



# Housemarque at GDC!

## Monday

The Narrative Innovation Showcase 2022

**Eevi Korhonen**

Can We Do It with Particles?: VFX Learnings from 'Returnal'

**Sharman Jagadeesan**

**Risto Jankkila**

## Thursday

Breaking the Cycle: The Making of 'Returnal'

**Harry Krueger**

RETURNAL



So first of all, thank you all for coming to my NFT talk. I know this is a small and unknown field, and it means so much to me that you're all here.

Naaaah, I can't do that to you guys. It's the end of the week, the first GDC after the pandemic hit. Y'all enjoy it?

I'm hoping that we can wrap the week up with a bang for everyone here.

We at Housemarque have been here all week. You might have already been to some of these sessions in fact, and if not - check them out on the vault after.

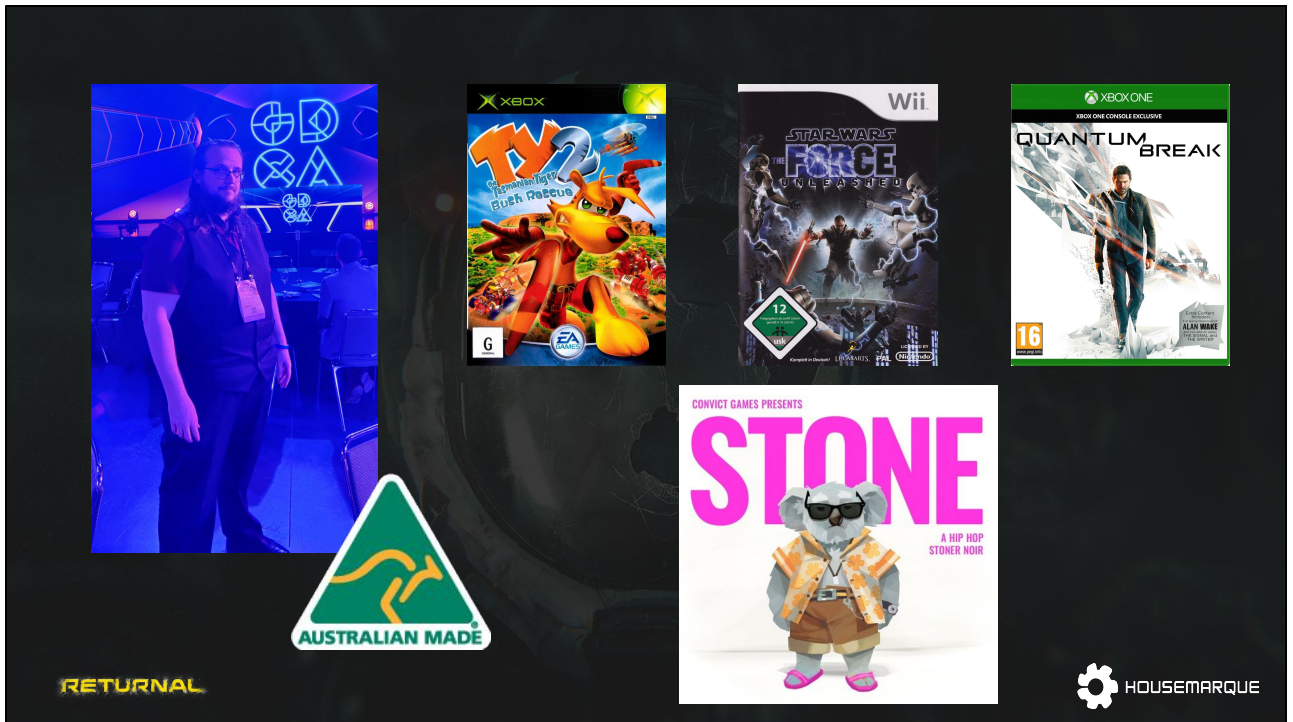
Bit of housekeeping too - the usual stuff, phones on silent.

Let's get on with it! [FORWARD]



This talk today is on how we handle the procedural world layout in Returnal. While it is ostensibly a programming talk, there was a desire to have it work for the visual arts track as well.

As such, it's going to favour explanations that actual humans can understand for the most part rather than hardcore details. [FORWARD]



A bit about myself first. So my name is Ethan Watson. And I'm living in Finland and working at a little studio called Housemarque. [FORWARD]

I'm a polyamorous bipolar Australian by day, [FORWARD]

and a 19-year veteran of the industry by night. And I mean by night, since the Geneva convention states that game developers must work crunch at all times.

I've been an engine programmer for most of my career [FORWARD]

And I'm even an award-nominated voice actor these days.

This little game here got me a nomination in the NAVGTR awards for comedy lead. My competition there was John DiMaggio for his role of Jake the Dog in an Adventure Time game.

Guess who won that one.

Come on, Bender, just let me have one freaking award will ya? How many shelves can you even fit in your home, huh? Bite your own shiny metal [FORWARD]





Aaaaanyway. We are, of course, here to talk about Returnal today, which is a small indie game you might have heard of.

Returnal is a pioneer of the emerging third-person tentacle simulator genre. It is an action game first and foremost, with heavy psychological elements, hard difficulty, and a rogue-lite structure where death is an integral part of the game.

While there is plenty to talk about, in particular we're going to talk about how we handled the ever-changing world layout in the game today.

Returnal is rather unique in the big-budget console gaming space in that we don't have pre-defined world layouts. And that's what we're here to talk about today.  
[FORWARD]

# Making This Was A Team Effort



RETURNAL



I'm also not here presenting purely my own work. Many of the core decisions for how the procedural layout would be handled were made before I joined Housemarque in 2018

Former and current Housemarque employees that worked on these systems include

- Toni Aaltonen
- Kimmo Ala-Ojala
- Ari Arnbjornson
- Lauri Mäkinen
- Henri Mustonen
- Joonas Nätyнки
- Henri Reunanen
- Jere Sanisalo
- Matias Sundberg
- Paul Wagner
- And myself

We also had help from our our partner studios Climax and Whitemoon Dreams to get not just the technical implementation done but also the design implementation you see in the game today. This was in every way a collaboration between tech and design, and there's no way we'd have done this without everyone bringing their own specialties and perspectives to the table.

We didn't start from scratch though. We'd already had some experience with

procedural world layouts in one of our prior games. [FORWARD]

## Alienation - And Beyond

- \* Procedural layout used in select locations
- \* Gave confidence to apply to a full game



RETURNAL



In 2016 Housemarque released Alienation for the Playstation 4.

It was a top-down twin stick shooter, and we used procedural layouts in select locations in the game - this one on display is the Battleship section. We did find this to be quite successful.

But go big or go home, right? In keeping with the roguelike genre that we were aiming for, the vision for Returnal was that the entire game's world layout would be procedurally generated.

Which meant we needed to expand the system's capabilities.

We did prototype some different ways to expand the generator, but that's probably worth a whole talk in itself.

[FORWARDx2]

# The Returnal Approach

- \* Fairly freeform
  - No grid, layout as you please
- \* Arbitrarily sized rooms
  - Enclosing volume hand-placed
- \* Designated safe connection points

RETURNAL



After some experimentation, we settled on [FORWARD]

a very freeform approach to procedural layouts. In particular, [FORWARD]

a grid based approach was ruled out as being too restrictive for the artistic vision. Designers wanted no restrictions, so they got no restrictions. This would prove to be both a good and a bad thing. Foreshadowing! The lack of restrictions [FORWARD]

also applied the rooms themselves. There was no limit placed on room dimensions in the slightest. The art and gameplay teams were free to go ahead and do whatever they wanted. There were only two things we actually required of them: [FORWARD]

We needed our team to place down by-hand a concave volume that would enclose the entirety of the room in question. This would be used for all kinds of tests, such as working out whether a lockdown and miniboss enemy should be spawned for example, but it was also critical to the generator itself. We also required [FORWARD]

designated safe connection points so that we could connect everything together with our generator. [FORWARD]



# Get In To The Game In Five Seconds

- \* Load in designated “boot layer” sublevels to start room
  - Let player run around ASAP
- \* Generate layout
- \* Load in LOD representation of rooms
- \* Load area immediately surrounding player
- \* Generate terrain
- \* Generate “trees”

RETURNAL



We were also targeting a Playstation 5, with its fancy-pants new SSD that - and this is my view here, the SSD *is* the next generation tech of the current consoles. Ray tracing, eh, whatever, that doesn't completely redefine what we're able to do from a design perspective. The SSD is a literal game changer, and we're just scratching the surface of what's possible.

Still, to get in to the game in five seconds we needed to structure things carefully.  
[FORWARD]

The first thing in our boot sequence is to load designated “boot layer” sublevels. These are select chunks of geometry that allow things like cutscenes to play,  
[FORWARD]

and even allow the player to run around the environment as quickly as possible. And that's actually all you need to get in to the game in five seconds, but we have loftier goals. [FORWARD]

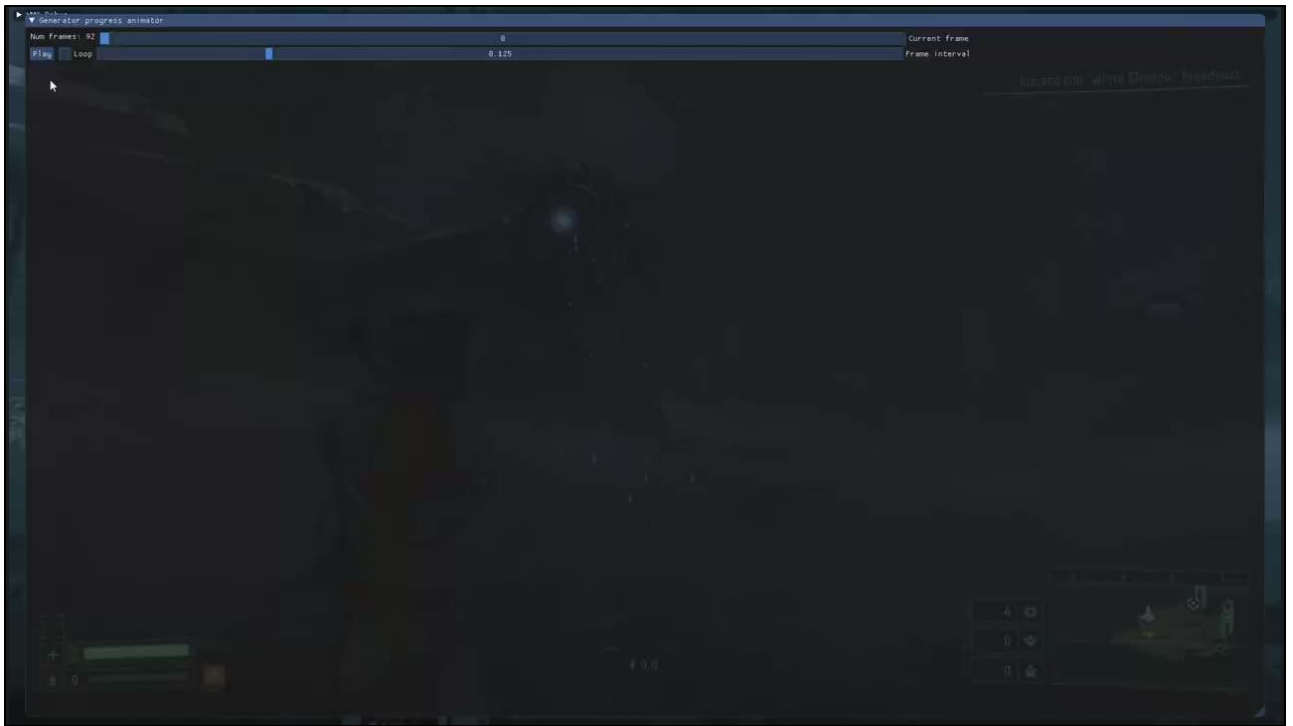
Once those boot layers are in, only then do we generate a layout. That's the advantage of starting in the same place in every biome, we can generate once the game is already playable. And once that's generated, [FORWARD]

we turn on the level streaming system. It starts off by loading in all the low detail representations of each room, [FORWARD]

before we start loading in all the high detail meshes and gameplay logic for every room around the player's position. But even that's not quite enough. [FORWARD]

We actually need to do a terrain generation [FORWARD]

and populate it with vegetation to sell the world to you. [FORWARD]



It's quite unusual for a AAA game to handle procedural world layouts.

There are some that do it in a limited manner, and usually as a side feature to the main event.

To base the entire game around the concept is something that was quite common in the early days of our industry - Pitfall might be the earliest example I can think of, moving on to games like The Sentinel in the 8-bit home computer era and even Worms and Diablo as the years progressed.

Procedural layouts for an entire game is essentially relegated to small indie games these days however as a way to reduce the workload required in generating hours of unique content with a small team.

But there you have it, as you have seen with this video we made procedural world layouts work - and won plenty of accolades and awards as a result.

So for the rest of this talk, I'm going to take you through that five seconds described in the previous slide. [FORWARD]



[ TURN ON Q&A FEATURE ]

Y'all ready?

Cool, let's get on with it. [FORWARDx2]



We're going to get right into the meat of it all first, since everything hinges on knowing how the generator works.



# What Is A Room?

- \* In UE terms
  - A level and associated sublevels
- \* In our terms
  - A volume defining extents
  - Minimum of one connecting point
    - Two for corridors, three minimum for combat rooms
  - Various locators for treasure, spawn points, etc
  - Actor to provide metadata to the generator

RETURNAL



First things first, the basics. If you're doing a procedural game, you'll likely have rooms to simplify the task. But what is a room? [FORWARD]

As far as Unreal Engine is concerned, [FORWARD]

it's just a level and associated sublevels. [FORWARD]

But we require a few more things from it to make the system work. [FORWARD]

I've already mentioned the volume enclosing the room, [FORWARD]

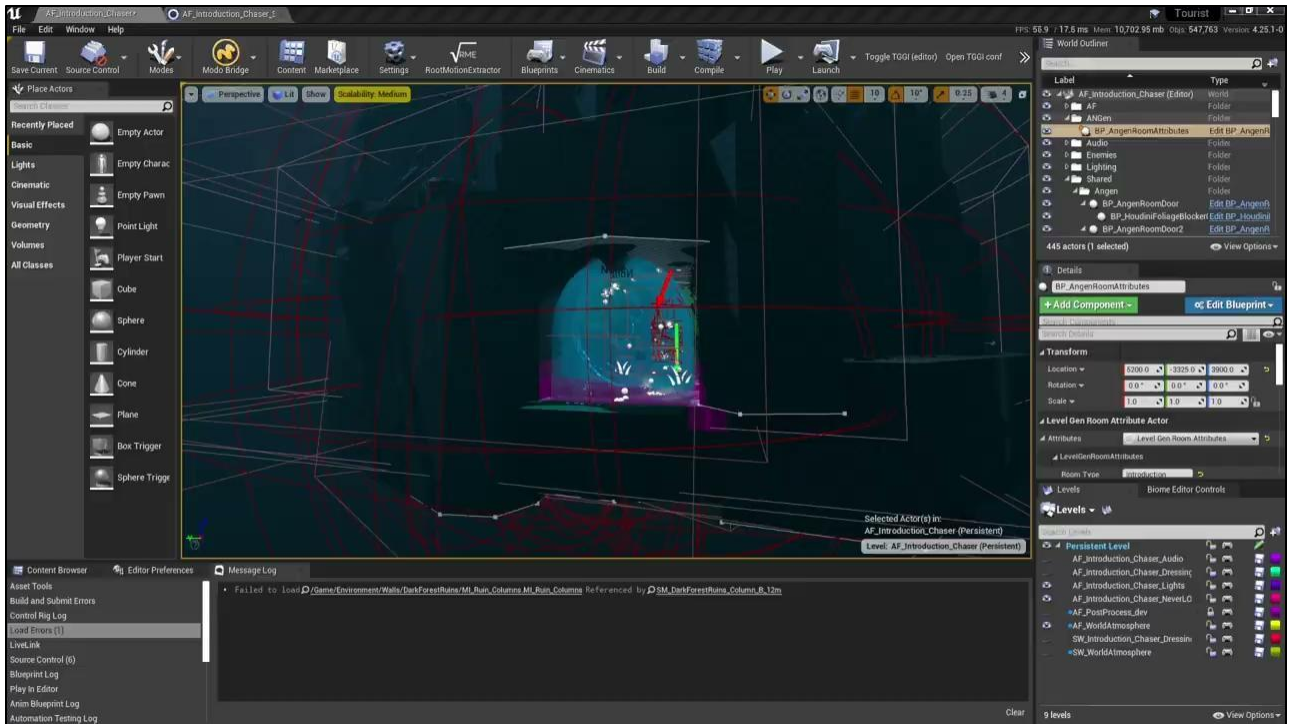
and in terms of the connections we require just one connecting point. On the design side, this was commonly used for starting rooms, treasure rooms, and some special rooms such as the Reconstructor rooms [FORWARD]

The design team also expanded those requirements for other room types, with corridors typically having two connections and combat rooms requiring a minimum of three connections.[FORWARD]

We also don't place much directly in the levels apart from static geometry and various triggers. Treasure, spawn points, and all sorts of things are defined by locators and given over to code and scripting systems to place after generation time. [FORWARD]

We also require a specific actor to be placed in the level, in order to extract data for

the generator to work from. Just a simple little struct that gets saved out to another file. [FORWARD]



And as you can see, there's really not much special here

That blue blob in the centre is one of the connecting points, but otherwise everything you'd normally expect to find in your UE Editor is there and waiting for you.

This particular level is one of our introductory levels, the one where you first encounter Kerberons and find a key.

We'll keep coming back to this particular room throughout the talk, so there's no need to make too much sense of it yet.

The flyby will show you various things such as the spawnpoints and locators I just mentioned. [FORWARDx2]

# You Have Rooms. Now What?

- \* Level generator
- \* Functionality written in C++
- \* Logic implemented in blueprints
  - DSL considered, but designers wanted blueprints

RETURNAL



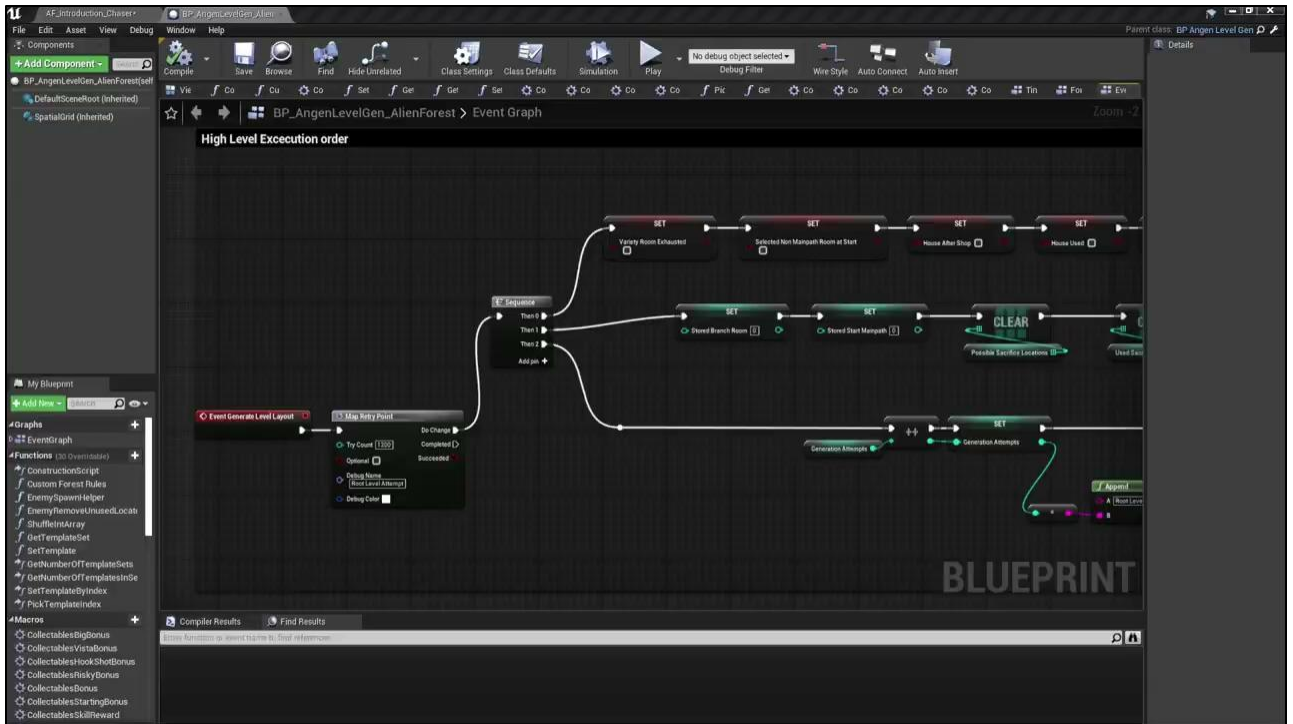
So. What's next? [FORWARD]

Well, you need a level generator to lay out all your rooms into a cohesive gameplay experience. [FORWARD]

A good chunk of this system is written in C++, but [FORWARD]

with plenty of hooks that people using the blueprint visual scripting toolset can hook in to in order to define custom logic per-biome. [FORWARD]

A scripting language was considered for this, but our designers weren't comfortable with writing script. As such, the decision was made to do this all with in blueprints. [FORWARD]



Which, yeah, our designers were quite adept in.

And as you can see here. They even wanted to make things look neat to help with maintenance.

This looks quite manageable compared to some blueprint examples you've seen over the years, yeah?

Well. As we scroll out a little... Yeah. You can see the magnitude of the logic we're dealing with here.

The designers were cool with it, but I just want you to keep in mind this little glimpse of the complexity we're looking at as it will be very relevant towards the end of this talk. [FORWARDx2]



# Determinism

- \* Using the same inputs, get the same outputs every time
- \* Layout generator designed around this idea
- \* Unreal Engine, well, isn't deterministic
  - Required writing our own blueprint functionality for things like random number generators to ensure determinism

RETURNAL



Now, how many people in this room know what determinism is? [FORWARD]

Quite simply, it's the idea that given the same inputs you'll get the same outputs every time. [FORWARD]

In our terms, this means the same basic set of data we give the generator will result in the same world layout every single time. One problem, though: [FORWARD]

Unreal Engine, well, it's not deterministic. If you rely on default UE functionality, things will be *ever so slightly different* every time you play your game. [FORWARD]

This required us to write our own functionality for things like random number generators to ensure determinism. And yeah, it's important. Debugging, daily challenges, all these things work because we have a deterministic system. [FORWARD]

# Generator Setup

- \* Custom blueprint per biome
- \* Small amount of inputs
  - Random seed
  - Gameplay flags
- \* ...So not exactly 32 bits per generator, but still a very small dataset

RETURNAL



So how do we set up our generators then? [FORWARD]

To start with, there's a custom blueprint for every biome in the game.

There are six biomes in the game - oh, wait, we released a new patch, didn't we? So there's seven biomes in the game, and they have their own custom blueprint.

There's also separate blueprints for the daily challenges. [FORWARD]

And we really only deal with a very small amount of inputs to make this work:  
[FORWARD]

A 32-bit integer as a seed basis for the random number generators; and [FORWARD]

separate gameplay flags that affect how the generator logic chooses rooms. This ended up being an FName array, [FORWARD]

So it's not exactly 32 bits per generator. We could have fit the flags in to the random seed for sure, but it was easier for the designers to work in that way so we kept it.  
[FORWARD]

# Some Gameplay Flags Examples

- **FirstPlay**
  - The introduction layout up to the first miniboss
- **ParasiteIntro**
  - Never places the parasite cutscene once set
- **HouseKeyFound**
  - Allows key room to be randomly placed

RETURNAL



A couple of examples from the overgrown ruins are: [FORWARD]

FirstPlay [FORWARD]

Which triggers a specific sub-script that always places the same layout up until you've defeated the first miniboss or it yeets you in to the next cycle. [FORWARD]

There's also the ParasiteIntro [FORWARD]

Which will always place the parasite introductory room next to the Spitmaw Blaster introductory room until you've triggered that little cutscene. [FORWARD]

And there's also HouseKeyFound [FORWARD]

Which affects the placement of the room where you find the house key. It's near the end of a layout when you haven't found it, but once you have found it the room is allowed to be randomly placed in the biome. [FORWARD]

# Layout Logic

- \* Place room
- \* Test if volume overlaps with any other room
  - No overlap? Place another room
  - Overlaps? Fail, rewind to a recovery point
- \* Repeat until
  - All required rooms placed
  - Too many failures, start again

RETURNAL



With these basic parameters in place, it's time to perform the actual layout. It's a fairly conceptually simple process. [FORWARD]

Take a room. Place it in the world. [FORWARD]

Then test if it overlaps with any other room. [FORWARD]

If it doesn't overlap, YAAAAAY then excellent. We're free to continue and place more rooms. [FORWARD]

If it does overlap though, then that's considered a failure. And what we do there is rewind the generator to a recovery point - this could be the previous room or a few rooms back in certain cases - and try again. [FORWARD]

And you keep on trying again and you keep on trying again [FORWARD]

Until the generator has laid out the required rooms for a gameplay session, or [FORWARD]

It encounters too many failures. In which case, it starts over from scratch. [FORWARD]

# Room Selection

- \* Done with custom biome logic
- \* Overgrown Ruins specialisations
  - Introductory sequence
  - Shop at halfway point

RETURNAL



How do we get a room to put down in the world though? [FORWARD]

That's actually done in the custom logic for each biome. And to give you a bit of an example, [FORWARD]

We'll take a quick look at the Overgrown Ruins biome. This is the first biome you'll experience in the game, and it's the one with the most customisation. A few examples include [FORWARD]

The introductory sequence, where all the mechanics of the game such as shooting and health pickups are introduced.

There's a custom subsection of the script I showed you that places the necessary rooms until it's done, then continues the generation on as normal. [FORWARD]

There's always a shop at the halfway point too.[FORWARD]





The custom logic knows how many rooms it wants to put in the biome and when, and will check its progress to select that shop room.

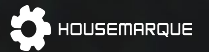
It's also a natural splitting point for some biomes, such as the Crimson Wastes where it's placed right before a teleport to another part of the map;

and the Abyssal Scar, where it's placed before you jump in to a very large hole and see the second half of the biome. [FORWARD]

## Room Selection

- \* Done with custom biome logic
- \* Overgrown Ruins specialisations
  - Introductory sequence
  - Shop at halfway point
  - House at different points depending on player progress

RETURNAL



We also have a human-style house that can change locations depending on the player's progress. [FORWARD]



It's a critical part of our story, and as such we want to ensure it will always be placed in the world every time you run the generator.

Where it's placed though can change.

You'll find it before the shop the first time you play the game. Later runs, you might find it immediately after the start room. It's all up to that custom logic. [FORWARD]

# The “Deck of Cards” Template System

- \* Template
  - Hand crafted set of rooms used in a layout
- \* Deck
  - A set of templates
- \* Take a template from the deck, create a layout
- \* Restart the cycle, take another template
- \* All templates used? Shuffle and start again

RETURNAL



It eventually transpired that we had a bit of a design problem. It was entirely possible to get a run of seeds that would result in very similar room selections. So the designers came up with a system they called a deck of cards. [FORWARD]

This meant defining a template [FORWARD]

which is essentially just a hand-crafted set of rooms to be used in a layout. Rather than a huge master list, it's just a large set of smaller lists. [FORWARD]

The deck of cards itself [FORWARD]

is constructed with all the templates for a biome. When the generator tries to create a layout, [FORWARD]

a template is removed from the deck and those sets of rooms is used to define that layout. [FORWARD]

Restating a cycle results in another template being removed from the deck and using that one instead. [FORWARD]

And if your deck is empty? Simple. Shuffle them all together and grab a template from there.

This required virtually no code support. Thanks to having logic separated from the

technical implementation, designers were free to come up with these kinds of solutions themselves.

[QUESTION TIME]

[FORWARD]





Now. I have a question for everyone here. Can I interest you in some [FORWARD]





Doors?! Yes, doors! Here at Housemarque, we're specialists when it comes to doors in your video games.

You might think you've seen doors before, but with Returnal we've upped the door game for the entire industry! [FORWARD]



So what we have here is your standard issue, unlocked, alien-origin door. Aaaah, boring. Everyone has one of those. But do you have [FORWARD]



A doorless door? [FORWARD]

How good is that, you don't even have to open the thing! Convenience for everyone.

But what if you want to have a doorway that goes to nowhere? [FORWARD]



No problem! Why not try out a rubble door! [FORWARD]

Completely non-traversable, and the ultimate in home security. After all, how can anyone break in if you can't get through the door yourself? [FORWARD]



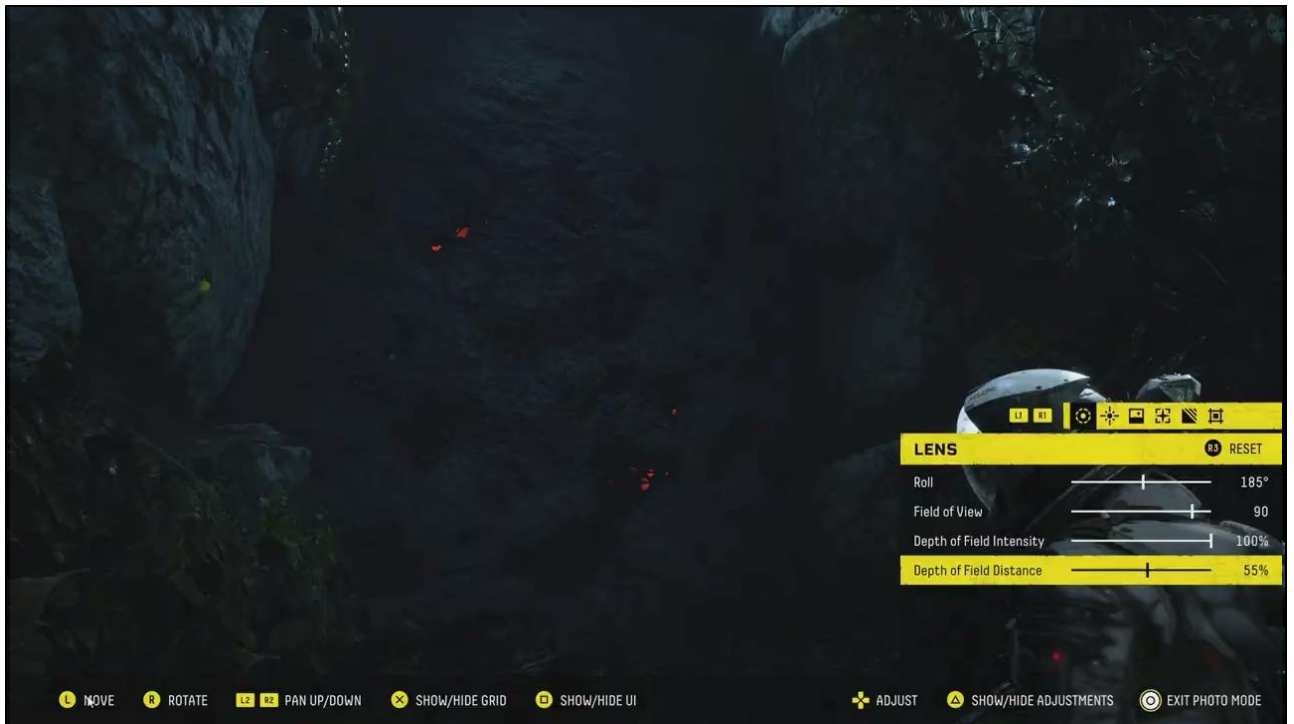


But what if you find yourself with an area that you want to be inaccessible at some times, but allow break ins at others? Oh, do we have a solution for that too.  
[FORWARD]

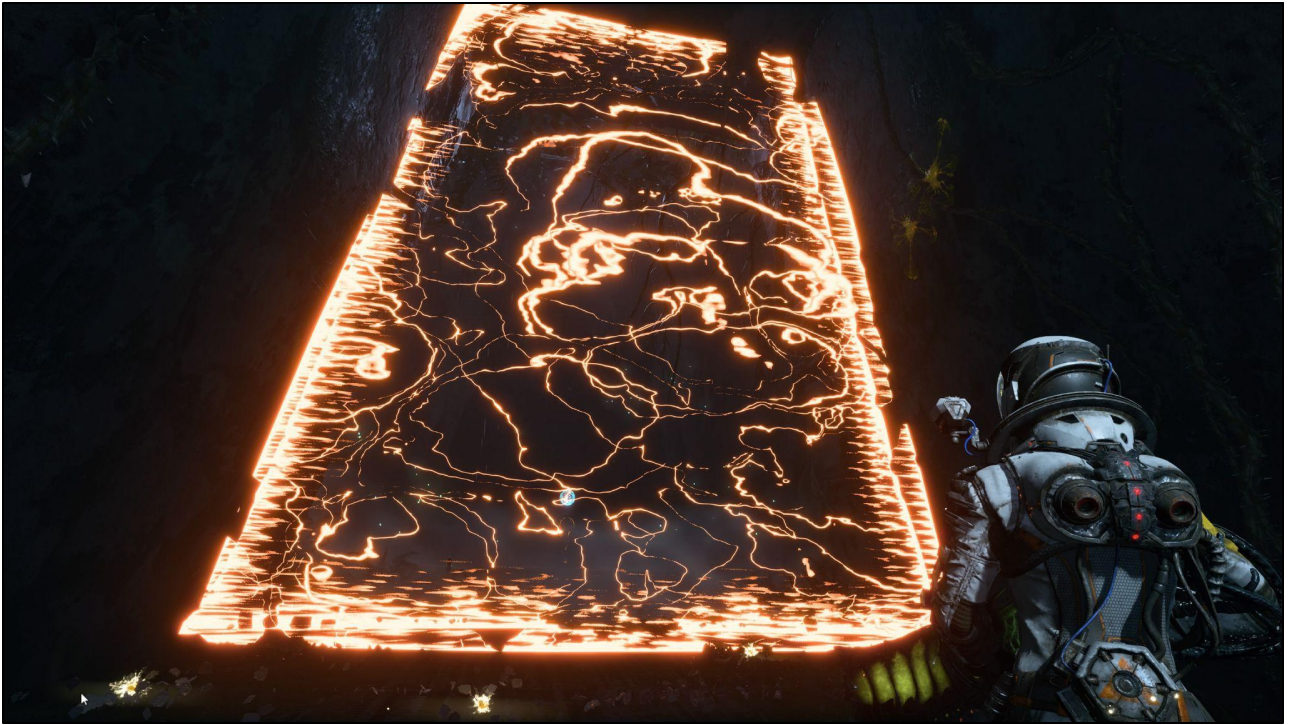


Try out the destructible door! Same physical location, different time. [FORWARD]





And with a bit of the ol' time-tested brute force, you can get through it with ease and at your own leisure. [FORWARDx2]



Maybe you're heavily in to your tech and prefer a more electronic means of having a door.

With this barrier door, you'll never be accused of being a luddite. [FORWARD]



But maybe you're incredibly secretive, and want no one to know where your door is.

Not even you.

There's actually a door in this scene. Can *you* spot it?

Looking for a door, looking for a door, where could it be [FORWARD]

Ah! There it is! Perfect, no one will ever find that on their own. [FORWARD]



And finally, we have, oh, wait, that's not a door. What even is that? I worked on the game and I don't even know, can someone tell me please??? [FORWARD]

## So, Uh, Doors

- \* Classic example of “first use case named it”
- \* Actually just connection points between rooms
  - Rubble door is called **BP\_RandomBlockedDoorWall\_FakeRoomSmall**
  - Mesh surrounding connection meets conditions for overlap avoidance
  - The desert shot actually involved blending hand-crafted floors with procedural landscape

RETURNAL



So, ahem, uh, doors. [FORWARD]

I'm sure everyone here is familiar with the first use case of something naming it, regardless of how inappropriate that term ends up being. [FORWARD]

Doors are in fact just those connection points I've been talking about. [FORWARD]

The rubble door I mentioned, in fact, is just a way of blocking off a connection that goes nowhere. And just for fun, that's the name of it. Bit of a mouthful, yeah? Well it gets the job done [FORWARD]

The connection points do have some artistic restrictions. The mesh needs to be carefully crafted to avoid overlaps and weird artefacts. [FORWARD]

The invisible door in the desert shot even actually required a system to blend the room's hand-crafted floors with the procedural landscape - we'll get to the landscape later on.[FORWARD]



# Randomisation After Layout

- \* Items placed separately after world is laid out
  - Just spawning items isn't enough...
  - We need a bit more variation inside rooms than what items show up

RETURNAL



We've been talking about the layout of the world, but what about loot? Health items?  
[FORWARD]

These are in fact handled in a separate phase after the layout is finished. And just like the layout itself, it's entirely designer driven. [FORWARD]

That, however, wasn't actually enough to handle all the variation we wanted.  
[FORWARD]

We have some really interesting use cases of variation that isn't covered by spawners, and that required another solution. [FORWARD]



## Sublevel Selector

- \* Loading system calls a function for each sublevel in a room
  - Default implementation returns true
- \* Rooms implement function to specialise behavior
- \* Layout remains constant, but elements get conditionally changed

RETURNAL



And that was the sublevel selector.

For those that are unaware, a level in Unreal Engine is a collection of a base level and any number of sublevels that you can set to load and unload at your desire.

For our system, we wanted to automate this as much as possible, so [FORWARD]

Our loading system will call a user-defined function for each sublevel in any given room [FORWARD]

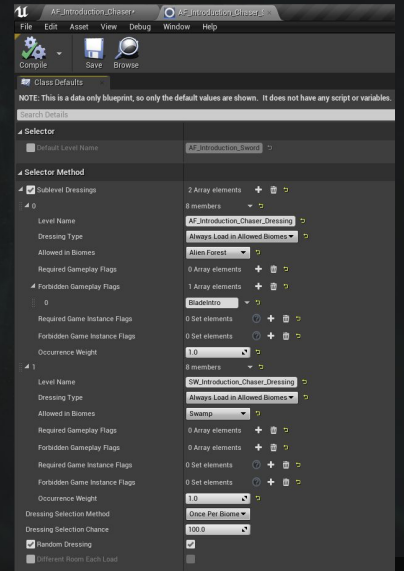
With the default implementation returning true, and thus indicating that you want to load in this particular sublevel for this session. [FORWARD]

Each room is thus free to override this function in order to specialise elements of the room. [FORWARD]

The end result is that the layout is constant and deterministic, *but* we can change up the room as much as we want without needing to expand the level generator to handle sublevels. [FORWARD]

## Sublevel selector example uses

- \* Special enemy spawning
- \* Bonus room activation
- \* Hazard layout variations
- \* Alternate takes on biome



There are a number of use cases for this system, mostly facilitated by a particular kind of sublevel selector that parameterises common functionality into something a designer can tweak and not even have to write blueprint logic for.

Things like special enemy spawning, bonus rooms, and hazard layout variations are some fairly common use cases.

But rather interestingly here, is that we also support alternate biomes in a room via this method. [FORWARD]



Take for example the previously-shown introductory room. It's where we meet our little friends, the Kerberons, for the first time in the game. Aw, it's trying to murder me with its tentacle powers, isn't that cute? [FOWARD]



And this here is the exact same room, at the halfway point in the game in a different time period and with our cute little friends the Kerberons replaced with those murder robots. Stupid murder robots...

Different environment settings, different vegetation, different geometry - all of this is handled by using sublevels and picking and choosing from the sublevels you want via a sublevel selector.

This was a very useful tool for all departments.

[QUESTION TIME]

[FORWARD]





Next, we need to talk about how we go about bringing this all in to memory.

# Loading

- \* UE4's world composition system didn't handle our dynamic layouts
  - Had to roll our own
- \* Grid based streamer
  - Room AABB placed on grid
  - Streaming volumes placed on grid
  - Collect rooms, increment/decrement reference counts
  - Pass desired loading state to UE's world streamer

RETURNAL



I've touched on the loading system before, but it might surprise you that [FORWARD]

We couldn't use UE4's world composition system. It works by dividing the world up in to tiles at design time, having levels for each tile, and manages loading that way.

This is great if you're building a static open world, but we were not. [FORWARD]

So we rolled our own loading behavior.

We didn't change UE's underlying streaming behavior, we just managed ourselves when a level would be added to the overall world for loading purposes. [FORWARD]

We achieved this with a grid-based streamer. Cell sizes were significantly smaller than most rooms, and when you placed a room in the world [FORWARD]

You also placed it on the grid via it's axis aligned bounding box. [FORWARD]

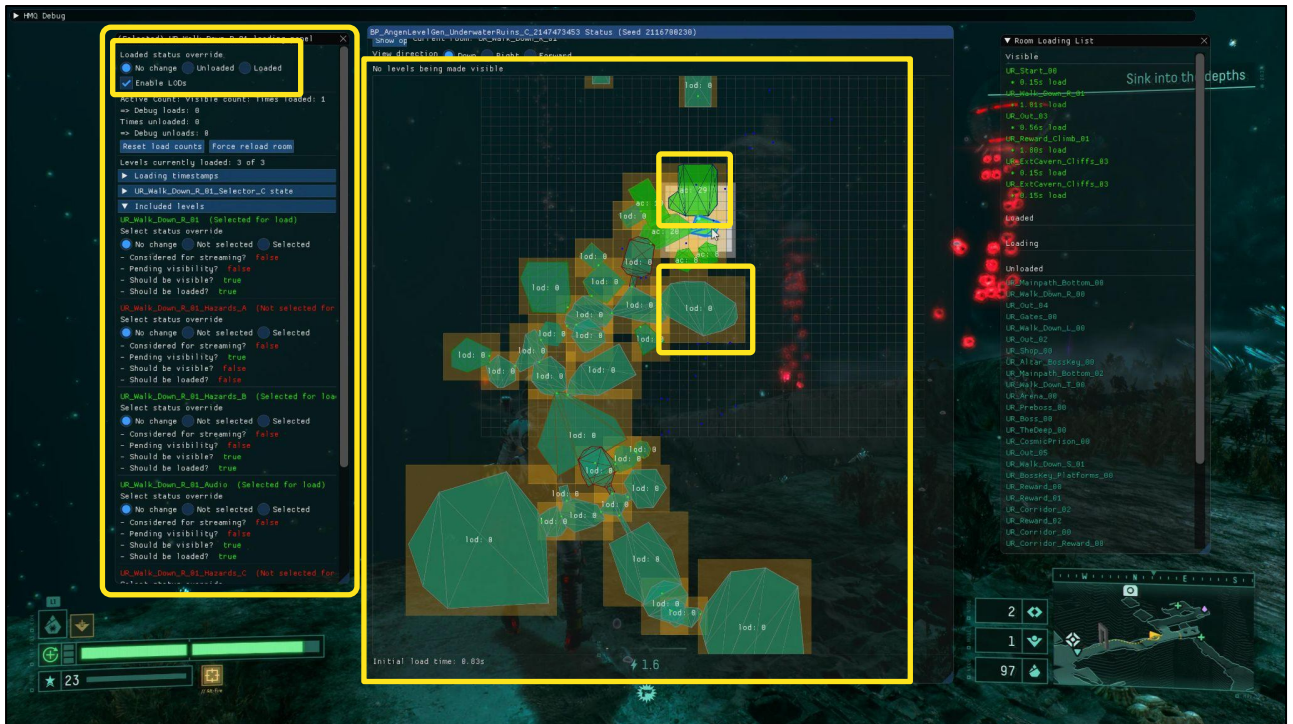
We also have multiple streaming volumes that get placed on the grid. These are used to control what gets loaded in and loaded out.

Whenever a volume's grid cells change, an event is fired stating which cells the volume is entering, followed by which cells the volume is leaving. This is just a little implementation detail, [FORWARD]



As each room actually operates on a reference counted system. Incrementing first followed by decrementing ensures we don't start unloading a level that actually still needs to be loaded in. [FORWARD]

But when something does need to load in or out, we just pass that information along to the bog standard Unreal Engine world streamer. [FORWARD]



I want to point out how incredibly important it is to have good debug interfaces for your system.

I'm a big big proponent for "information at a glance".

Many programmers know the experience of looking at a callstack or a hex value in the debugger and knowing exactly what the problem is.

But an artist or a level designer? They need to have the same specialist knowledge of a programmer to be able to do the same.

And what if you need to debug a problem without access to a code debugger? Well, you need other tools for that.

This here is the debug interface for my streamer. As some of you might spot, it is written in Dear ImGui. I almost never play the game without it open.

I put a lot of effort in to making it readable at a glance. [FORWARD]

As you might imagine, this map-looking thing is in fact the world layout.

The green objects [FORWARD]

are all the rooms that are currently loaded in to memory at full detail and everything is

interactable ready to go. [FORWARD]

The cyan-coloured rooms are all the rooms currently being represented by low-detail mesh. I don't need to look at numbers here, the colours tell me everything I need to know.

And in fact, this was more than enough information for everyone on the team to debug the streamer themselves. I successfully removed myself as a bottleneck with this debug feature.

A visual representation is really just the minimum quality bar though. There's also a number of helper panels you can open. [FORWARD]

This panel on the left, for example, doesn't just give you a detailed rundown of a room you've selected - [FORWRD]

it also lets you interact with the room and change its state. This particular set of buttons and toggles changes overrides its loading state.

One of our gameplay programmers in particular heavily used that functionality to develop and polish a few features, as it was way more reliable and quicker than running around the world hoping to trigger room loads and unloads. [FORWARD]



And here, thanks to the debug visualiser you can see the system in action. The arbitrary room volumes I've been talking about are all surrounded by yellow bounding volumes.

Whenever the white streaming volumes touch the bounding volume of a room, the reference counts get added and decremented.

The streaming volume system gave us a lot of flexibility.

One thing we learnt at Remedy on Quantum Break was that having a single streaming location just isn't enough for design intent, so from the start I decided to not repeat that mistake.

By placing additional streaming volumes down on the grid, we handle things such as loading in the target room for teleport pads.

That requires a bit of distance fiddling, since you want to unload enough around the player to free up memory and load in enough at the destination to not immediately let the player see out the world.

It also has other non-obvious uses, such as safe locations. A player in this Derelict Citadel biome can fly halfway across the map without landing.

Fall down a gap? Whoa, you're gonna have problems if your safe point is not loaded

in to memory.

Using a 1 metre cubed streaming volume on the last known safe location solves loading and respawning issues. [FORWARDx2]





# IT'S THE “ACTUAL CODE” SECTION!

RETURNAL



And oh, finally, we're getting to some code here in our programming talk! About time, huh?

There's one particular thing I wanted to talk about in detail here, since it is an emerging way of programming in C++.

# Ranges On My Spatial Grids!

- \* Programming interface for spatial grid uses ranges
  - C++20 not supported by the compilers we were using
- \* I've been using ranges for years in the D language
  - Rolled my own implementation using iterators
  - Not C++20 standard, will update to match so that we can switch over ASAP

RETURNAL



Short story: I figured that since I was writing a streaming grid from scratch that, hey, why not keep up with the cool kids and implement [FORWARD]

Ranges as a default interface for queries on the grid. [FORWARD]

Of course, C++20 was just plain not supported by the compilers we were using for the duration of our project. [FORWARD]

I am quite familiar with ranges though as I've been using them in the D programming language for years. Andrei Alexandrescu's talks about ranges in C++, if you go and look them up, are based directly on his research in that language. [FORWARD]

With high confidence in the theory behind them, I just rolled my own with iterators.

Which, yeah, it's clunky.

Iterators have to keep state to function, and once you get several ranges deep in to a statement it can get fairly heavy weight. [FORWARD]

What we need to do here is move over to C++20 as soon as possible, and not rely on non-standard implementations.

Given my history in D though, the ranges I wrote stuck to the terminology I knew rather than what is defined as standard. But that won't be a huge problem.

[FORWARD]

# Ranges - Current Usage

```
template< typename ActorType >
auto USpatialGridObject::GetActorsInCells( const FVector& Position, float
MaxDistance )
{
    return IndexRangeBox( Position, MaxDistance )
        | Map( [ this ]( int32 ThisCell ) -> auto& { return
ThisGrid->Cells[ ThisCell ].Objects; } )
        | Join
        | Map( [ this ]( USpatialGridObject* Object ) -> ActorType* {
return ( Object->ThisObject.Get( true ) != ThisObject.Get( true ) ) ? Cast<
ActorType >( Object->ThisObject.Get( true ) ) : nullptr; } )
        | Filter( []( ActorType* Actor ) { return Actor != nullptr; } )
        | Unique;
}
```

RETURNAL



Let's just take this example here, to get every unique actor of a given type from the grid.

It looks a bit messy thanks to trying to fit it on to a slide, but the code itself is actually just a bunch of really simple parts being snapped together.[FORWARD]

First, we create a range of indices for the grid. Each cell in the grid is stored in an array, so each element of this range is an integer that centres around a position with the defined radius. [FORWARD]

From there, we take that integer, look up the cell we're in, and get a reference to the grid object array it contains. [FORWARD]

We then join them together, treating the range of arrays as a single range of grid objects. [FORWARD]

And then we work to converting that grid object to the actor that registered it.

Thanks to garbage collection and the fact that the actor probably isn't the type we're after because of the template parameter, we treat null as an expected failure value [FORWARD]

And get rid of it from our results. But we're still not done, since actors can inhabit multiple grid cells. [FORWARD]

So we use the unique filter. And boom. Code that used to be complicated and messy is handled in a few lines. [FORWARD]



# Ranges - Current Usage

```
for( ASomeType* FoundActor : GetActorsInCells< ASomeType >( Position, Radius ) )  
{  
    /* Do your thing! */  
}
```

RETURNAL



Even better, at the call site it's just a single foreach statement.

It makes for immeasurably cleaner code, and we solve some memory allocation problems with this;

and the burden of maintenance is vastly reduced since there's way less code you need to learn. [FORWARD]

# Ranges - C++20 Conformant

```
template< typename ActorType >
auto USpatialGridObject::GetActorsInCells( const FVector& Position, float
MaxDistance )
{
    using std::ranges;
    using std::views;
    return IndexRangeBox( Position, MaxDistance )
        | transform( /* ... */ )
        | join
        | transform( /* ... */ )
        | filter( []( ActorType* Actor ) { return Actor != nullptr; } )
        | unique;
}
```

RETURNAL



When we eventually update to C++20 only the implementation of that function needs to change.

The only real difference is that `Map` becomes `transform`. Otherwise the code is essentially identical.

Ranges are great, especially if you're in to functional programming and are used to lazy evaluation of complicated code. [FORWARD]

# Ranges

- \* Interesting realisation thanks to this
  - Now my spatial grid represents a specialised hash map
  - Spatial hashing is nothing new, but I arrived at the same idea through ranges
  - Gives me thoughts on how algorithmic optimisation can be applied instead of just SIMD optimisation

RETURNAL



Implementing the grid with ranges [FORWARD]

Actually led me to a rather interesting realisation. [FORWARD]

My spatial grid from an interface perspective actually represents a specialised hash map. [FORWARD]

Now, spatial hashing is nothing new at all. It's been around for a while, but I ended up at the same idea anyway thanks to implementing things with ranges. [FORWARD]

As such, now there's a whole new range of algorithmic optimisation I should be considering instead of just SIMD optimising all the lookups. But that's a problem for future Ethan to worry about. [FORWARD]

# Duplicate Rooms

- \* Sometimes, we want more than one instance of a room in a run
  - Fractured Wastes' branches all end in the same room
- \* UE4 technically supports it, but an implementation bug will result in infinite waiting for level load
  - Generate unique names to pass to world streamer

RETURNAL



Let's get back to the actual layout. A bit of a bug came about thanks to our custom loader. See, [FORWARD]

Sometimes we actually want more than one instance of a room in a run. There's the odd treasure room to think of, but in particular [FORWARD]

Each of the Fractured Wastes' branching paths end in a different instance of the same room. [FORWARD]

Multiple instances of a level is actually something UE4 is quite capable of doing, but the implementation of the world streamer only expects unique level names.

If you insert three instances of a level and try to load the second instance, the streamer will only match up loading state with the first instance of the level - it's expecting things to be loading but the first instance is unloaded and not loading and you'll end up permanently stalling the streamer entirely. [FORWARD]

This can be fixed without an engine modification by generating a unique name for the level yourself and passing it along. Low tech solution, but hey it works! [FORWARD]

# LODs

- \* Take an entire room, generate low detail mesh
- \* Built in to Unreal Engine
  - Except we couldn't use default implementation thanks to it relying exclusively on world composition
  - No problem, we'll repurpose code to our needs

RETURNAL



We also have some rather wide vistas that you can observe in our game, and that requires a few pieces of tech to work properly. First up, LODs. Pretty simple idea: [FORWARD]

Take an entire room, generate a low detail mesh from it, and only have that loaded in memory at all times instead of the full level.

Swap in the full room when you get closer, and boom. You've got yourself an open world streamer. [FORWARD]

This is actually built in to UE4 [FORWARD]

IF you're using the world composition tool. Which we weren't. Because we can't. Oops. [FORWARD]

No problem though, it's easy enough to just copy the code and repurpose it to our needs and update our streamer to take LODs in to account. [FORWARD]



## UE's Voxel LODs

- \* Needed to bake to potato quality for generation speed and memory concerns
- \* Hard to get looking good at higher quality without using tons of memory
- \* Most rooms look just fine with this method though once environmental effects are applied

RETURNAL



Hooooo boy. Thanks to being a playstation 5 game, we were throwing some high poly data at it.

UE's built in method for generating level LODs is to voxelise and generate a new mesh and material for it. [FORWARD]

But hoo boy. We were throwing some high poly data at it. We needed to basically use potato-quality settings for a LOD to bake in a reasonable amount of time. [FORWARD]

There's some real resource issues with the voxeliser, especially since UE has an annoying habit of retaining GPU memory in commandlet mode even after entirely releasing resources and forcing a garbage collection. [FORWARD]

Still, potato quality actually worked well for most of our rooms. Environmental effects tended to hide the worst of it.

And, uh, then there's instances like this. [FOWARD]



Woof.

Yeah, do not want. There's a number of things going on there, and not just the potato quality settings

When you're voxelising something, if your geometry's straight lines aren't perpendicular to axis lines, then it tends to look really messy really quickly.  
[FORWARD]

# Houdini Generated LODs

- \* Immediately solved quality concerns
- \* Reuse same materials for consistent look
- \* Came in way too late for overall usage
  - ...but were applied in very high impact places

RETURNAL



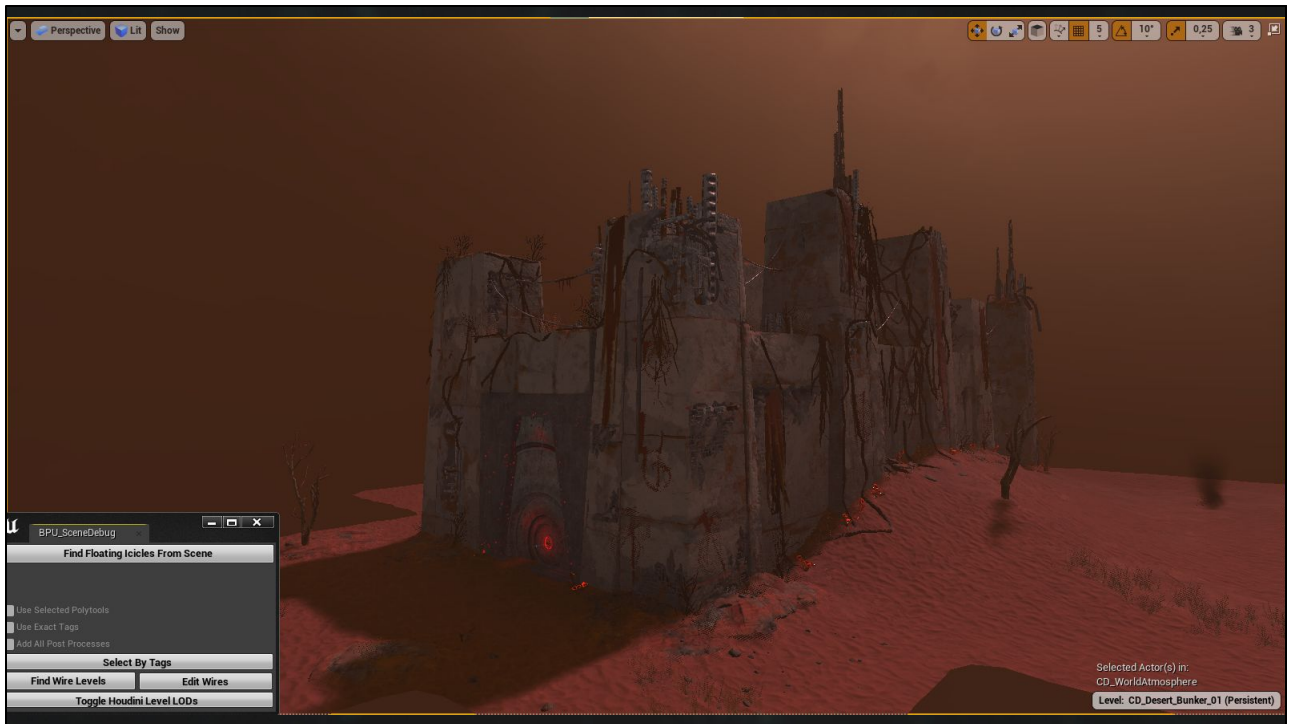
As we already use Houdini quite a bit for our environment art, well hey, why don't we just use Houdini to generate LODs? [FORWARD]

And immediately it solved quality concerns. Just one day of our technical artist's time was all we needed to get something comparable in poly count that look significantly higher quality. [FORWARD]

We could also reuse the same materials for a consistent look, which comes with its own dangers. [FORWARD]

But as this came in rather late in the development process, we couldn't switch over to using it everywhere. [FORWARD]

It ended up being used in very high impact places, which essentially negated any issues we might have had reusing high quality materials everywhere. [FORWARD]



Just take the previous example, now redone in Houdini.

That's definitely the kind of LOD you'd take home and introduce to your parents.

We really should have been using Houdini for LODs way earlier in the development process.

So, lesson learnt there.[FORWARD]

## We're Still Not Quite There

- \* Mesh decals needed to be taken in to account
  - Mesh projected to flat plane
- \* Used liberally to create variation

RETURNAL



But there's actually something still missing from the Houdini LODs. [FORWARD]

And it comes down to our usage of mesh decals. [FORWARD]

One of our technical artists developed a method to project meshes created in a certain way to a flat plane, [FORWARD]

allowing the environment team to just slap mesh arbitrarily all over the game and trick you into thinking you're looking at hand-crafted unique meshes everywhere.  
[FORWARD]





Just as an example. You might be thinking this building in the distance is looking mighty nice. But if we get closer [FORWARD]



Once the full level gets loaded in there is actually a noticeable visual difference. And if we zoom in just a little bit [FORWARD]



What we see here are actually [FOWARD]

Some of those mesh decals I was talking about. They weren't going to be easy to replicate in Houdini with the time we had. But I already had a trick up my sleeve to handle that. [FORWARD]

# NeverLOD

- \* Type of sublevel that loads in with LODs instead of rest of room
  - tl;dr - Persistent level chunks
- \* Use cases include
  - Effects such as fog
  - Teleporter pads
  - Mesh decals

RETURNAL



And that's what we called the NeverLOD. [FORWARD]

It was named as such because it is a type of sublevel that loads in with the LOD representation and stays in memory instead of being streamed in and out with the rest of the room. [FORWARD]

And yeah, it's another case of "first use case named it" since it's really just a persistent level chunk.

Getting that messaging across to the rest of the team was difficult. So that's another lesson learnt, always remember to name things correctly. [FORWARD]

But they actually came about [FORWARD]

Because of effects such as fog. Some rooms required fogging to be visible externally, and copying the actor over to the LOD level would result in visible popping between simulations. So in they went to the persistent block. [FORWARD]

Teleporter pads found a natural home in there, as they needed to always be in memory so that you could locate them on the map and initiate the loading sequence. [FORWARD]

And, yes, the mesh decals I just spent a couple of slides talking about made their way in there so we could ship the game on time.

[QUESTION TIME]

[FORWARD]

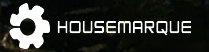




# THE FINAL PIECES



RETURNAL



# Landscape

- \* Fill in the void between rooms
  - Vistas wouldn't work without it
- \* Adapt landscape to room layout
  - Most obvious in Crimson Wastes

RETURNAL



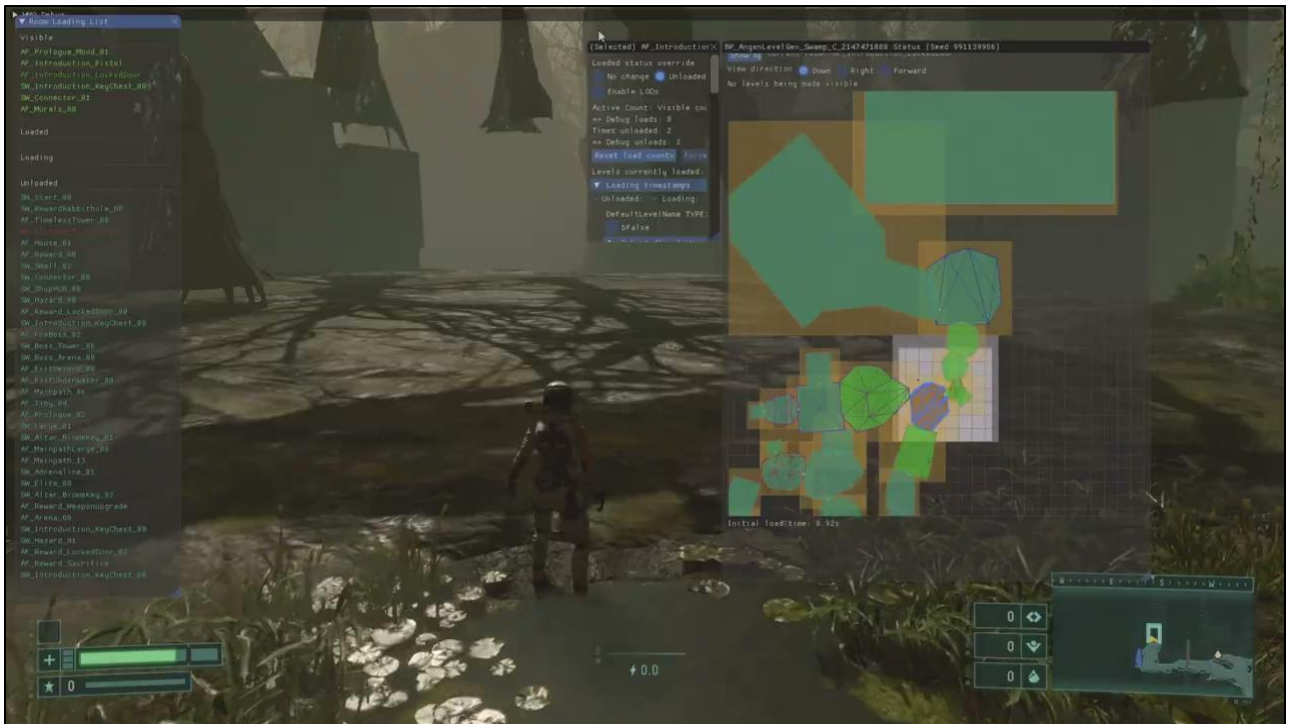
There's only a two more things left for that initial five seconds, and the first of them is the landscape. [FORWARD]

And yeah, [FORWARD]

the vistas I mentioned would just be void if it weren't for the landscape. UE4 comes with a landscape system, but again we ended up writing our own. And that's simply because .

We need to actually modify our landscape at runtime to adapt to our procedural room layout. [FORWARD]

It's most obvious in Crimson Wastes, anything outside of the critical path before you get to the cliffside is basically terrain. [FORWARD]



In this video though, once again we're at the introductory room we've looked at a few times.

Right now, I've forced the LODs off and unloaded the room itself to show that our landscape has been pushed up to meet the room. If you looked at this from the outside, there would be no visible gaps.

It's generated in such a way that when you do actually load in the full room, the landscape doesn't interfere with anything placed inside.

And that's purely thanks to the manually-defined enclosing volumes. We just plain don't let landscape vertices get inside them. [FORWARDx2]

# Trees

- \* Fill the landscape with vegetation
  - Except for where you have a room overlap
- \* ...Well, not just vegetation
  - Another example of “first use case named it”

RETURNAL



There's one last thing you might have spotted in that video, and that's trees.  
[FORWARD]

Fill our landscape up with vegetation. We don't want it looking like a lifeless flat surface, so fill it up [FOWARD]

And delete anything that has an overlap with a room volume.

Of course, we call this our tree generator [FORWARD]

But we don't just place trees with it. And as you're about to see, [FORWARD]

It's yet another example of “first use case named it”. Ready? And [FORWARD]



Look at all these trees! Can you see them? [FORWARD]

There's a tree, and there's a tree, everywhere is trees!

Actually, I'm going to start a petition to rename real-world skyscrapers to trees.

That's going to be way easier than naming things appropriately in a game codebase, I tell you what [FORWARD]

## Get In To The Game In Five Seconds

- \* Load in designated “boot layer” sublevels to start room
  - Let player run around ASAP
- \* Generate layout
- \* Load in LOD representation of rooms
- \* Load area immediately surrounding player
- \* Generate terrain
- \* Generate “trees”

RETURNAL



But hey, we're finally there! I've been talking for [QUOTE TIME] solely about five seconds at the start of your play session in Returnal, and honestly I've left out quite a bit.

But there it is. That's the basic gist of how we handle procedural world layout in Returnal.

Although. Uh. [FORWARD]





## CONFESSION TIME

So, uh, I actually have a confession to make to everyone.

You see, uh, I lied. (Nervous laugh) Yeah, I'm a dirty liar. You see, what I did was I called this talk "Never The Same Twice". And, uh, that's just patently untrue.  
[FORWARD]

# The Odds Of Playing The Same Layout Twice

- \* Experiencing the same layout yourself:
  - Negligible
- \* Experiencing the same layout as a friend:
  - Negligible
- \* Experiencing the same layout as someone in this room:
  - Actually kinda high

RETURNAL



So let's engage in a little exercise here. Let's talk about the odds of playing the same layout twice. Fairly low, you'd think? [FORWARD]

If we're talking about you personally, then [FORWARD]

Yeah it's quite low. [FORWARD]

What about you and a friend playing the same layout on different consoles?  
[FORWARD]

Also rather unlikely. [FORWARD]

But what about the chances that you in this room have experienced the same layout as someone in this room? [FORWARD]

Well, that's where it all falls apart. [FORWARD]

# Error Tolerant Generation

- \* Place a random room down
- \* Test room volume for overlap
  - \* Pass
    - Continue as normal
  - \* Fail
    - Rewind and try again

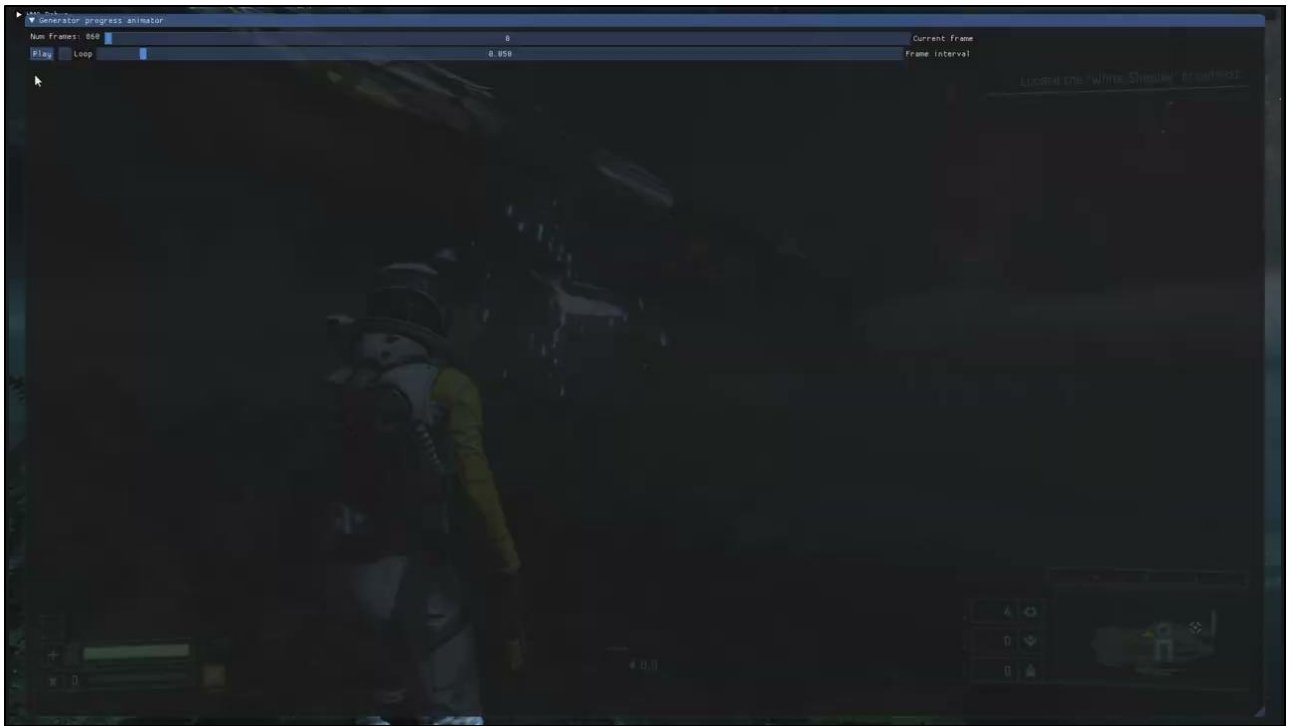
RETURNAL



Remember how I described earlier that our error correction in the generator was to rewind to an earlier state and to eventually give up and try again if it hit too many errors?

Yeah, so, uh,

This is actually kinda bad. Well, more like it's monumentally bad. What do I mean?  
[FORWARD]



The freeform nature of our system and the design of the first biome's rooms meant that world layouts tended to curve in on itself and quickly create overlaps, requiring constant rewinding.

This little video I've got here is illustrating what happens with such an approach. Create, create, fail, discard, try again. Create, create, fail, discard, try again. And so on and so forth.

Interestingly, take a look here. We gave the team no limitations, and yet the layout is constantly prone to near-ninety degree angle changes.

What we actually needed was a system that will generate a layout and unlock the starting room's door in an imperceptible amount of time, and instead we had a system that would just fail over and over again.

This video is actually one of our best behaving layouts, clocking in at 31 attempts before succeeding and finishes in a couple of seconds.

But for many inputs we'd be waiting 30 seconds, one minute, two minutes, and wouldn't get a layout.

This isn't just bad for players, it was severely impacting iteration and testing times.

There was only one way to make this work without throwing away everything and

starting from scratch. [FORWARD]

# Offline Level Generation

- \* Serialise successful layout to disk
- \* Do it 1000 or more times for each combination of
  - Biome
  - Possible gameplay flag combinations
  - “Deck of cards”

RETURNAL



And that was with the offline level generator. Short story? [FORWARD]

Take a layout, serialise it to disk [FORWARD]

And then do it something like 1000 times for each possible combination of [FORWARD]

Biomes, [FORWARD]

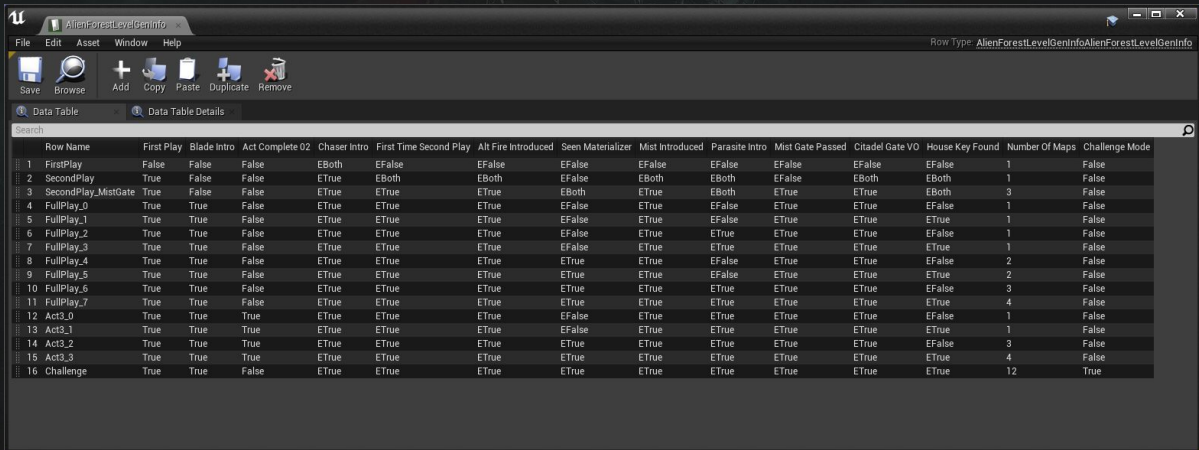
Gameplay flags, [FORWARD]

And deck of cards.

Now, you might be thinking that given the flag system I've described that this could get out of hand very quickly. So let's check some examples[FORWARD]



# Offline Level Generation



Row Type: AlienForestLevelGenInfoAlienForestLevelGenInfo

Row Name	First Play	Blade Intro	Act Complete 02	Chaser Intro	First Time Second Play	Alt Fire Introduced	Seen Materializer	Mist Introduced	Parasite Intro	Mist Gate Passed	Citadel Gate VO	House Key Found	Number Of Maps	Challenge Mode
1 FirstPlay	False	False	False	EBoth	EFalse	EFalse	EFalse	EFalse	EFalse	EFalse	EFalse	EFalse	1	False
2 SecondPlay	True	False	False	ETrue	EBoth	EBoth	EFalse	EBoth	EBoth	EFalse	EBoth	EBoth	1	False
3 SecondPlay_MistGate	True	False	False	ETrue	ETrue	ETrue	EBoth	ETrue	EBoth	ETrue	ETrue	EBoth	3	False
4 FullPlay_0	True	True	False	ETrue	ETrue	ETrue	EFalse	ETrue	EFalse	ETrue	ETrue	EFalse	1	False
5 FullPlay_1	True	True	False	ETrue	ETrue	ETrue	EFalse	ETrue	EFalse	ETrue	ETrue	ETrue	1	False
6 FullPlay_2	True	True	False	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	ETrue	ETrue	EFalse	1	False
7 FullPlay_3	True	True	False	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	ETrue	ETrue	ETrue	1	False
8 FullPlay_4	True	True	False	ETrue	ETrue	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	EFalse	2	False
9 FullPlay_5	True	True	False	ETrue	ETrue	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	ETrue	2	False
10 FullPlay_6	True	True	False	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	3	False
11 FullPlay_7	True	True	False	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	4	False
12 Act3_0	True	True	True	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	ETrue	ETrue	EFalse	1	False
13 Act3_1	True	True	True	ETrue	ETrue	ETrue	EFalse	ETrue	ETrue	ETrue	ETrue	ETrue	1	False
14 Act3_2	True	True	True	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	EFalse	3	False
15 Act3_3	True	True	True	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	4	False
16 Challenge	True	True	False	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	ETrue	12	True

RETURNAL



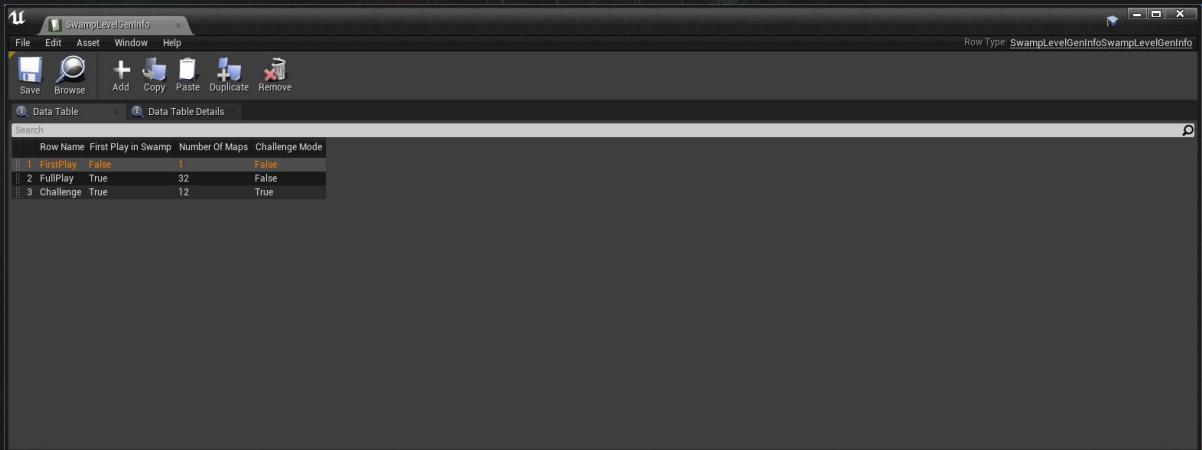
This is the offline generator information asset for the first biome, the Overgrown Ruins.

It's actually our most complicated biome thanks to all the mechanics that need to be introduced.

But even then, the actual possibilities that you come across meant that we only needed to account for 15 different combinations of flags for the campaign, and a special case down the bottom for the daily challenge maps.

Quite manageable indeed. [FORWARD]

# Offline Level Generation



The screenshot shows a software window titled "SwampLevelGenInfo". It has a menu bar with "File", "Edit", "Asset", "Window", and "Help". Below the menu is a toolbar with icons for "Save", "Browse", "Add", "Copy", "Paste", "Duplicate", and "Remove". The main area is a "Data Table" with a search bar and a table containing 3 rows. The table has columns: "Row Name", "First Play in Swamp", "Number Of Maps", and "Challenge Mode".

Row Name	First Play in Swamp	Number Of Maps	Challenge Mode
1 EmptyPlay	False	1	False
2 FullPlay	True	32	False
3 Challenge	True	12	True

RETURNAL



The biomes in the back half of the game are nowhere near as complicated.

This is the flag combinations for the Echoing Ruins, and is fairly representative of the second three biomes - we need to know if you've completed the "first time visiting" sequences, but otherwise just go ahead and do your thing. [FORWARD]

## Offline Level Generation

- \* Small amount of flag permutations makes offline level generation feasible
- \* Never generate a layout on the user's machine
- \* Fun stat - these consume 3 gigabytes of disk space

RETURNAL



So, while the first biome could have gone wrong [FORWARD]

The amount of permutations were entirely manageable.

We can serialise a ton of layouts to disk, and instead of using RNG to generate a layout we can instead use RNG to select a layout from a pool. Sweet!

Buuuuuuut this is why I lied to everyone: [FORWARD]

Thanks to this system, we never generate a layout on the user's machine. Not even once. [FORWARD]

We did pay a small size price on disk for this amount of data, but end of the day is that it was critical to that "Five seconds" mantra. [FORWARD]

# Offline Level Generation

- \* Should have actually had it from the start, since it enables:
  - \* Debugging
  - \* Testing determinism
  - \* Consistency between builds
  - \* Opens up certain aspects of design
    - Hand-crafted layouts
    - Over-the-air daily challenges

RETURNAL



And it's absolutely something we should have had from day one.

Not as a gameplay feature. No, we should be doing the procedural generation at runtime.

But it opens quite a number of other features such as debugging.

We can serialise a layout as part of a crash dump, or you can have a single layout for testing that's immune to input changes.

Determinism testing also becomes a reality, and as such you also get consistency between builds.

And yes, it does also give you design flexibility. Right now, daily challenges use those basic inputs described earlier in the talk. But what if you want a layout that's ten bonus arenas in a row? That currently requires new code. With serialised layouts, we just send that to the user as part of the daily challenge instead. [FORWARD]



But that's ultimately part of our "Lessons Learnt" for the project. So to wrap up, let's go over some of them. [FORWARD]

# Lessons Learnt

- \* Error avoidance is better than error tolerance
  - Sensible constraints instead of none
    - Actually increases design freedom
  - Pattern analysis and suggestion for layout paths
    - Catch at design time room sizes that are missing
- \* Verticality needs more work

RETURNAL



Offline level generation came about [FORWARD]

Because our generator was all about error tolerance. Switching over to an error avoidance model will give more robust results. [FORWARD]

It will mean adding some constraints to the system [FORWARD]

But those constraints actually increase design freedom, by making things such as loopbacks and overlaps possible. [FORWARD]

Using pattern analysis to determine paths instead of just requesting new rooms [FORWARD]

And combining it with automation testing will allow us to identify room sizes that we need for a more consistent experience. [FORWARD]

This would all go hand-in-hand with verticality. Do the above, now do it for three dimensions. We had limited use of vertical layouts in Returnal, and by accounting for the inevitable design desires up front you can write a more flexible and performant system [FORWARD]



# Lessons Learnt

- \* Generator shouldn't be so sensitive to tiny changes
  - Move an actor in a room? Seed gives you a new layout
  - Only elements that actually affect layout are volumes and connections
  - Bad for bug reproduction between builds

RETURNAL



There was also a persistent problem [FORWARD]

Where our generator was very sensitive to tiny changes. Just as an example, [FORWARD]

All you would need to do is move a single actor in any room and the generator would give you an entirely different layout. [FORWARD]

The only elements that actually mattered were the room volumes and the connecting points, but that's one of those hindsight things. [FORWARD]

This meant that trying to use a seed one build later when trying to track down a bug would just plain not work. So yeah, important lesson, determinism can be fragile. [FORWARD]

# Lessons Learnt

- \* Static objects in rooms needs to be subdivided into discrete levels
  - The streamer can't tell the difference between a room the size of a closet and a room the size of Canada
  - Move game logic in to separate sublevel, let the streamer get higher quality mesh into memory in smaller chunks to avoid memory blowouts

RETURNAL



In terms of world construction though, we started making level content well before the loading system was finalised. [FORWARD]

As such, it was going to be painful to go back through and update our levels to match the technical requirements. What you should be doing from the start is ensuring your static meshes and lights and anything else that doesn't change in to much smaller chunks. [FORWARD]

Remember how each room was defined by a volume? This meant that the streamer can't tell the difference between a room the size of a closet, and a room the size of Canada. (SORRY JOKE???) [FORWARD]

This also means that you need to treat game logic separately, since it generally isn't static and we wanted to use a Room as a gameplay concept. With discreet smaller static chunks, we could have managed memory a lot better than we did, and our systems already handled sublevels so splitting logic in to one would have been very effective. [FORWARD]

# Lessons Learnt

- \* We use levels as big giant prefabs
  - Only native form of actor container that can be serialised
  - A better prefab container will allow us to do much more from the design side in every department
- \* Boot layers are great until you need to implement suspend and resume

RETURNAL



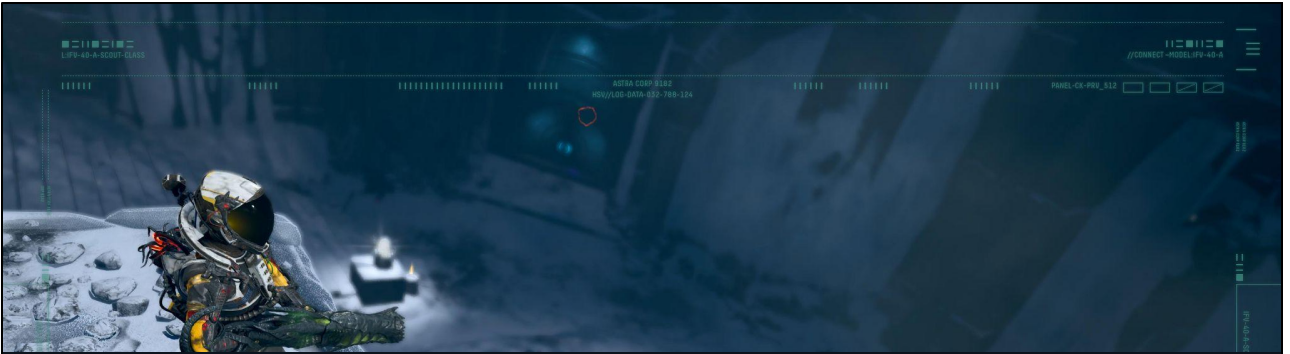
There's also a bit of an Unreal Engine-ism here [FORWARD]

Where we use levels as big giant prefabs. Why? [FORWARD]

Because it's the only native form of actor container that can be serialised. They're trying to treat Level Instances as prefabs in UE5, [FORWARD]

But that's still incredibly limiting. A real actor container will let us better handle spawn pools for example. And just one last thing: [FORWARD]

Don't do boot layers. They *were* an acceptable compromise to getting in to the game within five seconds, until suspend and resume became a thing and the player could restart a session from literally anywhere. We well exceed the five second rule when resuming a session as we can't make any assumptions about the world around a save point until we have everything generated and loaded - not just for technical reasons, but also because all the vistas we have in the game would visibly pop in. So yeah, boot layers. Not even once. [FORWARD]



# QUESTIONS?





**THANKS!**

