
            && isItemBelow( otherItemComponent, itemComponent )
        {
            overlappedItems.itemsBelow.push_back( otherItemComponent.id );
        }
    }
}
```

CASE BOARD MODULE: UPDATE DEPTH

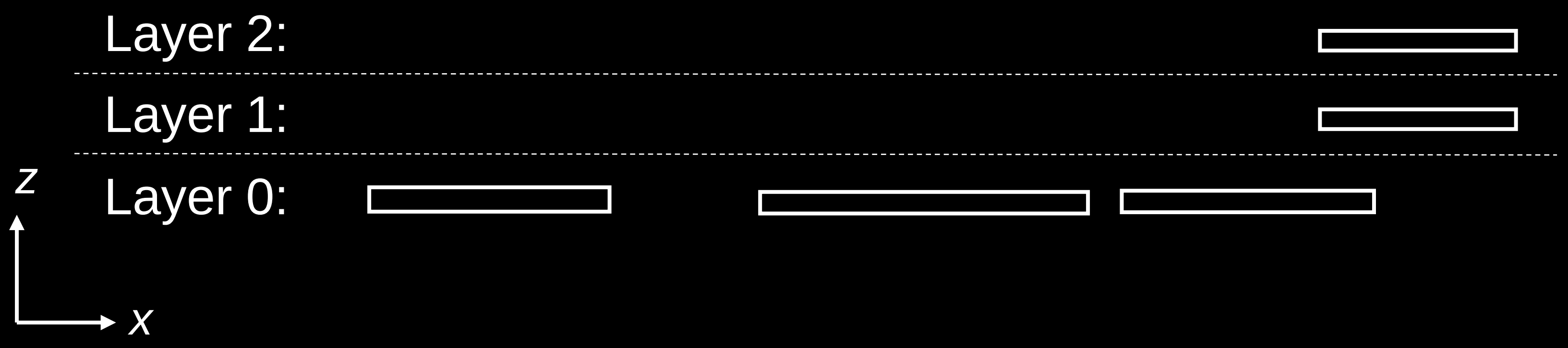
- After we have all overlaps and collected all items below each other, we can order them by Z-coordinate

- To do this, we create “layers” depending on how much an item has other items underneath that

- For creating these “layers” we need to have both read and write access for all items to resolve all dependencies.



y
x
VICTIM G
FROM...
CONFIDENTIAL
CAULDRON LAKE MURDER SCENE
Casey: Nightingale went off the deep end. FBI fired him.

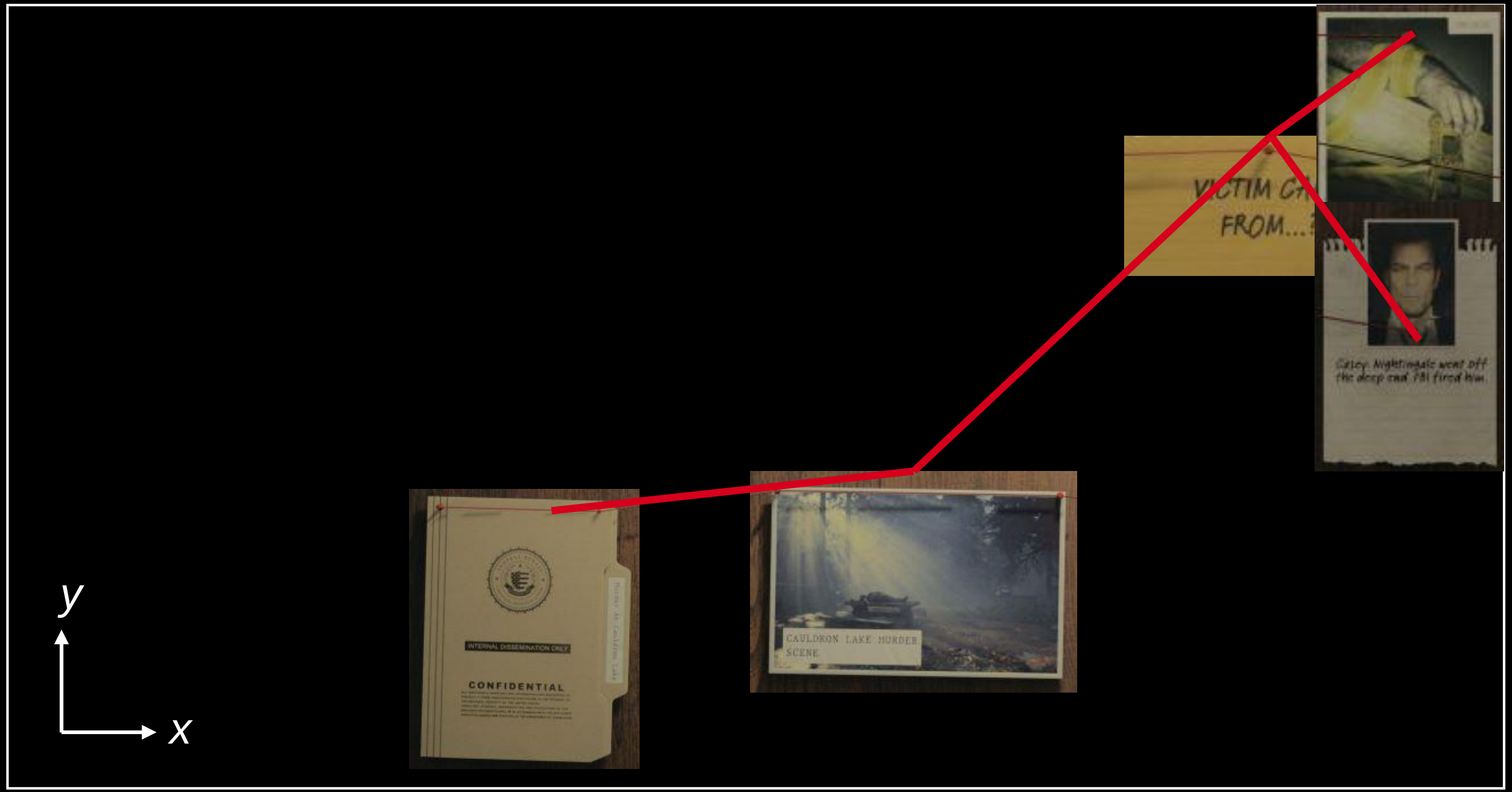


CASE BOARD MODULE: UPDATE DEPTH

```
namespace component
{
    struct ItemDepth
    {
        float value{ 0.0f };
    }
}

using UpdateItemsDepthQuery = ecs::Access
    ::Read< component::Item >
    ::Read< component::OverlappedItems >
    ::Write< component::ItemDepth >
    ::As< ecs::Query >;

void updateItemsDepth( UpdateItemsDepthQuery query,
                       env::CaseBoard& caseBoard )
{
    r::Vector< ecs::EntityHandle > layeredEntityHandles;
    // Fill the list of entities that are not on the first layer
    // and set initial depth position
    for( auto entity : query )
    {
        const auto& overlappedItems = entity
                      .getComponent< component::OverlappedItems >();

        auto& itemDepth = entity
                      .getComponent< component::ItemDepth >();

        itemDepth.value = 0.0f;
        if( !overlappedItems.itemsBelow.empty() )
        {
            layeredEntityHandles.push_back( entity.getHandle() )
        }
    }
    // ….
```

```
    // ….

    constexpr float layerThickness = 0.0005f;
    float currentLayerDepth{ layerThickness };
    while( !layeredEntityHandles.empty() )
    {
        auto it = layeredEntityHandles.begin();
        while( it != layeredEntityHandles.end() )
        {
            auto handle = *it;
            auto entity = ecs::getEntityFromQueryByHandle( query, handle )
            bool hasDependencies = false;

            const auto& overlappedItems = entity
                      .getComponent< component::OverlappedItems >();
            for( const r::String id : overlappedItems.itemsBelow )
            {
                hasDependencies |= isTheItemInTheList( id,
                                                  layeredEntityHandles,
                                                  query,
                                                 caseBoard );
            }

            if( hasDependencies )
            {
                ++it;
            }
            else
            {
                it = r::erase( layeredEntityHandles, handle );
                auto& itemDepth = entity
                                .getComponent< component::ItemDepth >();
                itemDepth.value = currentDepth;
            }
        }
        currentLayerDepth += layerThickness;
    }
}
```

CASE BOARD MODULE: UPDATE TRANSFORMS


VICTIM C
FROM...
Casey Nightingale went off the deep end. FBI fired him.
CAULDRON LAKE MURDER SCENE
CONFIDENTIAL
y
x

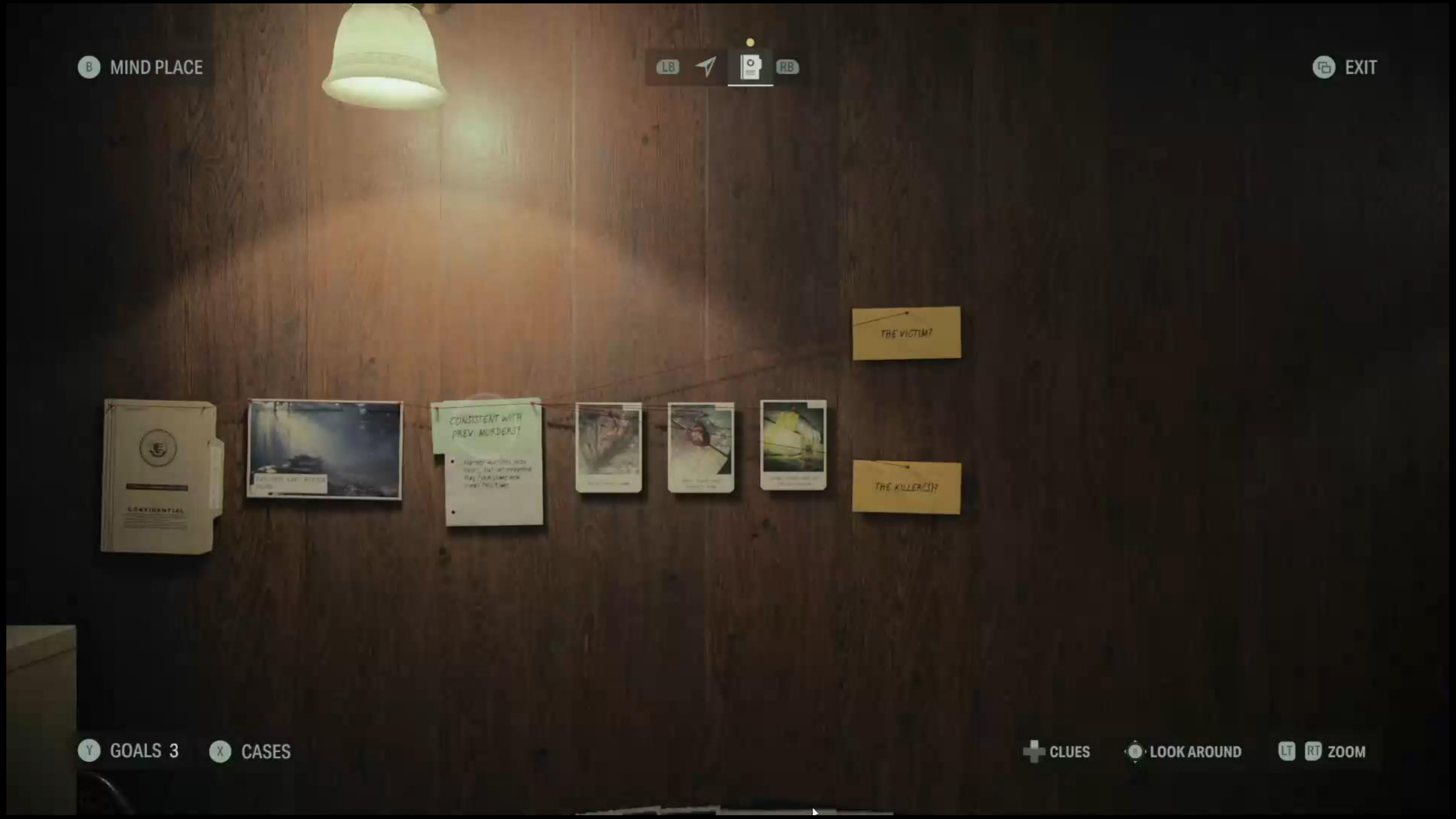
- Finally, after we calculated XY and Z values for an item we actually set a new transform
- Bringing this logic to a separate system allows us to "lock" transform components for writing only when we actually need them and don't block other systems from working with them if they can do it earlier in the frame

CASE BOARD MODULE: UPDATE TRANSFORMS

```
using ItemTransformsEntity = ecs::Access
    ::Read< component::Item >
    ::Read< component::ItemBoundaries >
    ::Read< component::ItemDepth >
    ::Write< component::WorldTransform >
    ::As< ecs::Entity >

void updateItemTransforms( ItemTransformsEntity entity, const env::CaseBoard& caseBoard, ecs::CommandBuffer ecb )
{
    auto[ itemComponent, boundaries, depth, worldTransformComponent ] = entity;
    // We take item boundaries center
    m::Vector3 translation( boundaries.center.x, boundaries.center.y, depth );

    // Item board-relative rotation is set by a configuration
    m::Quaternion rotation = getItemRotation( itemComponent );

    m::Transform boardRelativeTransform( translation, rotation );
    m::Transform worldTransform = caseBoard.boardTransform * boardRealtiveTransform;

    if( canAnimateItem( itemComponent ) )
    {
        motion::animateWorldTransform( entity, caseBoard.itemAnimationSpring, worldTransform, ecb );
    }
    else
    {
        coregame::setWorldTransform( entity, worldTransform );
    }
}
```

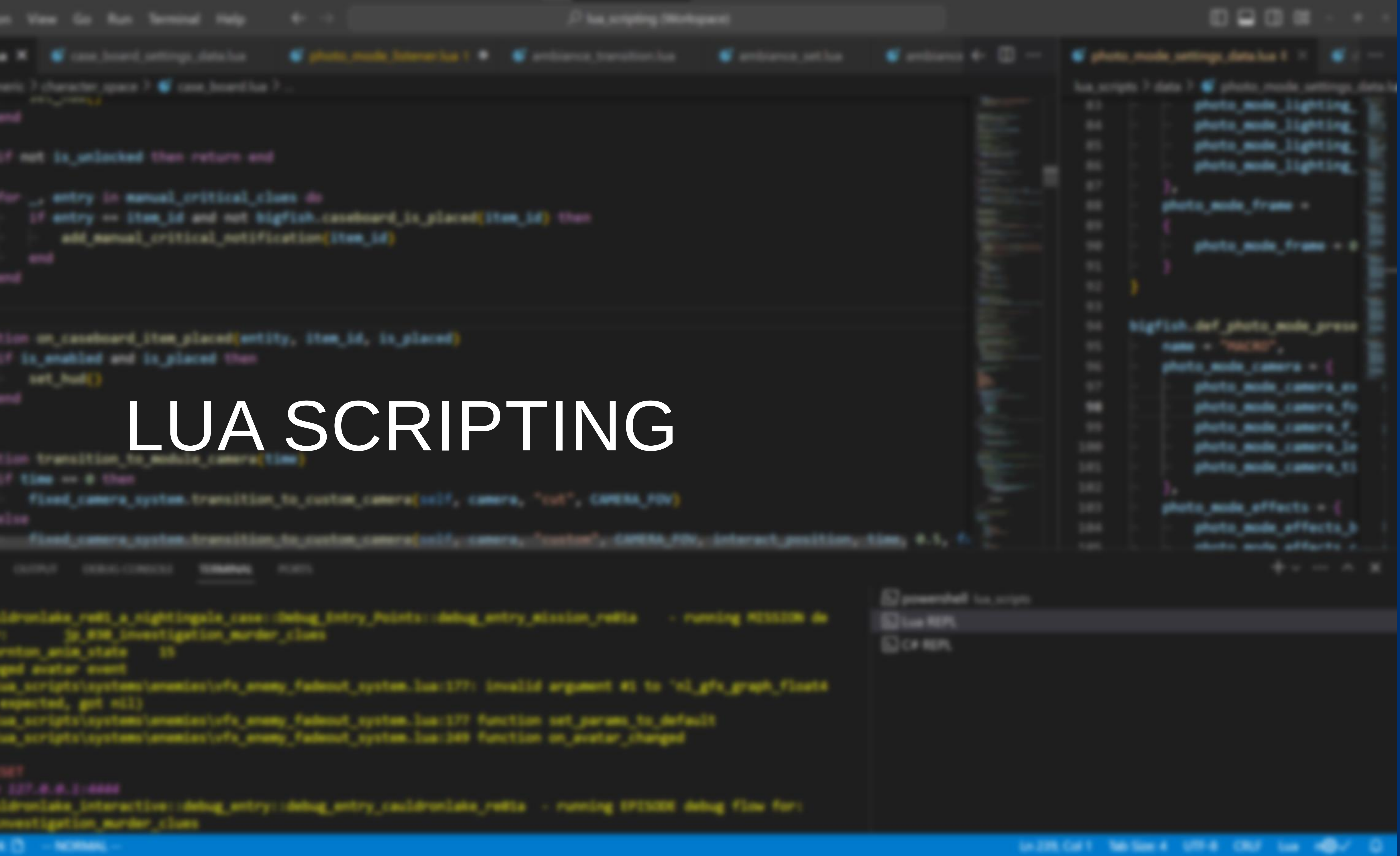
CASE BOARD MODULE: ANIMATING ITEMS

- Animation (spring motion) for entities is handled with a different system which can be used for other gameplay features
- For adding animation to an item we just need to add a motion **component**
- When an animation is over, we remove the **component**
- We're using a specific helper object - command buffer for registration of components layout, which is usually flushed at the end of the frame



CASE BOARD MODULE: ANIMATING ITEMS

```
namespace component
{
    struct SpringMotion
    {
        m::Transform targetTransform;
        SpringParameters springParameters;
    }
}

namespace motion
{
    using TransformEntity = ecs::Access
        ::Read< component::WorldTransform >
        ::As< ecs::Entity >

    void animateWorldTransform( TransformEntity entity, m::Transform targetWorldTransform, SpringParameters springParameters, ecs::CommandBuffer ecb )
    {
        auto& transformComponent = entity.getComponent< component::WorldTransform >();
        if( m::Transform::areNotEqual( transformComponent.value, targetWorldTransform ) )
        {
            // After adding SpringMotion component an entity will be processed by a motion system
            ecb.addComponent( entity, component::SpringMotion{ targetWorldTransform, springParameters } );
        }
    }

    using MotionEntity = ecs::Access
        ::Write< component::WorldTransform >
        ::Read< component::SpringMotion >
        ::As< ecs::Entity >

    void system( MotionEntity entity, const env::FixedTime& fixedTime, ecs::CommandBuffer ecb )
    {
        auto[ transform, spring ] = entity;
        m::Transform newTransform = calculateSpringMotion( transform, spring, fixedTime.deltaTime );
        coregame::setWorldTransform( entity, worldTransform );
        if( m::Transform::isEqual( newTransform, spring.targetTransform ) )
        {
            // After removing SpringMotion component an entity won't be processed by a system again
            ecb.removeComponent< component::SpringMotion >( entity );
        }
    }
}
```



LUA SCRIPTING

LUA SCRIPTING

- Some case board functionality should be exposed to game designers in a scripting system

- We use Lua as a scripting language

- We do not reflect data on the Lua side. Designers don't have to think about such categories as components, access definitions, etc. Instead, they operate with functions, passing entity IDs as arguments.

- There are 2 ways we can interact with the Lua side
  - C++ Lua bindings
  - Lua events

- Our Lua scripts run a single thread sequentially. When they are executed all components state is synced and none of the other systems run. Therefore, designers also don't have to worry about multithreading issues.

SCRIPTING: LUA BINDINGS

- Lua bindings are C++ functions organized into Lua modules which allow Lua scripts to call C++ code.

- Each Lua binding consists of three parts:
  1. A binding function that transfers data between C++ and Lua VM and checks the incoming arguments for errors
  2. Doc comments used for generating Confluence documentation and showing function info in VS Code (doc comments are described later)
  3. Type definition used by Luau type checker to check for errors in scripts (described later). The type definition information is automatically generated from doc comments.

- The general structure of a binding function is:
  - Get the parameters from Lua parameter stack.
  - Check the parameters for invalid types (e.g. function expects a string, but is given a number) and values (e.g. out of range values) and raise Lua errors if necessary.
  - Call a C++ function or class method that does the actual work.
  - Push return values to Lua stack and return the number of values. Lua supports multiple return values.

SCRIPTING: LUA BINDINGS

Lua

```lua
function place_item_if_not_placed(item_id, other_item_id)

    if not caseboard_is_item_placed(other_item_id) then

        caseboard_place_item(item_id)

    end

End`
```

C++

```cpp
/// caseboard_is_item_placed(item_id: string) -> boolean
/// @param item_id an id of an item for checking placement
/// Returns true if item is placed, false otherwise
int caseboard_is_item_placed( lua_State *L )
{
    // Get an access to all ecs component and environments
    ecs::SyncedWorld& world = getECSWorld( L );
    auto& caseBoard = world.getEnvironment< env::CaseBoard >();
    r::String itemId = lua::checkString( L, 1 );

    caseboard::Item* item = caseboard::findItem( itemId, caseBoard );
    lua::pushBool( L, item->state.isPlaced );
    return 1;
}

/// caseboard_place_item(item_id: string)
/// @param item_id an id of an item for checking placement
/// Returns true if item is placed, false otherwise
int caseboard_place_item( lua_State *L )
{
    // Get an access to all ecs component and environments
    ecs::SyncedWorld& world = getECSWorld( L );

    auto& caseBoard = world.getEnvironment< env::CaseBoard >();
    r::String itemId = lua::checkString( L, 1 );

    caseboard::placeItem( itemId, caseBoard );
    return 0;
}
```

SCIRIPTING: LUA EVENTS

- Lua events is the way to communicate from C++ to Lua scripts
- We use environment LuaEvents to collect events during the frame
- The collected events will be dispatched at the beginning of the next frame
- Every system that sends a Lua event becomes single-threaded since we need to have write access to LuaEvents environment

SCIRIPTING: LUA EVENTS

Lua

```lua
function on_caseboard_item_placed(item_id, is_placed)
    print("Item " .. item_id .. " is placed: " .. tostring(is_placed))
end

function init()
    add_global_event_handler("caseboard_item_placed", on_caseboard_item_placed)
end
```

C++

```cpp
void updateStates( ItemEntity entity, const env::CaseBoard& caseBoard, env::LuaEvents& luaEvents )
{
    auto& itemComponent = entity.getComponent< component::Item >();
    Item* item = findItem( itemComponent.itemId, caseBoard );
    // ...
    if( itemComponent.state.isPlaced != item.state.isPlaced )
    {
        // LuaEvents::CaseBoardItemPlaced - enum with an event number
        lua_events::broadcast( luaEvents, LuaEvents::CaseBoardItemPlaced, itemComponent.itemId, item.state.isPlaced );
    }
    // ...
}
```

MARCH 18–22, 2024 #GDC2024

CONCLUSION

– With ECS framework, the gameplay programmer's mindset is changing. Instead of working with objects, we work with the data

– Overall programming process starts to look like working with a database, which makes sense since we query and modify data (my first personal association is LINQ)

– Having the availability to add and remove components dynamically in runtime we can turn easily on and off systems for various entities

– Systems registration allows us to explicitly specify an order of the systems if they share access to the same data

– For achieving better performance it's better to split data between multiple components instead of keeping it in one.

– We can make a convenient and robust scripting interaction (e.g. Lua) without requiring designers to take care of data layouts and parallelism

THANKS

- Narrative team: Simon Wasselin, Molly Maloney

- UI Team: Riho Kroll, Matthew Jennings, Krzysztof Statkiewicz, Mirko Aus

- Programming team: Antti Kerminen, Antti Karkinen, Dustin Meijer, Timo Sihvo, Simon Kompaeur (Mi Pu Mi), Martin Kernjak (Mi Pu Mi), Sami Hakkarainen, Sebastian Eriksson, Petr Polezhaev

- Management: Mika Vehkala, Olek Lebiedowicz, Katriina Vakkila

And everyone else at the Remedy Entertainment Plc. especially Alan Wake 2 team

THANK YOU!

Q&A