25/2/2023 / DAVID MARCELIS

SCALING TOOLS FOR MILLIONS OF ASSETS

DECIMA
HORIZON
FORBIDDEN WEST

Welcome everyone, to my talk on Scaling tools for millions of assets. Before we get started, please remember to put your phones on silent.

Let’s start out with a little scenario. Imagine you're an artist (this might be difficult for a room of mostly programmers)… but! *CLICK* You’re an environment artist, looking through concept art for ideas.

As you've just finished all your tasks for previous project and you are excited to start on the next game.

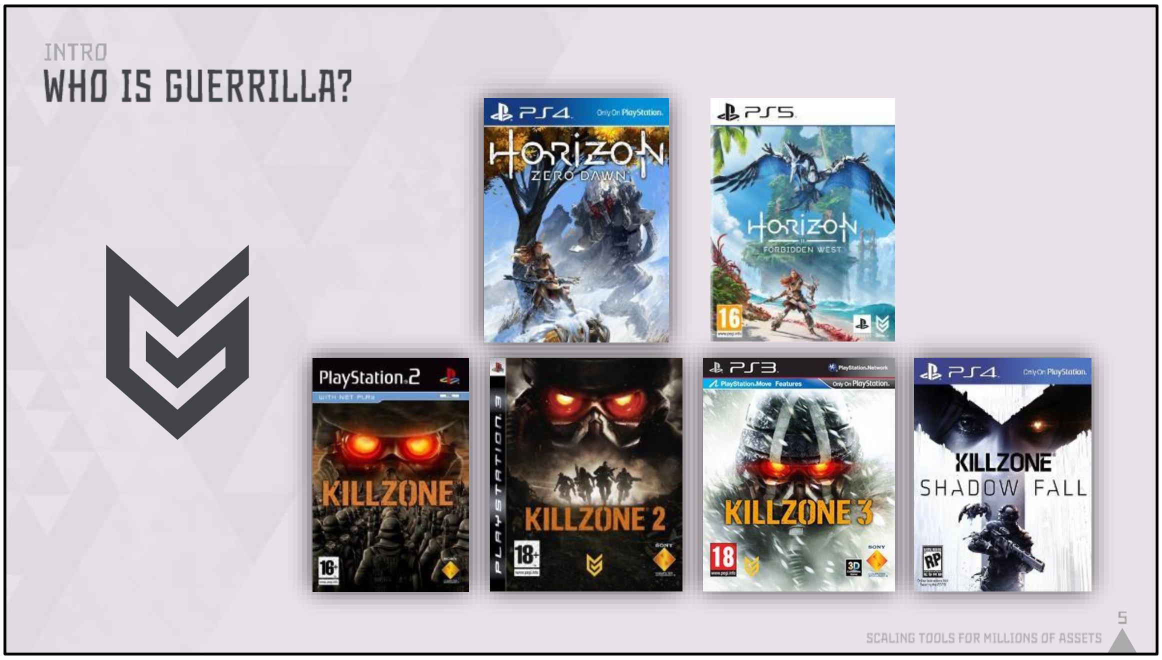
CLICK But then your realize your SSD can only fit one project so
you first delete the old one. Once that is finished, *CLICK*

You start the 8 hour sync to get all the assets for the new branch.
The next working day, you load up the level editor and search the Asset Browser for “Oseram" assets. While you wait for the results, *CLICK* you go get a coffee, check your social media, and call your mom. With your coffee in hand, your level loaded and the assets displayed, you're finally ready to work.

And then your producer comes up to tell you need to switch back to the old project to fix a critical bug, *CATTO CLICK* and you have to go through the whole thing again...

At the start of Forbidden West this is the situation our tools were in.

What if instead you could switch projects in less than a minute, be in the game without any additional download, and have any asset at your fingertips in real time?


INTRO

WHO IS GUERRILLA?

5

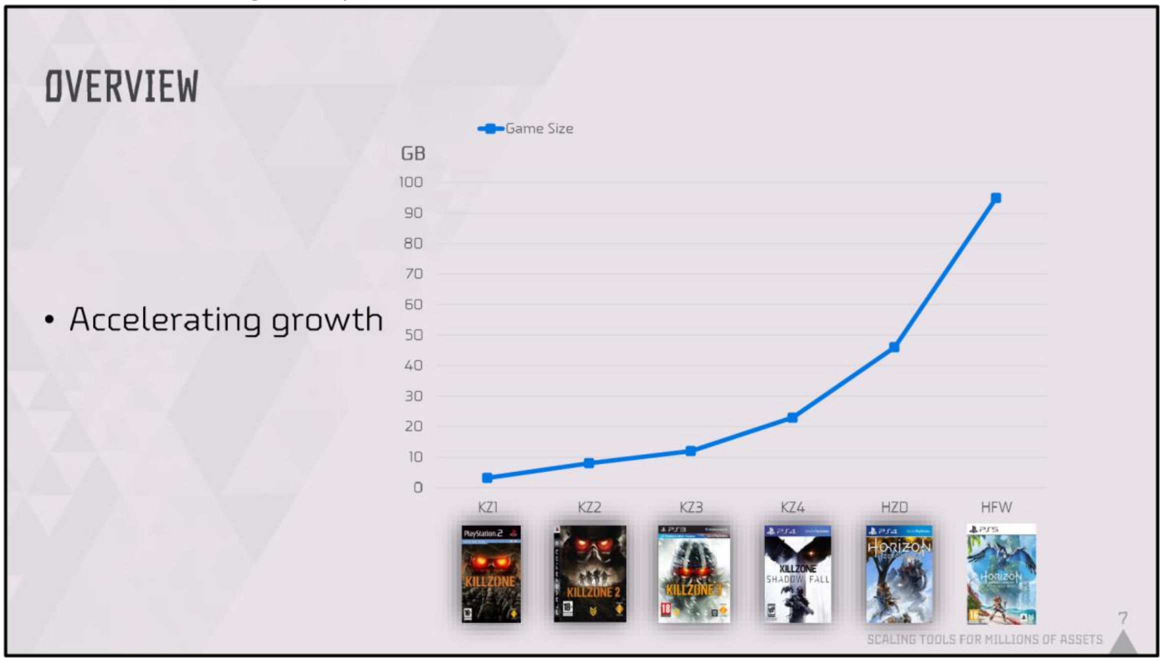
SCALING TOOLS FOR MILLIONS OF ASSETS

In this talk I will explain how we got there.
A little bit about me:

My name is David Marcelis.

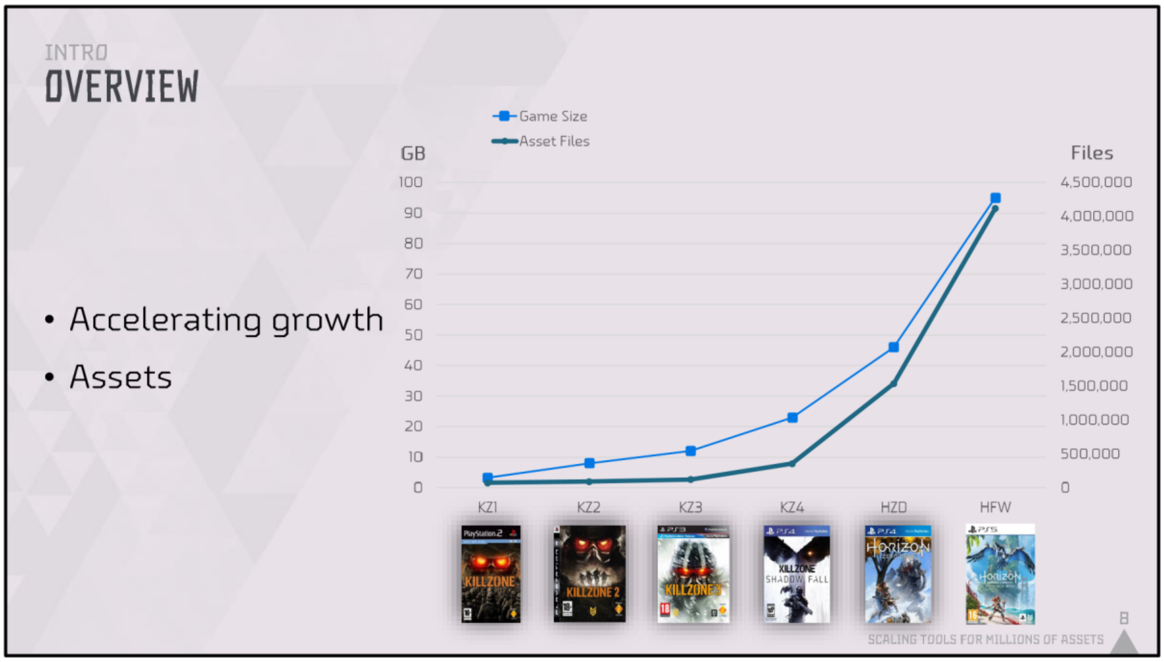
For the last 4.5 years I've been at Guerrilla as a Tools Programmer. As part of the infrastructure team I've worked on source control, build systems, packaging, and other tools-related work for Horizon Forbidden West. As well as supported the release of Horizon Zero Dawn on PC.
Guerrilla was originally known for their Killzone franchise, *CLICK* and more recently for the Horizon series with latest of installment being Horizon Forbidden West. Over the course of these games, the company has grown from a few dozens, to currently over 400 people.
At 95GB Horizon Forbidden West is twice the size of our previous game, and just barely under the PS5 package limit. Not only is the game world larger, but it's also more densely populated. With more detailed foliage, set dressing, and more playable

content. The average completionist run takes around 90 hours, and is filled with over


OVERVIEW

• Accelerating growth

Forbidden West being our biggest game so far is just the latest data point in a trend you must have noticed as players and developers: games are constantly getting bigger, *CLICK* But not only are they getting bigger, the rate of growth is also accelerating



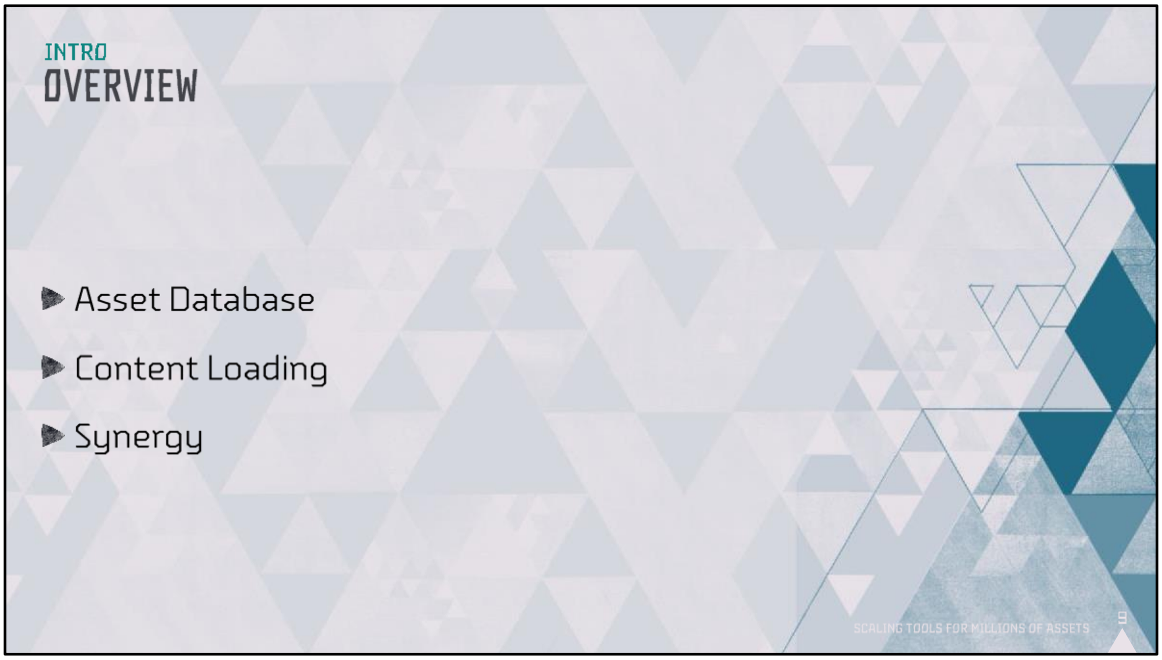
INTRO

OVERVIEW

- Accelerating growth
- Assets

SCALING TOOLS FOR MILLIONS OF ASSETS 8

This isn't just textures getting higher resolution, but also the number of assets involved is increasing. As more and more assets are added and more and more files are involved, the tools slow down and the project becomes cumbersome to work with. *If all these artists would just stop adding ultra high-fidelity moss to every single rock in the game, our life as programmers would be so much easier. But luckily they haven't, so we need to keep pushing the tech and get to give cool GDC talks.* In this talk I will explain how we improved the core infrastructure behind our tooling to make working with 4 million assets feel like a breeze.



INTRO

OVERVIEW

- Asset Database
- Content Loading
- Synergy

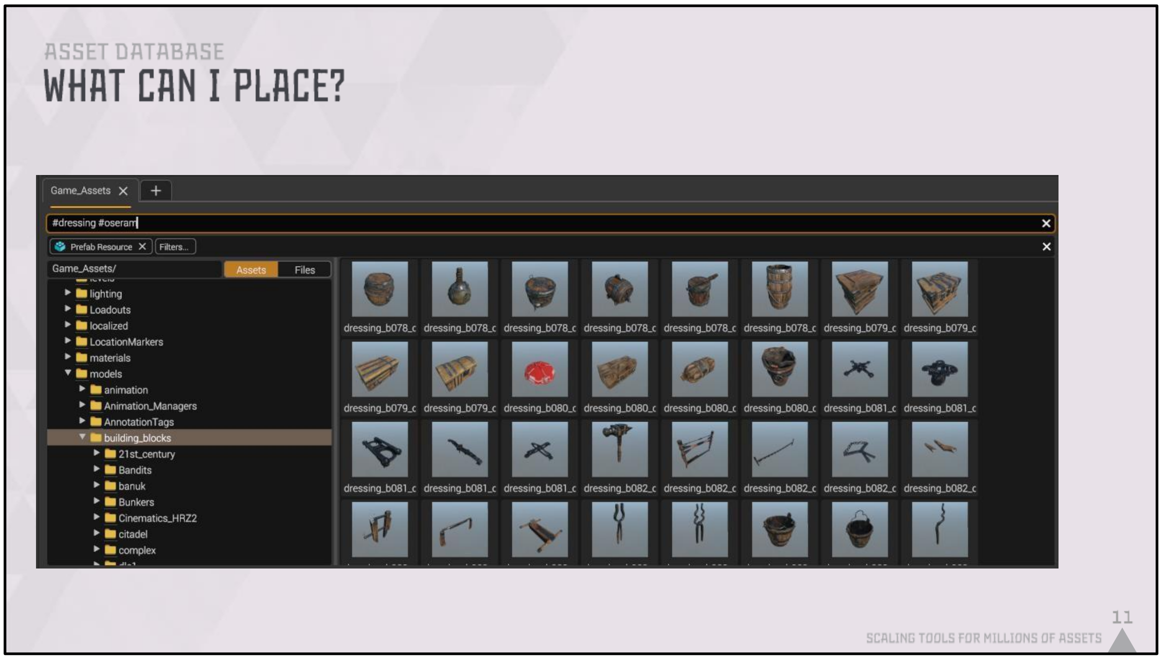
I’ll be covering the 2 major systems that were particularly struggling with this increase in scale:

- First the Asset Database that allows us to reason about all assets
- Secondly how we optimized content loading during *development* for iteration time
- And finally some powerful and maybe surprising ways these 2 systems synergize

ASSET DATABASE

SCALING TOOLS FOR MILLIONS OF ASSETS 10

Let's start out with how we handle this many assets. As a user you want to know things about the content you're working with.

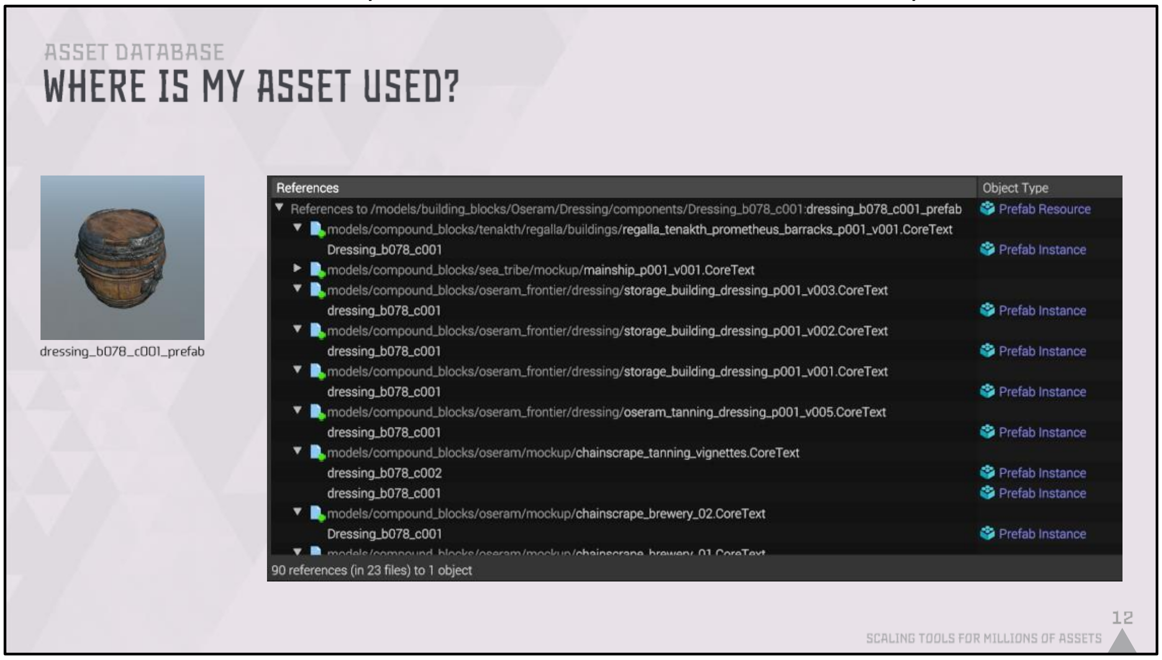


ASSET DATABASE

WHAT CAN I PLACE?

SCALING TOOLS FOR MILLIONS OF ASSETS 11

For example, when doing set dressing or level design, you want to see all objects that you can place in the world based on filters that match your current task.
This Asset browser is a very common and fundamental feature of any level editor.



ASSET DATABASE

WHERE IS MY ASSET USED?

dressing_b078_c001_prefab

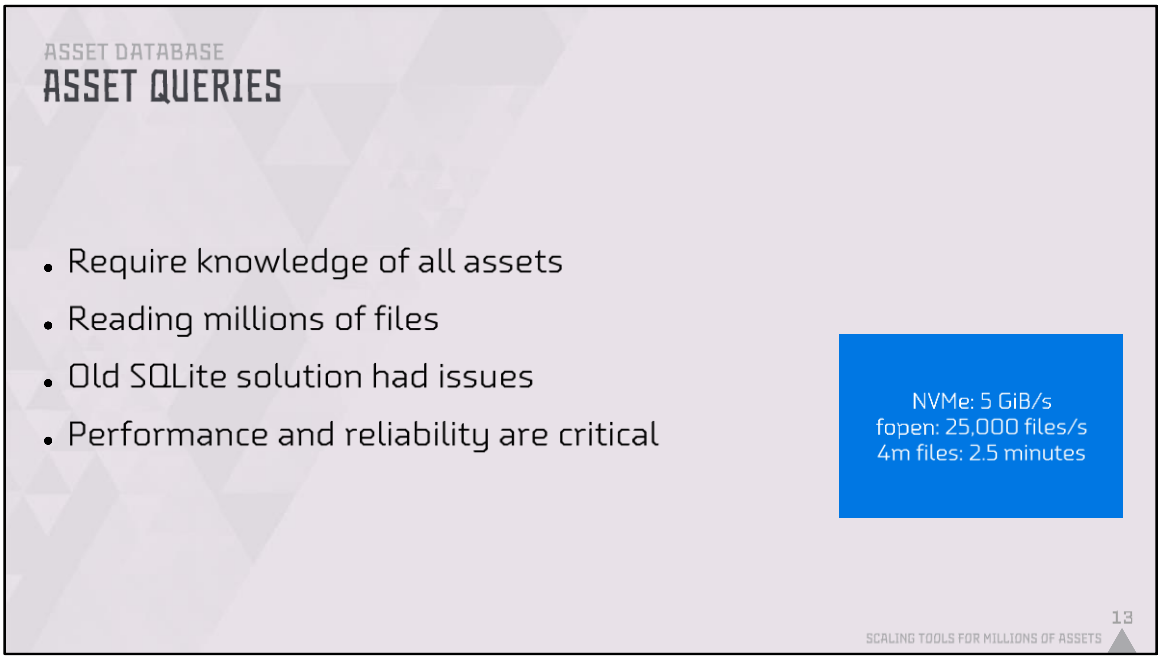
References	Object Type
▼ References to /models/building_blocks/Oseram/Dressing/components/Dressing_b078_c001:dressing_b078_c001_prefab	Prefab Resource
▼ models/compound_blocks/tenakth/regalla/buildings/regalla_tenakth_prometheus_barracks_p001_v001.CoreText	
Dressing_b078_c001	Prefab Instance
▶ models/compound_blocks/sea_tribe/mockup/mainship_p001_v001.CoreText	
▼ models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v003.CoreText	
dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v002.CoreText	
dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v001.CoreText	
dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram_frontier/dressing/oseram_tanning_dressing_p001_v005.CoreText	
dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram/mockup/chainscrape_tanning_vignettes.CoreText	
dressing_b078_c002	Prefab Instance
dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram/mockup/chainscrape_brewery_02.CoreText	
Dressing_b078_c001	Prefab Instance
▼ models/compound_blocks/oseram/mockup/chainscrape_brewery_01.CoreText	

90 references (in 23 files) to 1 object

12

SCALING TOOLS FOR MILLIONS OF ASSETS

But you might also want to know "is this asset ever used" or more specifically "where does this asset get used?"
A Reference Viewer can give you a complete list of all places that link to a particular assets, such as this barrel.



ASSET DATABASE

ASSET QUERIES

- Require knowledge of all assets
- Reading millions of files
- Old SQLite solution had issues
- Performance and reliability are critical

NVMe: 5 GiB/s
fopen: 25,000 files/s
4m files: 2.5 minutes

13

SCALING TOOLS FOR MILLIONS OF ASSETS

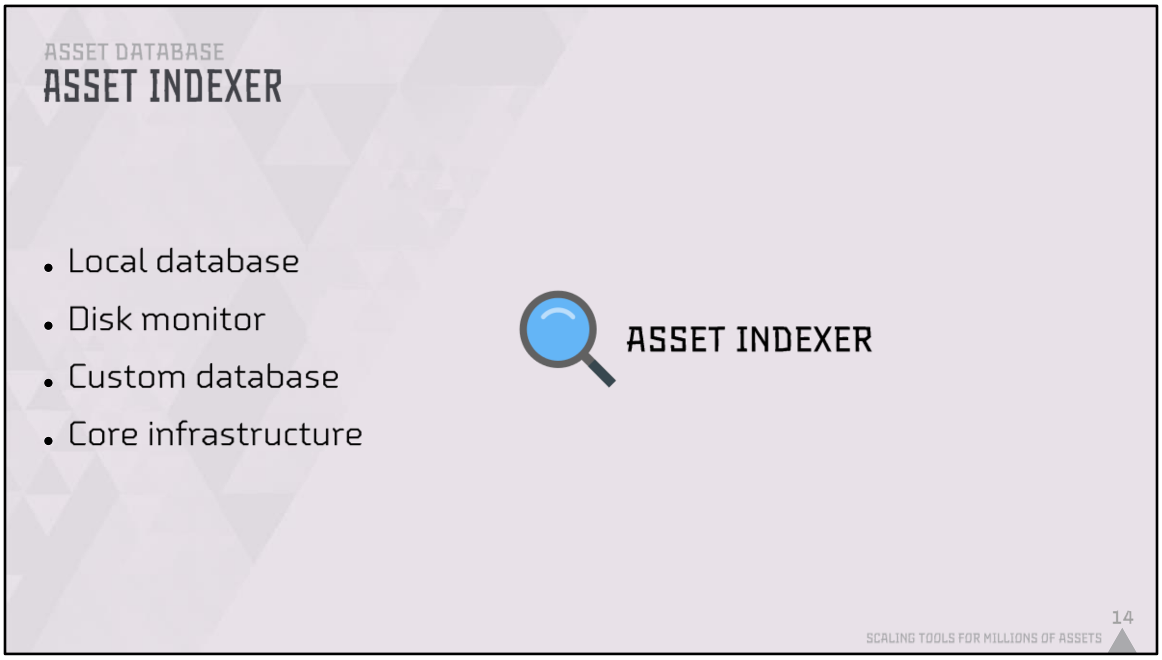
These are two examples of queries that require knowledge of all assets before you can give a conclusive answer. Reading millions of files every time want to find an object to place in the world is not feasible.

CLICK A little side note: this may be surprising, as modern NVMe drives can go over 5GiB/s on linear reads, however we are dealing with a lot of small files, and then it becomes the OS kernel overhead that is the limiting factor. In my test case with NTFS on Windows I was able to open 25,000 files/s, which

means just opening all 4 million files without reading any data would take 2.5 minutes.

So it’s clear you need some kind of database for this. At the start of Forbidden West we already had a previous solution based on an SQLite database. It was only used by the asset browser for which it worked well enough, but it was poorly performing and unreliable to the point of it being a common occurrence for people to throw away the database to force a rebuild in the hopes of a missing assets appearing.

We had large ambitions for the next game. It was going to be much bigger so performance and reliability of our tools are critical. This tech was holding us back, so we set out to completely rewrite
it



ASSET DATABASE

ASSET INDEXER

- Local database
- Disk monitor
- Custom database
- Core infrastructure

ASSET INDEXER

So we created the "Asset Indexer". As the name implies, it's a local database that provides an index into all our 100+ million objects and allows us to reason about the content without having to open any file.

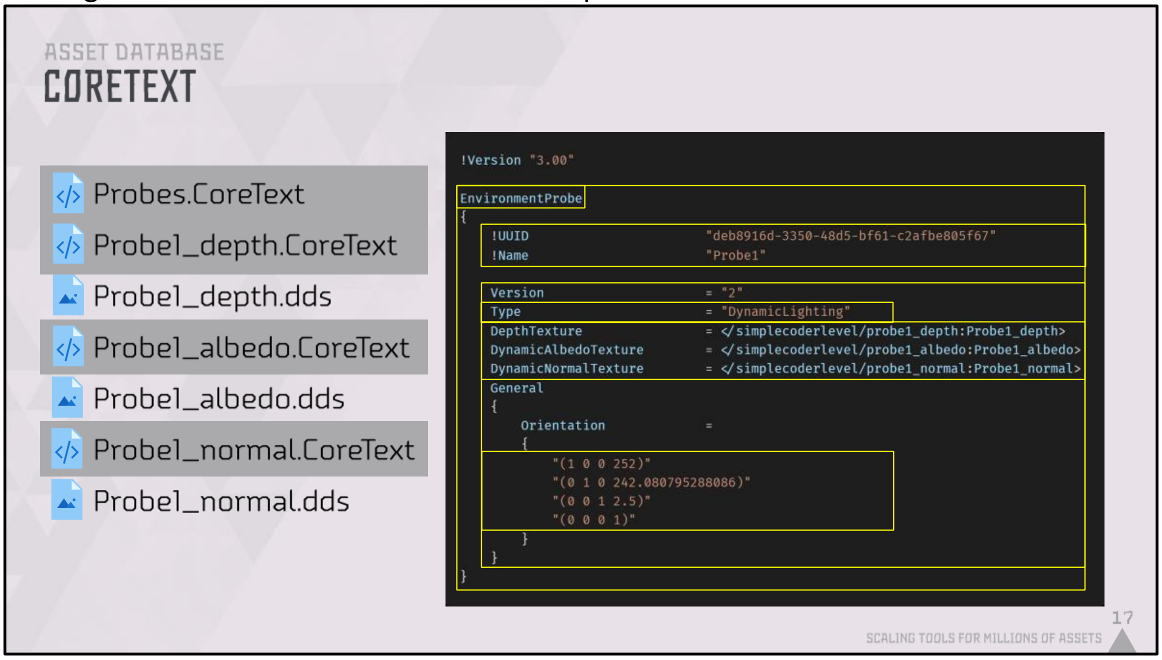
It is a service that runs on each workstation to scan the files on the user's disk and monitors any changes to those files.

The AssetIndexer uses a custom in-

house database. As you might come to learn during this talk, we *really* like making our own tech. So by creating a completely custom solution we are slightly happier as programmers. But we can also apply many tricks to make it faster and more compact than something offthe-shelve by using knowledge about how our specific data is structured.

This has become the core database that all our tooling runs against. Any tool that wants to know about files, objects, or connections between objects asks this database. That includes our editor, DCC packages, python scripts, and even the game itself.

To understand the details of the AssetIndexer and what it enables us to do, first a bit of background about how our content is set up.



ASSET DATABASE

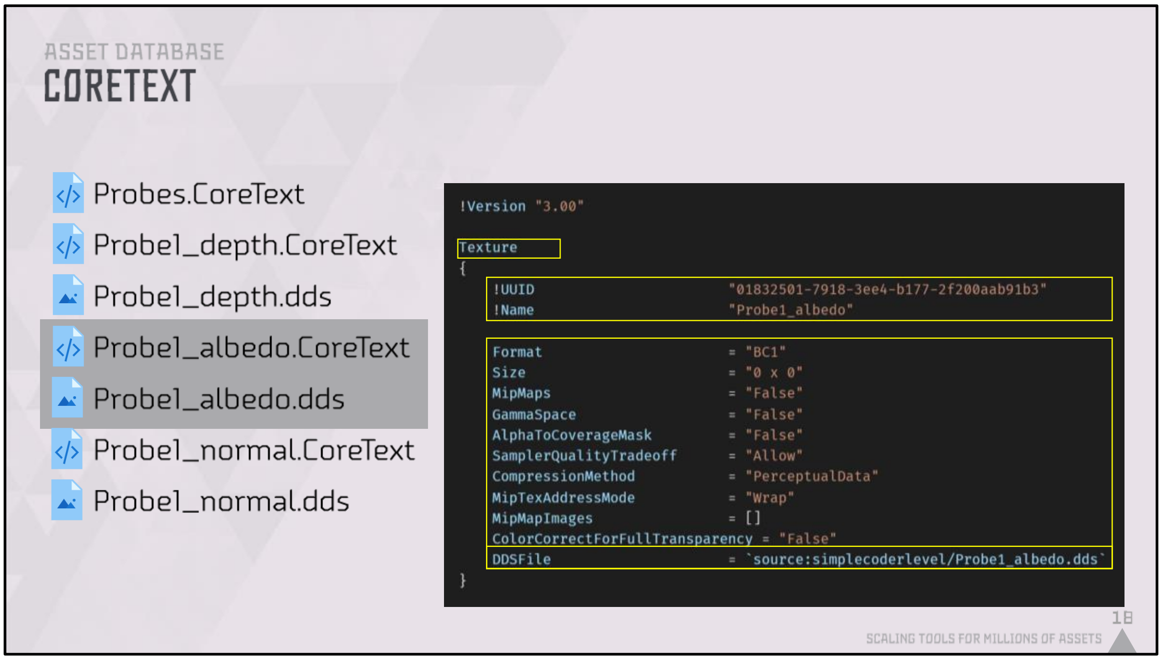
CORETEXT

- Probes.CoreText
- Probe1_depth.CoreText
- Probe1_depth.dds
- Probe1_albedo.CoreText
- Probe1_albedo.dds
- Probe1_normal.CoreText
- Probe1_normal.dds

```
!Version "3.00"

EnvironmentProbe
{
    !UUID                   "deb8916d-3350-48d5-bf61-c2afbe805f67"
    !Name                   "Probe1"

    Version                 = "2"
    Type                    = "DynamicLighting"
    DepthTexture            = </simplecoderlevel/probe1_depth:Probe1_depth>
    DynamicAlbedoTexture    = </simplecoderlevel/probe1_albedo:Probe1_albedo>
    DynamicNormalTexture    = </simplecoderlevel/probe1_normal:Probe1_normal>
    General
    {
        Orientation         =
        {
            "(1 0 0 252)"
            "(0 1 0 242.080795288086)"
            "(0 0 1 2.5)"
            "(0 0 0 1)"
        }
    }
}
```

We refer to all the files necessary to work on the game or produce a shippable build as “Game Assets”. These are the 4 million assets I mentioned, which are about 2TB of data.
There are also another 4 million files of what we call Work_Files. This contains original data that get exported into Game_Assets, such as Maya files for our models, or the Photoshop files for a texture. For the scope of this talk, I will only be focusing on everything in Game_Assets.
We have binary data such as audio wav files, dds and png textures, and some other loose files like bink movies.
But the actual structure of game is stored in CoreText. Coretext is a JSON-like text format that stores every asset you might think of: mesh, quest script, particle system, etc.
Let’s go though an example piece of content to explain this. For this I’ll use the Environment Probes, which provide pre-baked reflection data for the game, but the details of what it does are not important.
Here we have Probes.CoreText. It is a CoreText file that contains 1 object, an Environment Probe.
CLICK Every CoreText file contains a number of Object blocks. *CLICK* An object block starts with a typename. This matches a class in our C++ code that has been exposed so an instance can be created at runtime. *CLICK* Every object is then expected to have a UUID and a Name. *CLICK* What follows are the type-specific attributes, such as the type of lighting it captures and the transform. As you can see, all attribute values are stored as strings. Each type like a transform, Boolean, or enum has its own string representation that is defined in code. *CLICK* The exception being links to other objects, which are stored in a fixed format that is shared by

all types. These have a different syntax using angular brackets that distinguishes them from other attributes. *CLICK* Here you see the links to the three texture objects that contain the baked data. These texture objects live in their own files. Let’s follow one of the links.


ASSET DATABASE

CORETEXT

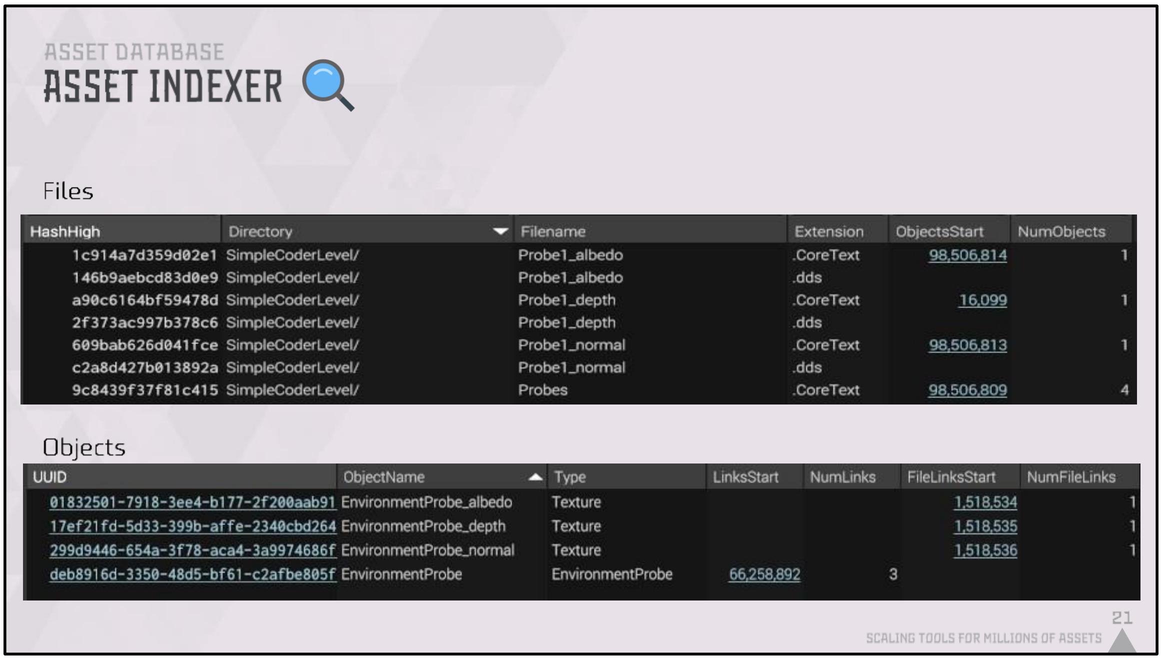
- Probes.CoreText
- Probe1_depth.CoreText
- Probe1_depth.dds
- Probe1_albedo.CoreText
- Probe1_albedo.dds
- Probe1_normal.CoreText
- Probe1_normal.dds

```
!Version "3.00"

Texture
{
    !UUID                         "01832501-7918-3ee4-b177-2f200aab91b3"
    !Name                         "Probe1_albedo"

    Format                        = "BC1"
    Size                          = "0 x 0"
    MipMaps                       = "False"
    GammaSpace                    = "False"
    AlphaToCoverageMask           = "False"
    SamplerQualityTradeoff        = "Allow"
    CompressionMethod             = "PerceptualData"
    MipTexAddressMode             = "Wrap"
    MipMapImages                  = []
    ColorCorrectForFullTransparency = "False"
    DDSFile                       = `source:simplecoderlevel/Probe1_albedo.dds`
}
```

18

SCALING TOOLS FOR MILLIONS OF ASSETS

Following that link to EnvironmentProbe_albedo.CoreText, *CLICK* we again see the typename of the object, *CLICK* the UUID and objectname, *CLICK* some texture settings like compression format and resolution, again stored as strings, *CLICK* and finally a link to the dds file that holds the actual data. This is a different link format from the object links, but again strictly defined and using a different syntax indicated with backticks. This enables us to reason about relations between objects and binary files. *CLICK* Obviously we don't edit text files all day, that's so 10 years ago.



ASSET DATABASE

ASSET INDEXER

Files

HashHigh	Directory	Filename	Extension	ObjectsStart	NumObjects
1c914a7d359d02e1	SimpleCoderLevel/	Probe1_albedo	.CoreText	98,506,814	1
146b9aebcd83d0e9	SimpleCoderLevel/	Probe1_albedo	.dds		
a90c6164bf59478d	SimpleCoderLevel/	Probe1_depth	.CoreText	16,099	1
2f373ac997b378c6	SimpleCoderLevel/	Probe1_depth	.dds		
609bab626d041fce	SimpleCoderLevel/	Probe1_normal	.CoreText	98,506,813	1
c2a8d427b013892a	SimpleCoderLevel/	Probe1_normal	.dds		
9c8439f37f81c415	SimpleCoderLevel/	Probes	.CoreText	98,506,809	4

Objects

UUID	ObjectName	Type	LinksStart	NumLinks	FileLinksStart	NumFileLinks
01832501-7918-3ee4-b177-2f200aab91	EnvironmentProbe_albedo	Texture			1,518,534	1
17ef21fd-5d33-399b-affe-2340cbd264	EnvironmentProbe_depth	Texture			1,518,535	1
299d9446-654a-3f78-aca4-3a9974686f	EnvironmentProbe_normal	Texture			1,518,536	1
deb8916d-3350-48d5-bf61-c2afbe805f	EnvironmentProbe	EnvironmentProbe	66,258,892	3		

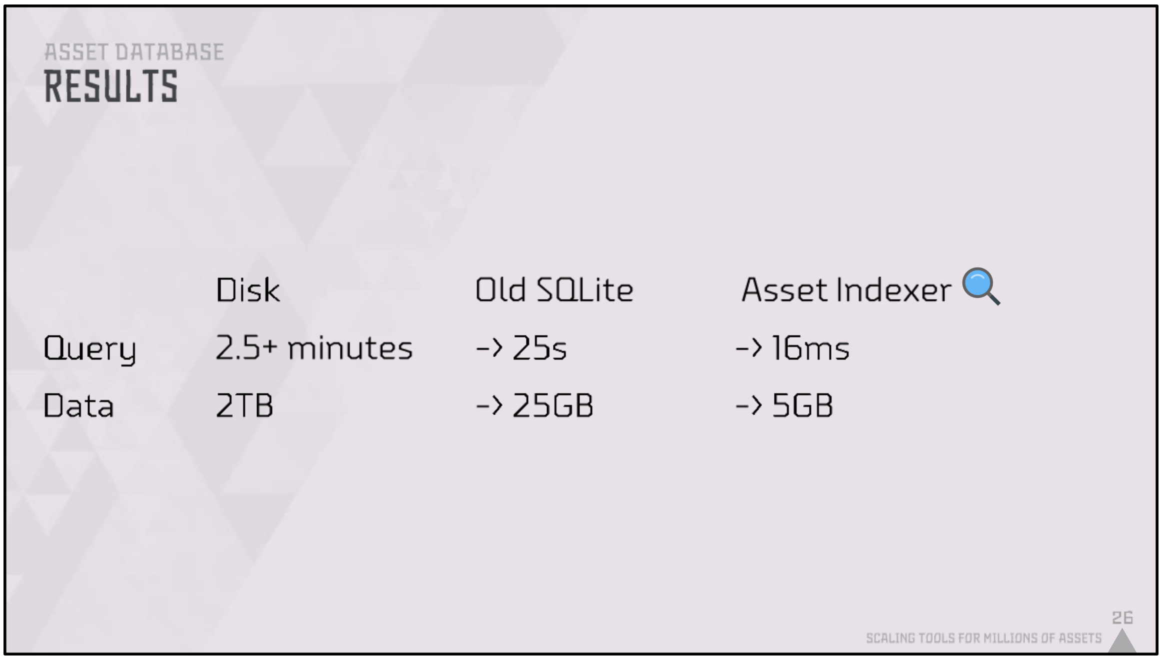
21

SCALING TOOLS FOR MILLIONS OF ASSETS

For that we have our editor. This is a basic view to inspect any CoreText file. *CLICK* Objects are represented as nodes, and links between objects as lines. PreviouslyI've only shown 1 object per file, but this can be freely structured in a manner that makes sense for the specific content, such as grouping all the Textures in the same file with the EnvironmentProbe. *CLICK* All of the string values are now displayed in an attribute editor that will show specialized fields such as numbers or dropdowns depending on the type. *CLICKS* And the links to the Texture objects are even displayed with a preview of the data.

To recap, every CoreText file has a number of objects in it. And every object has a Type, UUID, Name, and potentially some links all of which are indexed. As well as type-specific attributes which are not indexed.

Bringing it back to the Asset Indexer, the data that is common between all types is exactly what it indexes. We can efficiently store this data in a table of files and a table of objects. Because the CoreText format is valid for any type in the engine, someone can re-write the entire light reflection or entity system, and it still fits in this database format. This means we can create incredibly optimized tools and datastructures that apply to all content in the entire game.



ASSET DATABASE

RESULTS

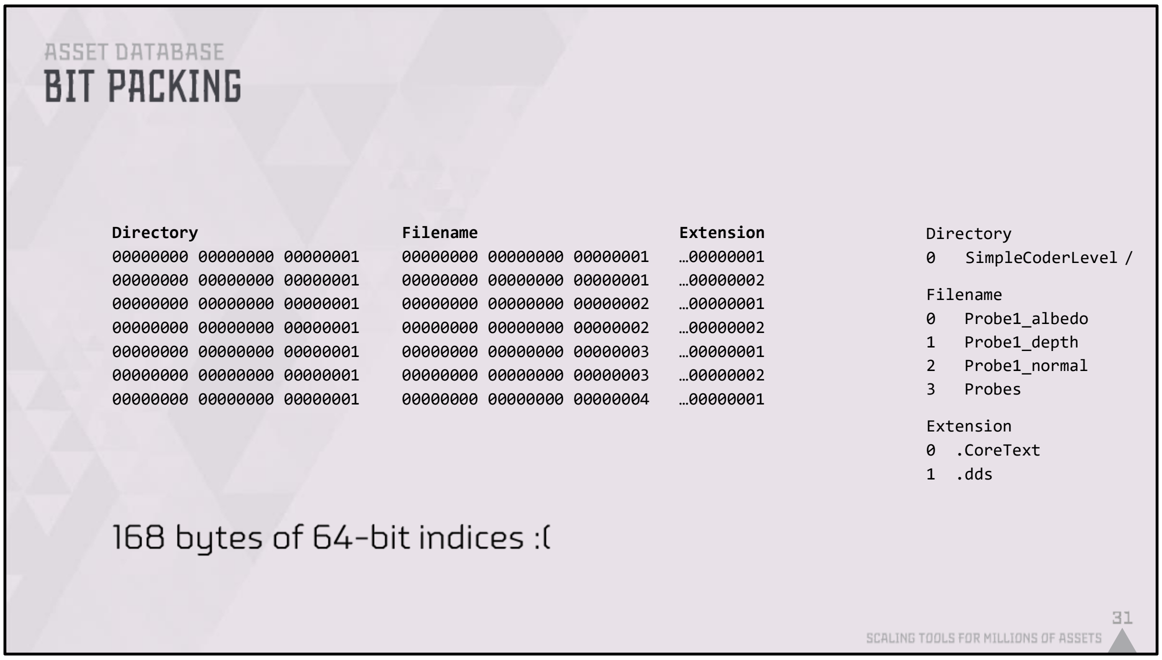
	Disk	Old SQLite	Asset Indexer 🔍
Query	2.5+ minutes	-> 25s	-> 16ms
Data	2TB	-> 25GB	-> 5GB

26

SCALING TOOLS FOR MILLIONS OF ASSETS

So with all 4 million files, 100 million objects, and 200 million links indexed, let's see what kind of queries we can answer with it.
CLICK For example, let's try to find all the EnvironmentProbes in our Game_Assets folder. In 2.8ms we find all 3721 Environment Probes.
CLICK Or let's find all objects that have Main Quest 2 in the object and file name. This takes a little bit longer to compare strings, but 16ms it found 17000 of them.
CLICK Maybe for one of these results we want to know if it's referenced at all. This is where our custom database comes in huge, as it has an acceleration table that can answer that its referenced once in 6 MICROseconds.

This is orders of magnitude faster than the old system.
A common operation like "query all objects with MQ02 in the name" would take at least 2.5 minutes if we were to open each file. With our old SQLite solution this was taking in the order of 10s of seconds. But with the Asset Indexer that's down to 16ms. And that's a real-world example of something people to every day in the Asset Browser. This is the difference between someone walking away to get a coffee vs realtime interactivity. *CLICK* Not only is it much faster, the size of the database for



ASSET DATABASE

BIT PACKING

```
Directory                    Filename                     Extension
00000000 00000000 00000001   00000000 00000000 00000001   …00000001
00000000 00000000 00000001   00000000 00000000 00000001   …00000002
00000000 00000000 00000001   00000000 00000000 00000002   …00000001
00000000 00000000 00000001   00000000 00000000 00000002   …00000002
00000000 00000000 00000001   00000000 00000000 00000003   …00000001
00000000 00000000 00000001   00000000 00000000 00000003   …00000002
00000000 00000000 00000001   00000000 00000000 00000004   …00000001
```

```
Directory
0   SimpleCoderLevel /

Filename
0   Probe1_albedo
1   Probe1_depth
2   Probe1_normal
3   Probes

Extension
0  .CoreText
1  .dds
```

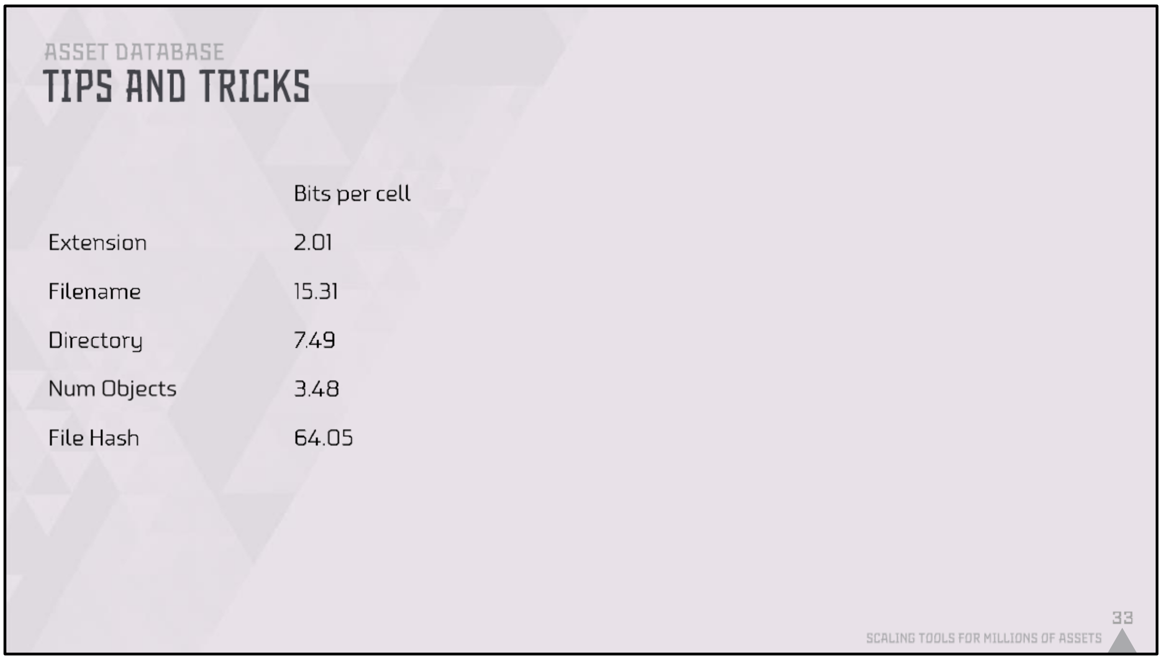
168 bytes of 64-bit indices :(

31

SCALING TOOLS FOR MILLIONS OF ASSETS

2TB of input data has shrunk from 25GB to 5GB. This means it can now be kept in memory, which one of the things that helps to achieve this high performance.
Let’s go over a few of the interesting things we did on the AssetIndexer.

For achieving the small size, we split path components into different strings pools and bit pack the data. Let’s look at an example to explain that.
Here we have the files from earlier. *CLICK* As you can see, all these files share the same directory, and there are only 2 unique extensions.
So we can save on memory by deduplicating these *CLICK* and putting them in string pools. Each pool contains unique strings.
And each cell just contains an index into one of these pools. In this example that reduces the total number of characters from 271 to just 81. That saves 70%!
But that’s not entirely fair, because we also need to store these indices. To handle all types of larger data, the database use 64 bit integers, but that means we are now spending 168 bytes on indices. Almost all our savings are gone again! So this is where the bitpacking comes in.



ASSET DATABASE

TIPS AND TRICKS

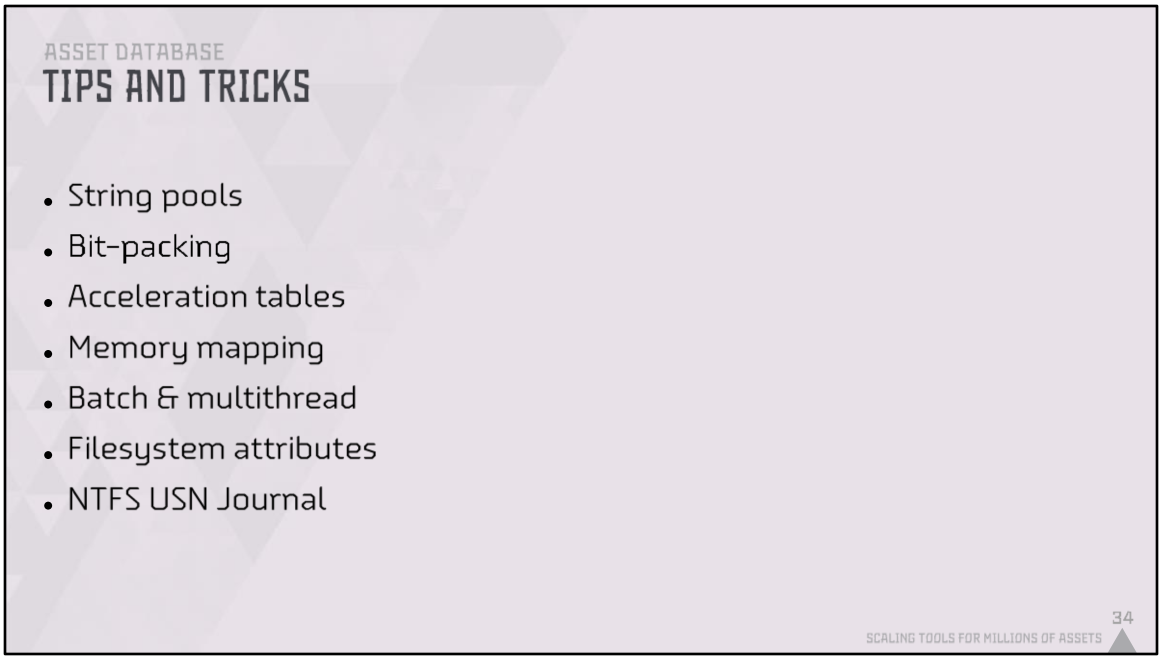
	Bits per cell
Extension	2.01
Filename	15.31
Directory	7.49
Num Objects	3.48
File Hash	64.05

33

SCALING TOOLS FOR MILLIONS OF ASSETS

By grouping several cells in a column together, we can determine the optimal packing. In this example all files are in the same directory, *CLICK* so we can simply store a single bit to index the directory for all cells. *CLICK* The file name has more variation, so every cell needs 2 bits to index its filename. *CLICK* For the extension we only need 1 bit per cell. *CLICK* This means in total we only need only 3 bytes to store all the indices!

The actual algorithm that determines this packing is much more complicated. It stores data in pages of cells, where a page can have a lookup table and offsets within a particular range, but the basic principle stays the same.
And this is fairly representative of the real world, the extensions really reduce to only 2 bits per cell!
It’s worth noting that this is very dependent on the variation in the data. We’ve laid out the database so it packs well, but some data doesn’t compress at all. A 64-bit hash is not going to bit-pack to anything smaller, unless we had a bunch of duplicated files or a bad hash.


ASSET DATABASE

TIPS AND TRICKS

- String pools
- Bit-packing
- Acceleration tables
- Memory mapping
- Batch & multithread
- Filesystem attributes
- NTFS USN Journal

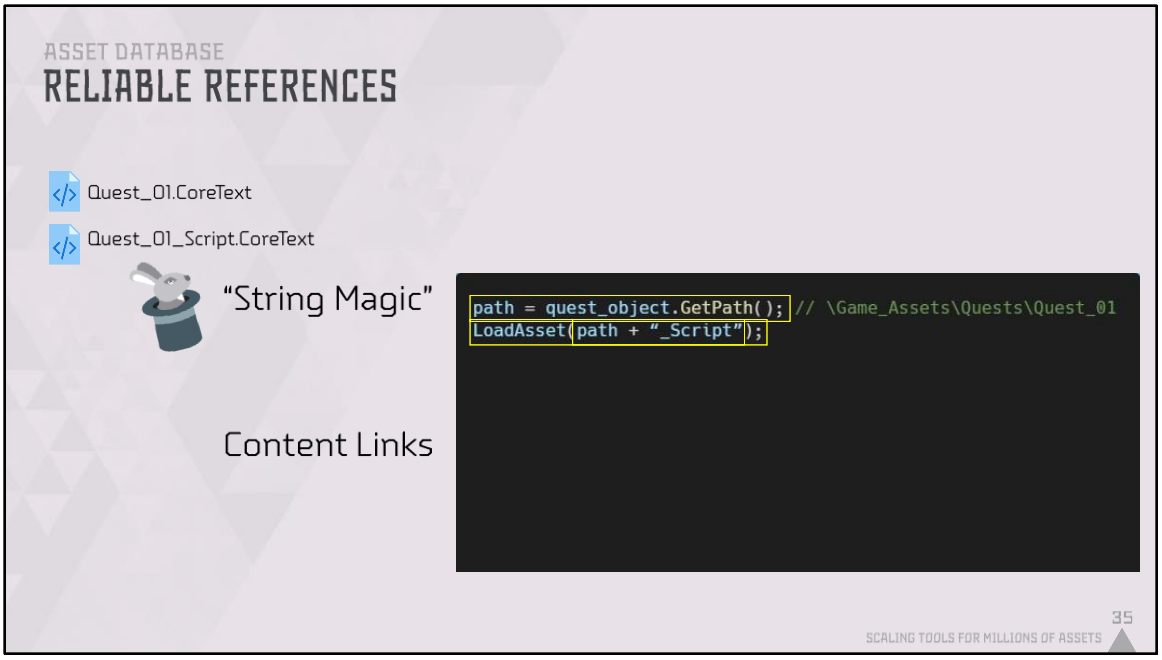
34

SCALING TOOLS FOR MILLIONS OF ASSETS

So string pools and bit-packing are a big part of what makes the database so compact.

Briefly going over some of the other things we did:

- Because we have complete control over our database, we can make custom acceleration tables specific for our dataset like I showed earlier with getting incoming links.
- Memory mapped the database into tools, much faster than sending data between processes using TCP/NamedPipes. I'll come back to this later on.
- Implement all queries as batched and multi-threaded by default
- Store the filesystem attributes such as modtime, filesize, and a hash. So we don't need to access the disk for that.
- For monitoring changes on disk, we previously used Windows' ReadDirectoryChanges API. But have since switched to using the FileSystem's NTFS USN Journal directly as it much more reliable. It allows us to check all disk I/O has been processed by the database and to replay any activity that happened while the AssetIndexer wasn't running without having to re-scan the entire project.



RELIABLE REFERENCES

Quest_01.CoreText

Quest_01_Script.CoreText

"String Magic"

Content Links

```
path = quest_object.GetPath(); // \Game_Assets\Quests\Quest_01
LoadAsset(path + "_Script");
```

One important mention on reliability is that we had to make sure that any code in our engine that loads an asset does so through links in content. Let's look at this fictional example to explain this.

Here are 2 files Quest_01.CoreText with the Quests name and description, and Quest_01_Script.CoreText with the scripted logic. *CLICK* Previously our code would get the *CLICK* path of the quest object,
CLICK add some special string to it, and then *CLICK* load that assets. The dependency between these 2 files is not expressed in the files themselves, but only deep inside the quest code.

CLICK We jokingly refer to this as String Magic, as to any outside code this object just magically appears. So any tool we make would be wrong or it would need to know about all exceptions like this.

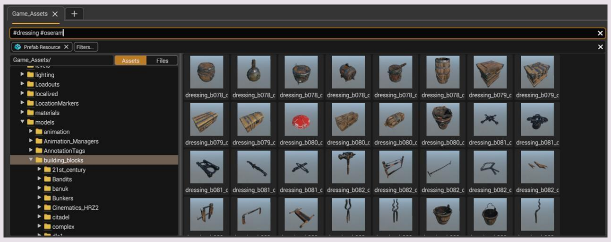
CLICK In our modern code, we create a link attribute on the Quest object that points to its script. The code can now simply load that link, and any generic tool will know that these 2 files are related.

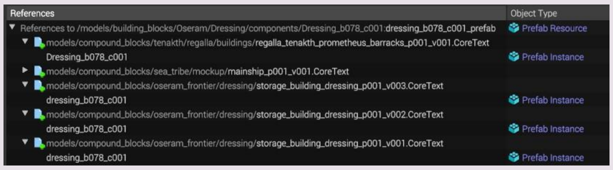
This was a big project to fix all the individual cases, but it’s vital in being able to base our entire engine and tooling on this concept.

ASSET DATABASE

USE CASES

- Asset Browser
- Reference Viewer
- More...?



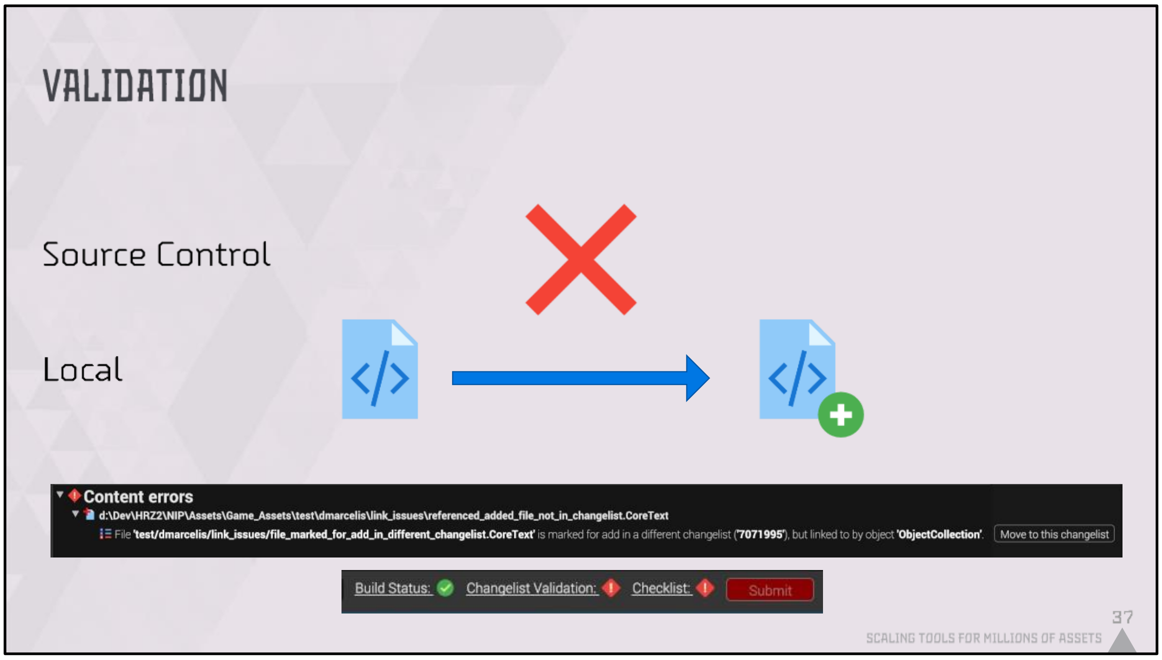


References	Object Type
References to /models/building_blocks/Oseram/Dressing/components/Dressing_b078_c001:dressing_b078_c001_prefab	Prefab Resource
models/compound_blocks/tenakth/regalla/buildings/regalla_tenakth_prometheus_barracks_p001_v001.CoreText	
Dressing_b078_c001	Prefab Instance
models/compound_blocks/sea_tribe/mockup/mainship_p001_v001.CoreText	
models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v003.CoreText	
dressing_b078_c001	Prefab Instance
models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v002.CoreText	
dressing_b078_c001	Prefab Instance
models/compound_blocks/oseram_frontier/dressing/storage_building_dressing_p001_v001.CoreText	
dressing_b078_c001	Prefab Instance

36
SCALING TOOLS FOR MILLIONS OF ASSETS

So, with a reliable and highly performance asset database we can now have an Asset Browser and Reference Viewer that are a joy to use.

But what other cool things can we do?

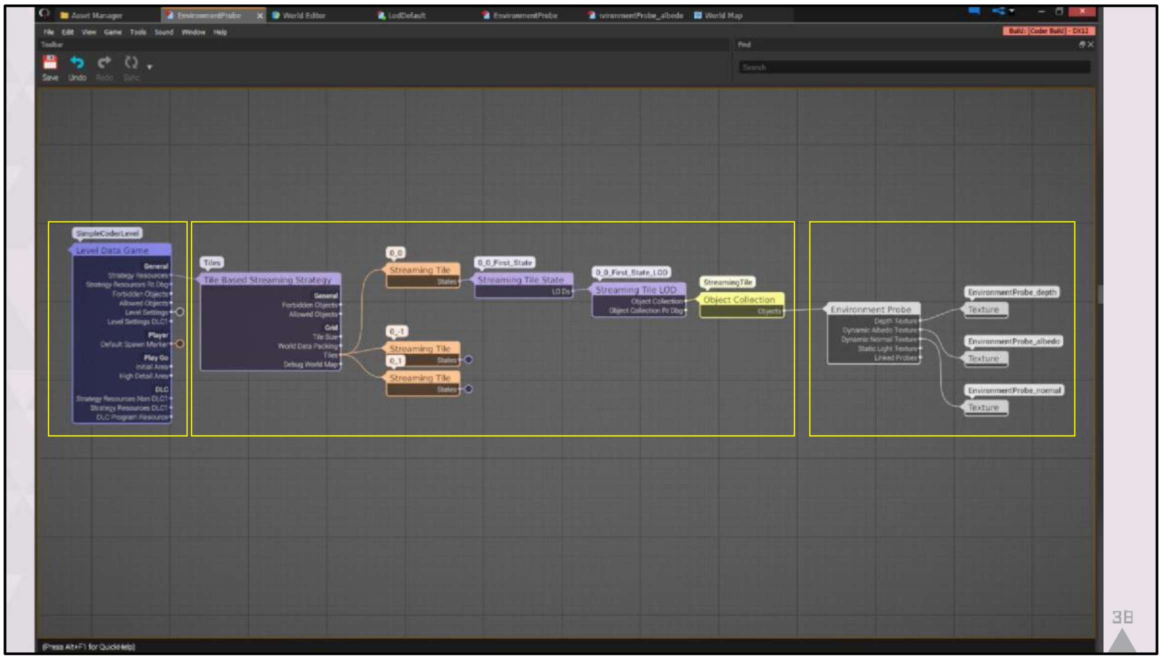


VALIDATION

```
Content errors
  d:\Dev\HRZ2\NIP\Assets\Game_Assets\test\dmarcelis\link_issues\referenced_added_file_not_in_changelist.CoreText
    File 'test/dmarcelis/link_issues/file_marked_for_add_in_different_changelist.CoreText' is marked for add in a different changelist ('7071995'), but linked to by object 'ObjectCollection'.  [Move to this changelist]
```

One scenario that all of you are probably familiar with is broken builds because someone forgot to submit a file. For us this was one of the most common build breakages. For example, someone adding a link to a new file, *CLICK* but only submitting the link and not the new file. Validating something like this could take minutes with the old system and wasn’t 100% reliable. So it was optional and only used by extra careful users.
CLICK With all this data available in the Asset Indexer, we can check for these cases in less than a second and have great confidence in the result. *CLICK* Because of this, we now enforce the validation and even block the user’s submit if we detect an issue. This has virtually eliminated this type of build failure. To the great joy of everyone, including the submitted who now doesn’t get an angry build master complaining that they broke the build.

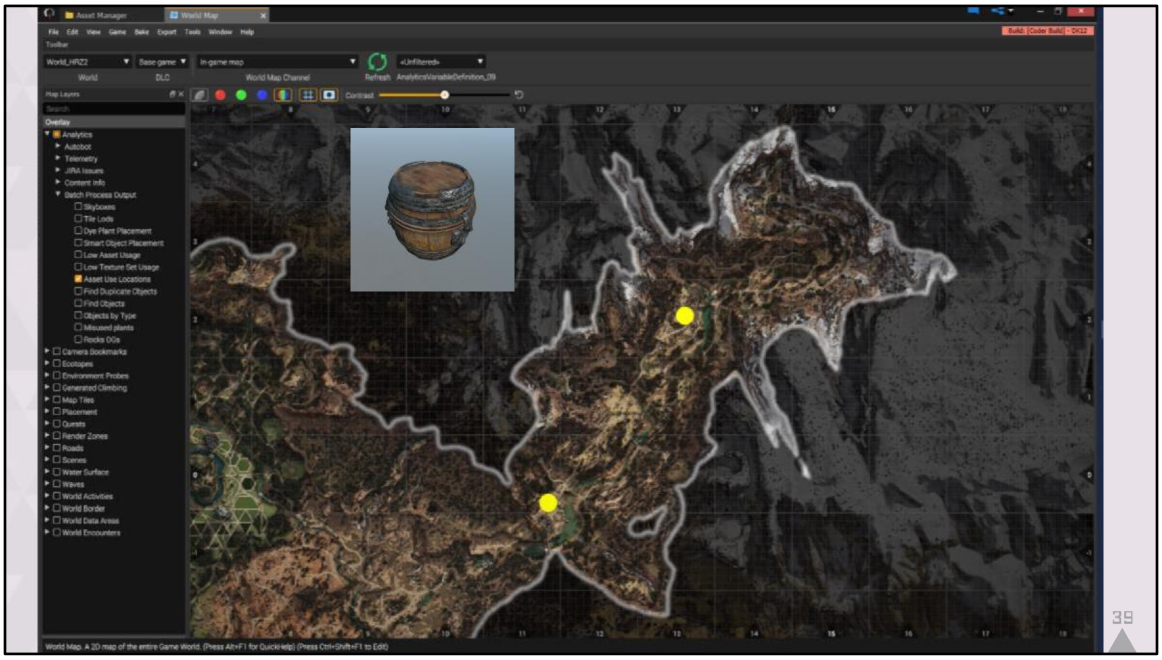
I must admit that we’ve gone a little overboard with the validation. It now does diffs of the files you’re submitting and even communicates with the Asset Indexer on a builder to check the if objects have been deleted at HEAD revision. But we realized it was possible, so we just had to do it. And it has really made everyone’s life better.



Taking this one step further, we can not only reason about the relationship between 2 individual objects, but about the entire game's content. We can find what level an object is linked to, as well as _how_ it is linked in.

CLICK Sticking with the previous example, here we have our trusty EnvironmentProbe and its Textures. *CLICK* This is linked into a Tile to only load the content that is directly around the player. *CLICK* And this tile is then linked into the Level.

CLICK By following the links of an object until we reach the Level, we can create the hierarchy of how an object is loaded in the game. This allow us to do rapid static analysis and validation of the content, answering questions like “do all tiles have EnvironmentProbes” or “which tiles contain a particular object”. All of this can be analyzed from just the AssetIndexer’s database without opening any CoreText file.



World Map. A 2D map of the entire Game World. (Press Alt+F1 for QuickHelp) (Press Ctrl+Shift+F1 to Edit)

We can then also take our map of the world, follow the link chain of each EnvironmentProbes and *CLICK* plot it on the map. This allows the lighting team to quickly check if an area is sufficiently covered.

Taking this even further, we can run more complex logic. *CLICK* For example to find expensive meshes that are only placed once in a specific area. Plotting this on the map, we can find single objects that have a heavy memory cost because their data isn’t shared with other instances.

CLICK Or we can plot that barrel from earlier! I happen to have picked an object for this example

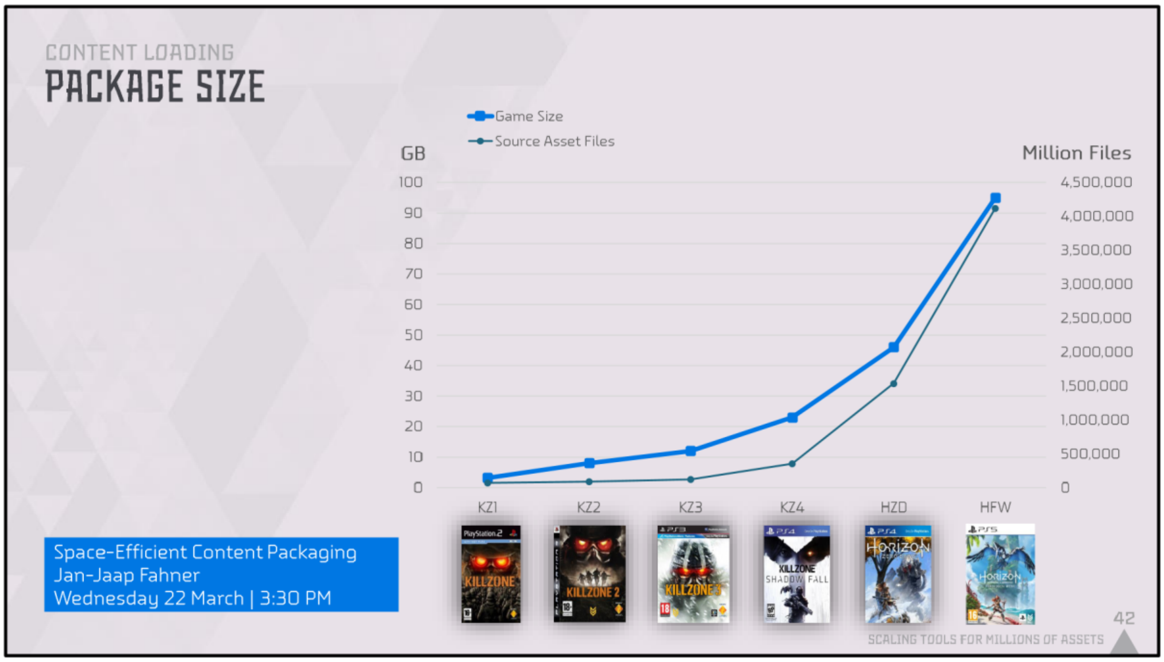
that only has 2 instances in this area, so maybe I should join Tech Art to help optimize content…

CONTENT LOADING

SCALING TOOLS FOR MILLIONS OF ASSETS

41

So in summary, we are super happy with this tech. It has sped up a lot of workflows and enabled us to do things that were previously impossible due to reliability or performance issues.
It's so successful that this tech has become the core of *all* of our development tools.
If an object or link isn't known to the AssetIndexer, it does not end up in the game.
Which brings us to the next section. So far, I've been talking about reasoning about assets. But another big part of development is loading the content into the game.



CONTENT LOADING

PACKAGE SIZE

Space-Efficient Content Packaging
Jan-Jaap Fahner
Wednesday 22 March | 3:30 PM

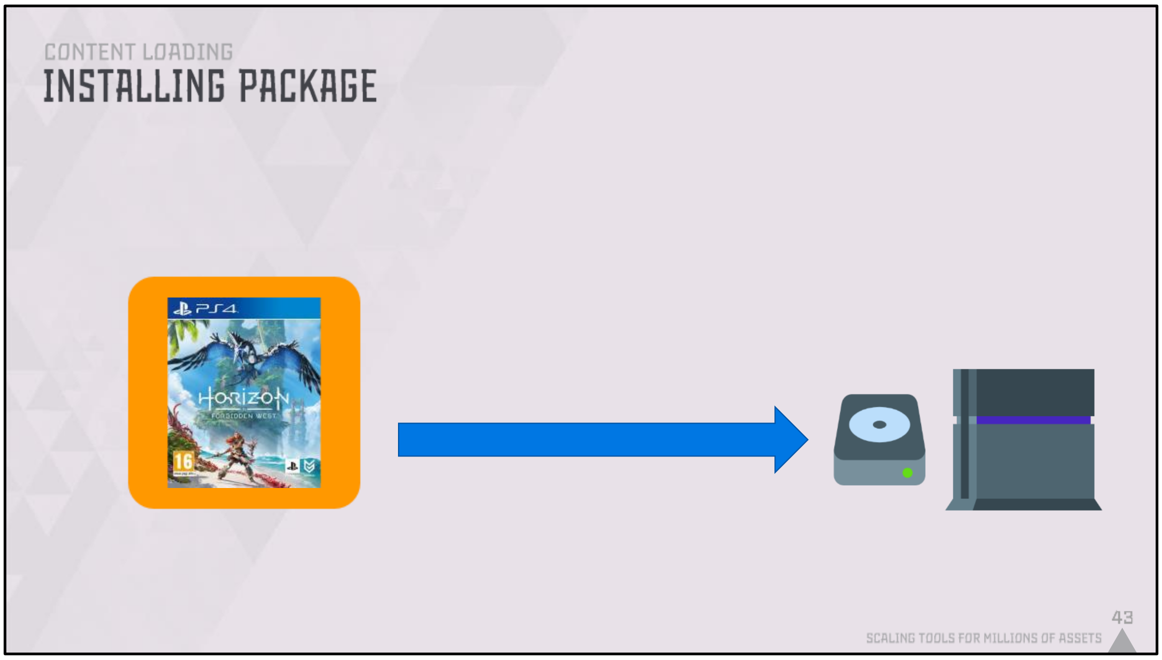
42

SCALING TOOLS FOR MILLIONS OF ASSETS

And here we have the other exponentially growing number: the package size. As I mentioned, it’s almost too big for a PS5 blu-ray. *CLICK* I’d highly recommend to check out Jan-Jaap’s talk later today about our completely new approach to packaging. But this talk is about tools. And that 95GB is the consumer package; the development package is even bigger as it also contains debug data.

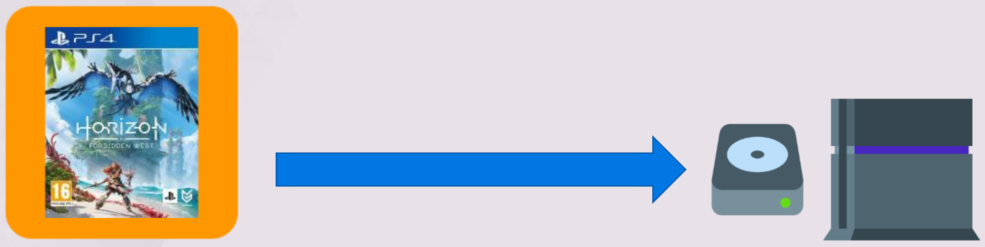
I will be explaining how we have completely redesigned how content gets loaded during development and can now start any build in 20

seconds.



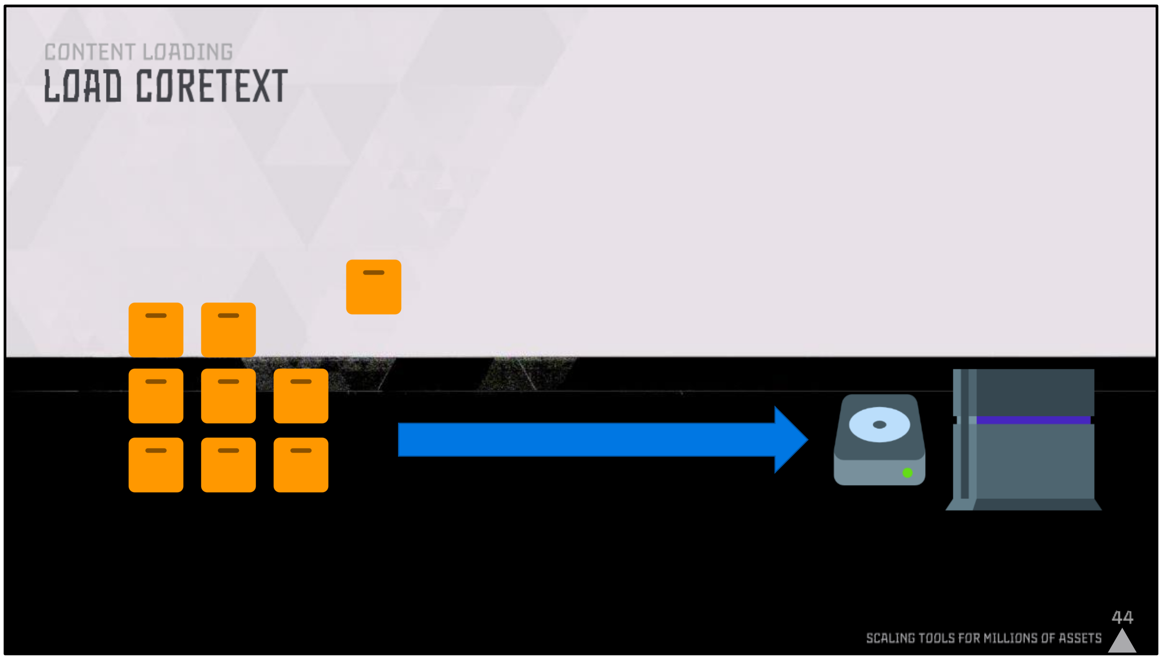
CONTENT LOADING

INSTALLING PACKAGE



43
SCALING TOOLS FOR MILLIONS OF ASSETS

But let’s start at the beginning and do a little history of this system starting from before Horizon Zero Dawn. Installing a package of over 100GB is absolutely terrible for your iteration time. In our case it takes about an hour to generate and another 30 minutes to install. Waiting 1.5 hours to test your changes on a devkit is not cool. So instead, we approached this differently.



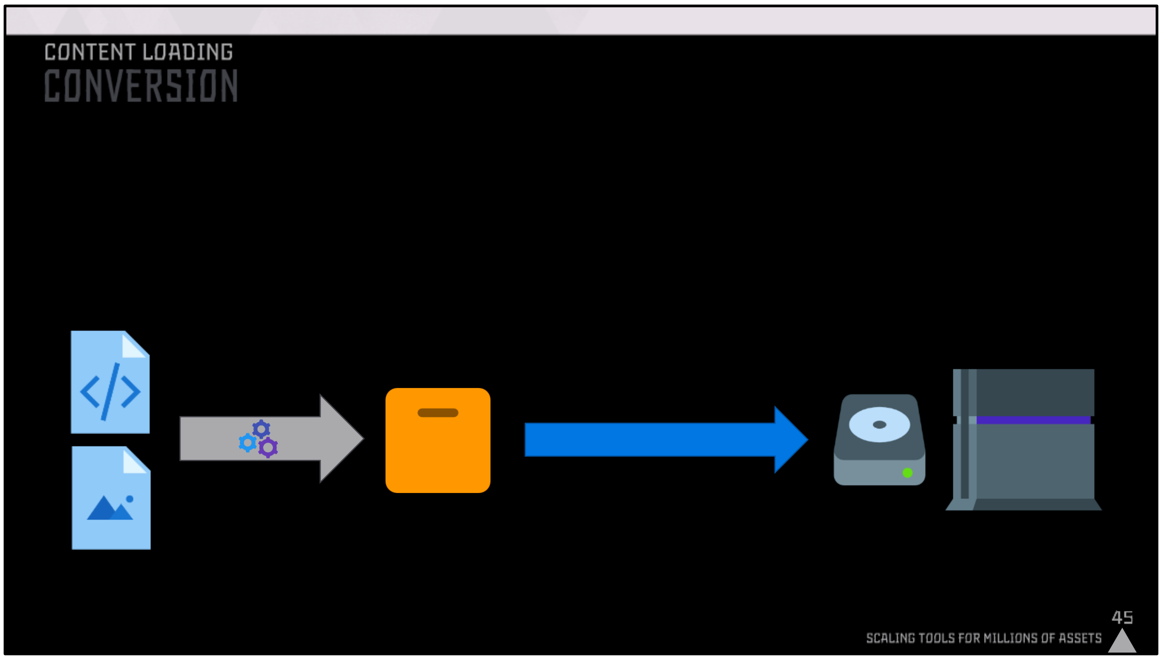
CONTENT LOADING

LOAD CORETEXT

44

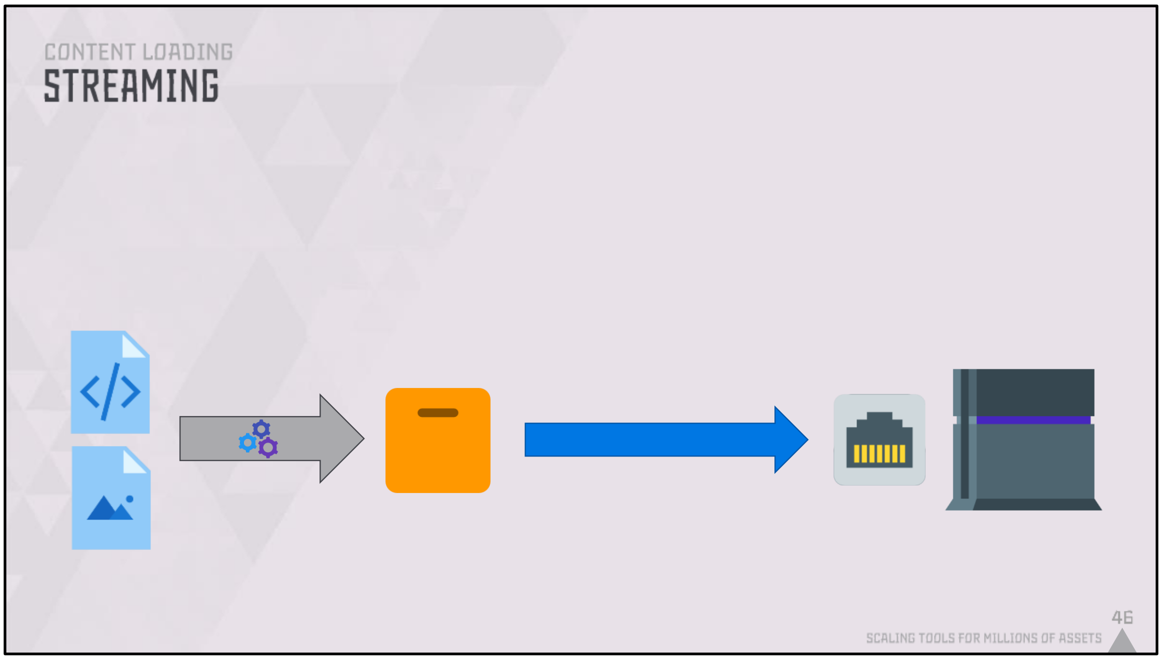
SCALING TOOLS FOR MILLIONS OF ASSETS

During development, the game didn't load from an installed package, but we loaded individual pieces of content. Iterating gets faster as you go for a finer and finer granularity. For us, that means we iterate at a per CoreText file.

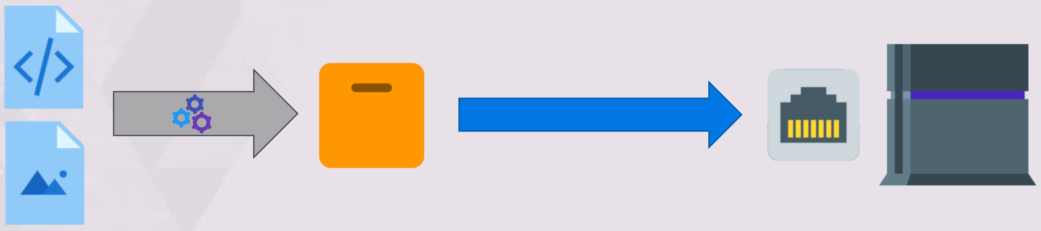


CONTENT LOADING CONVERSION

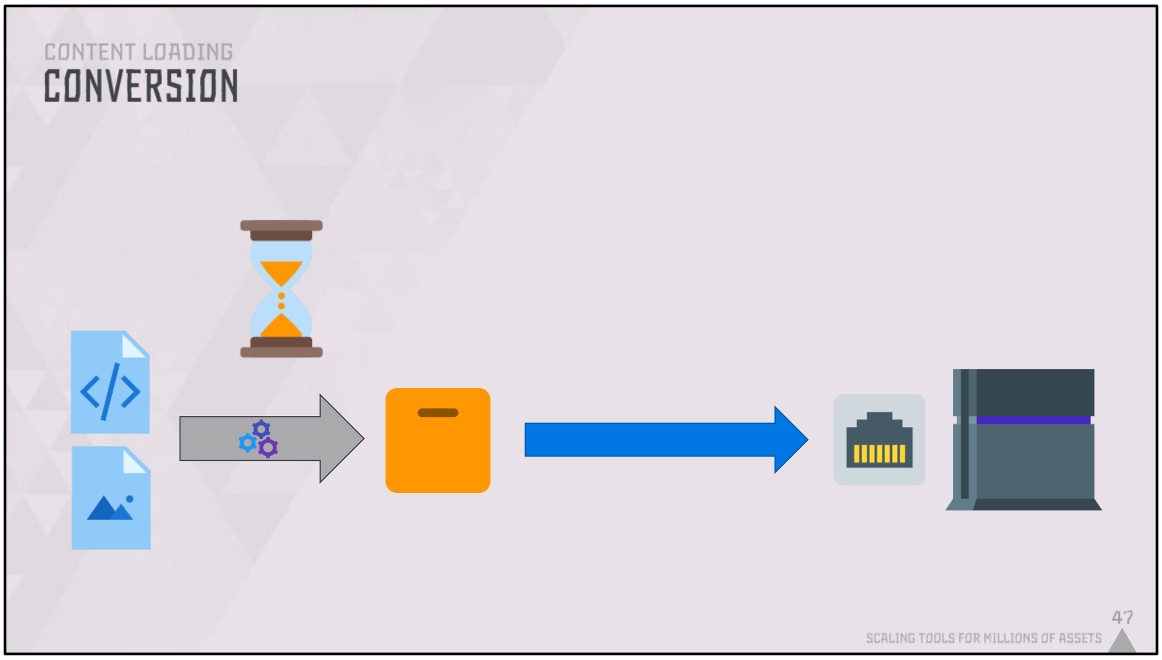
Make a change to your texture compression settings, and only that CoreText file gets reprocessed. As you might imagine, JSON-like text and png files are not very optimal for the runtime game, so there is a process in between that converts the source data into a binary format that the game reads.



CONTENT LOADING
STREAMING



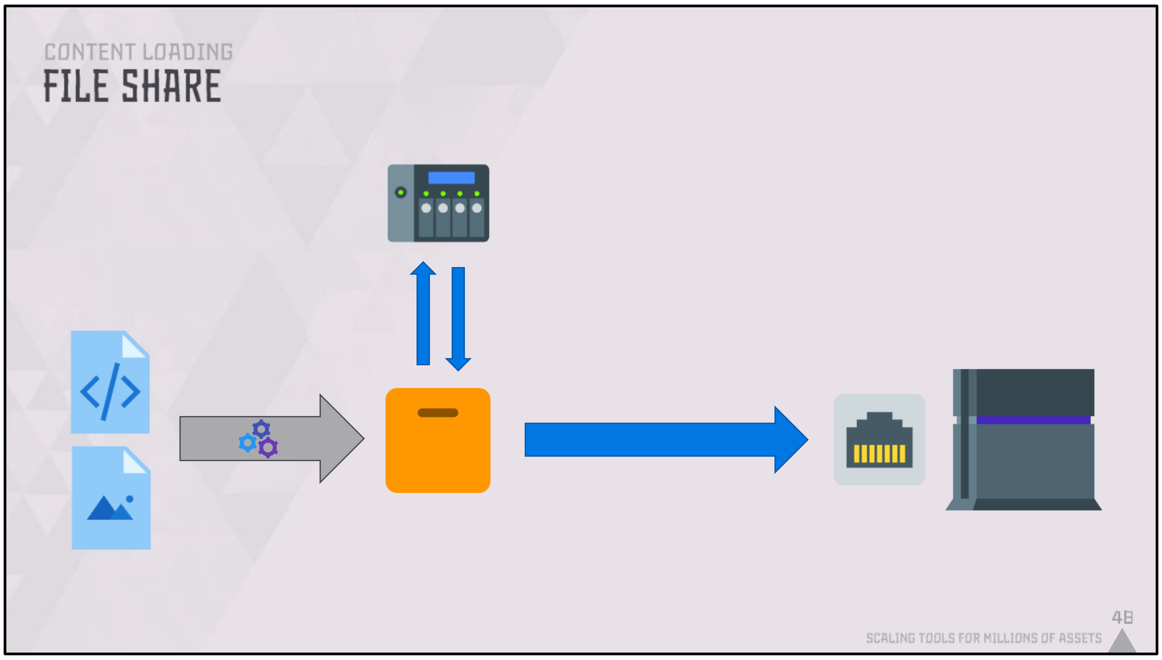
The PS4 devkits have an old fashioned spinning disk. Reading and writing to that is incredibly slow, *CLICK* so instead of copying those results to the disk, we streamed it from the workstations' SSD over the network to the game running on the devkit. This got us to 1Gbit per second network, which is the fastest the PS4 is going to get.



CONTENT LOADING

CONVERSION

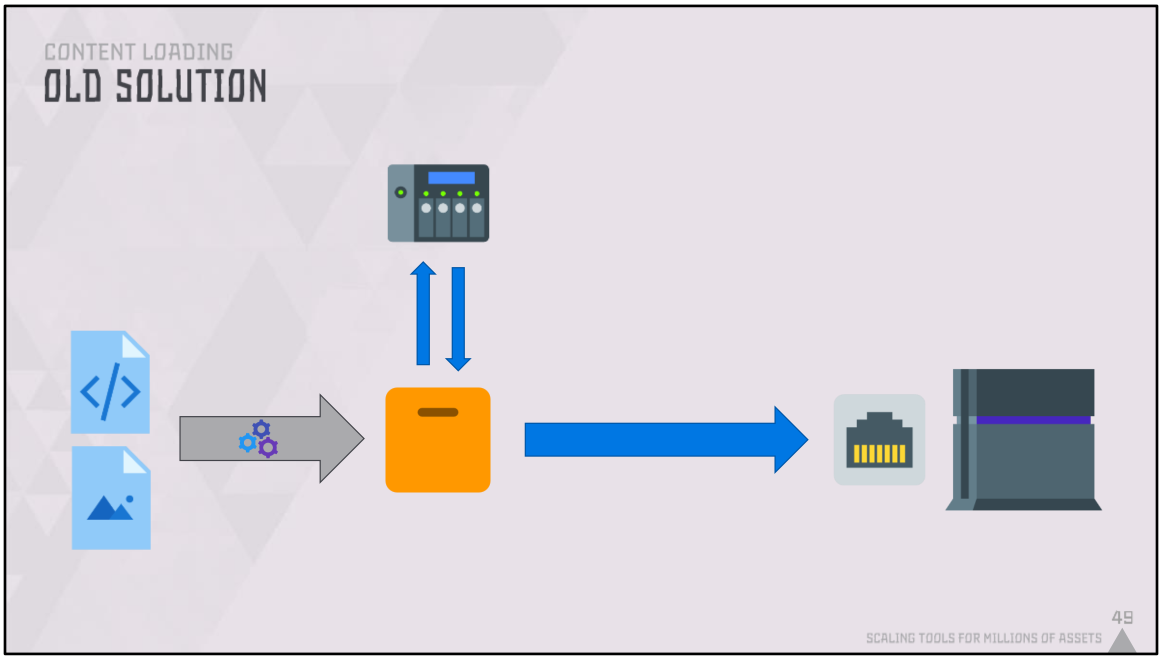
But we were still spending a lot of time on the workstation converting assets from CoreText into a binary format. This involves texture compression, shader compilation, audio compression, and more, which can all take a long time.



CONTENT LOADING

FILE SHARE

Most people are only editing a tiny portion of the content, so we could cache the results on a network share. The user’s workstation just downloaded the already converted content from there and then streamed it to the game.

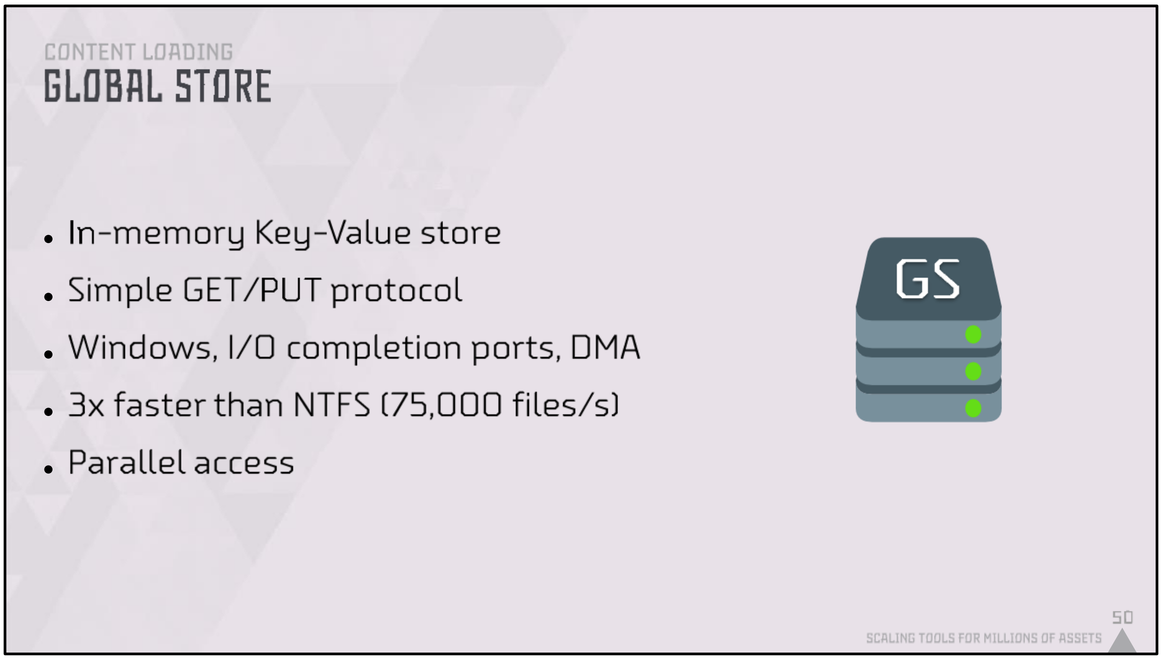


This is the state we were in during Horizon Zero Dawn. This is a perfectly functional solution, we're sharing all the heavy CPU work and caching the results, and not storing anything on the slow PS4 HDD.

But! This is still not as fast as we'd like it to be. After syncing to the latest revision, you'd spend 15 minutes downloading all the new converted content to your workstation before you could load the game. As I mentioned before, filesystems do not like many small files, and networked file shares are even

worse for this. Reading small files from our current SMB share only does about 700 files/s compared to the 25,000 of my local filesystem.

The network stack is perfectly capable of doing this, so we did some tests back in 2015 with different solutions. We tried SMB on Windows, Apache on Linux and dedicated hardware, but none of it met our expectations. So we did what we love to do in a situation like this, and rolled our own.



CONTENT LOADING

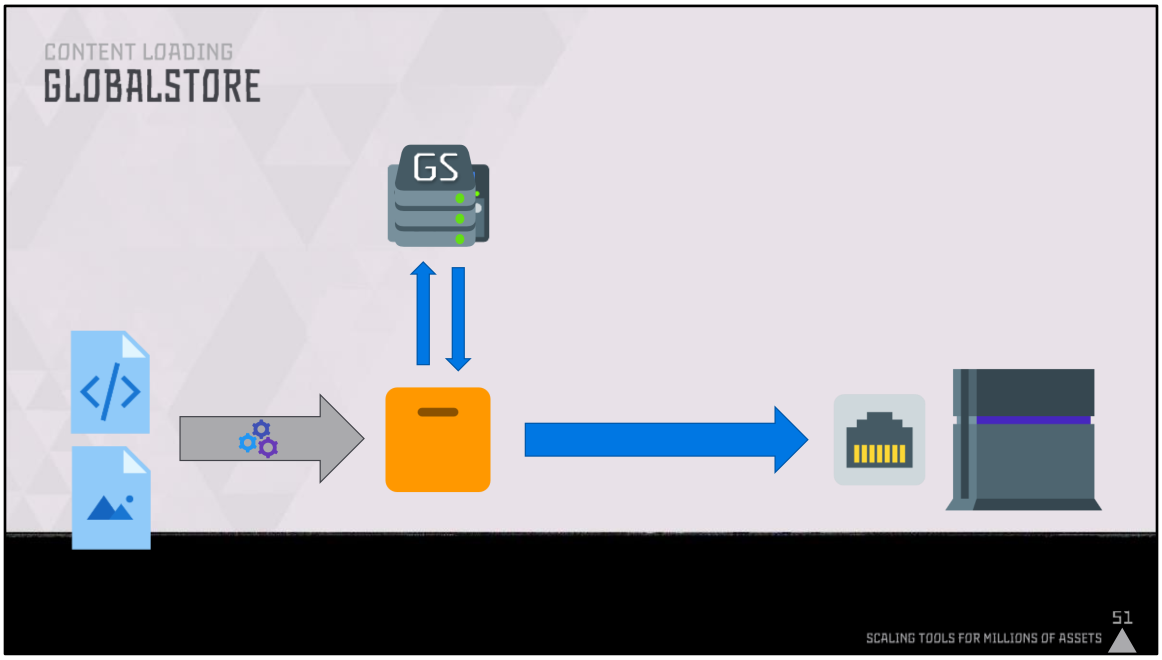
GLOBAL STORE

- In-memory Key-Value store
- Simple GET/PUT protocol
- Windows, I/O completion ports, DMA
- 3x faster than NTFS (75,000 files/s)
- Parallel access

50

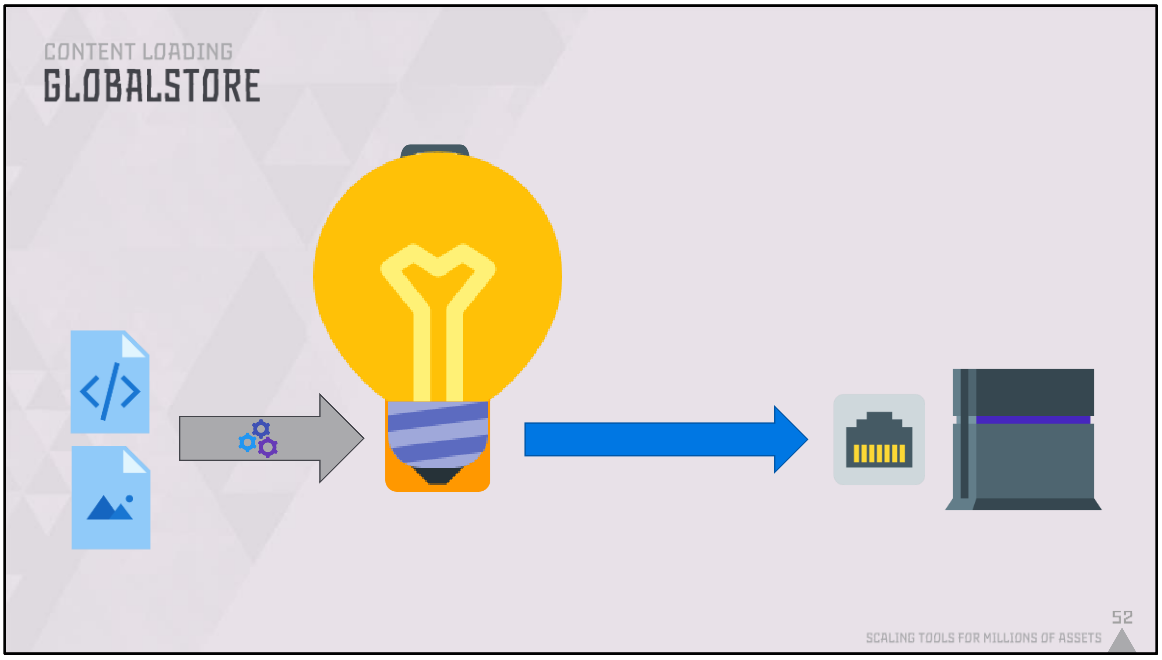
SCALING TOOLS FOR MILLIONS OF ASSETS

The GlobalStore is a key/value server that has large amounts of data in memory and is accessible through a simple get/put protocol. Think of this more as a network-attached hashmap than a file system. It runs on Windows and is heavily optimized with I/O completion ports and DirectMemoryAccess to transfer data to the network through pipelined requests. For small reads it's about 3x faster than NTFS on an NVMe. The GS manages 75,000 files/s compared to the 25,000 of my local disk. And that is just on a single thread, this system is designed for highly parallel access.


CONTENT LOADING

GLOBALSTORE

CLICK Replacing SMB with the GlobalStore for our cache was much better. For typical access the GlobalStore is 100x faster than the SMB share. This is great! But we're now still spending a bunch of time downloading all the converted content after every sync, and all of this time is again the OS overhead…

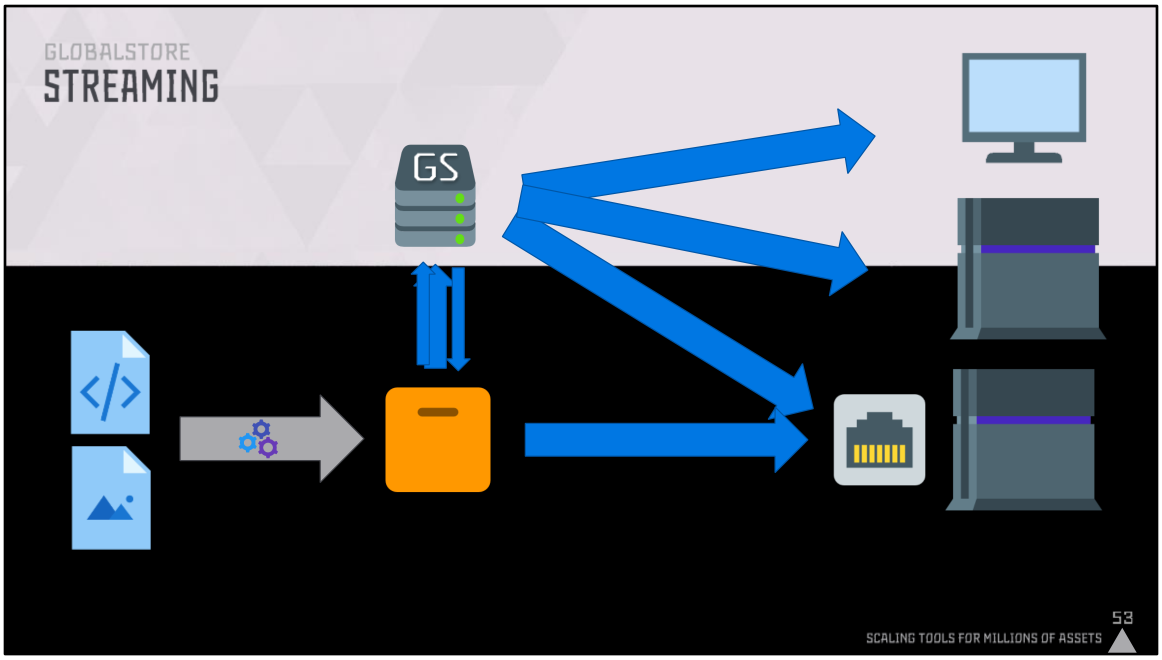

CONTENT LOADING

GLOBALSTORE

52

SCALING TOOLS FOR MILLIONS OF ASSETS

And this is where the lightbulb turned on. We have a server that is capable of serving millions of requests per second, and the game is already streaming all of the data over the network….


GLOBALSTORE

STREAMING

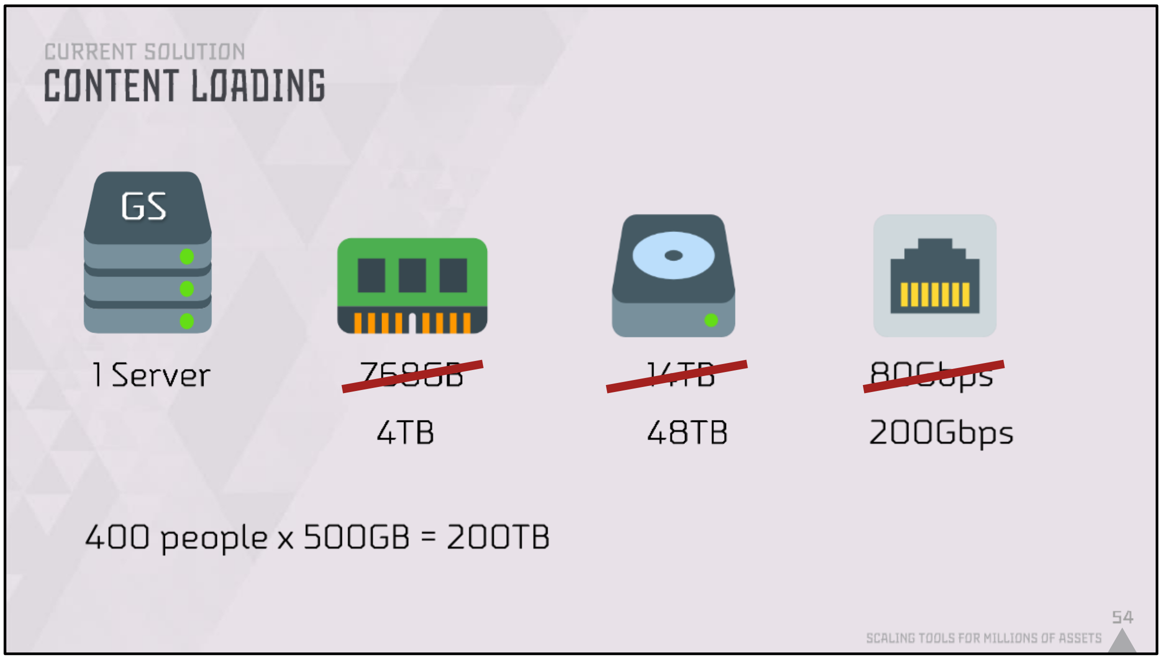
53

SCALING TOOLS FOR MILLIONS OF ASSETS

What if we just cut out the middleman?

When content is converted, it does not get written to disk, but gets uploaded to the GlobalStore. A process on the workstation tells the game what to load, and the game loads everything over the network from this server. This is the
system we now use everywhere and this is working very well for us. *CLICK* This one server

provides data to all devkits, workstations, and build machine. The user no longer needs to wait around 20 minutes while all the data gets cached locally before they can start the game. And loading itself even get 10-20% faster.



CURRENT SOLUTION

CONTENT LOADING

1 Server	~~768GB~~	~~14TB~~	~~80Gbps~~
	4TB	48TB	200Gbps

400 people x 500GB = 200TB

54

SCALING TOOLS FOR MILLIONS OF ASSETS

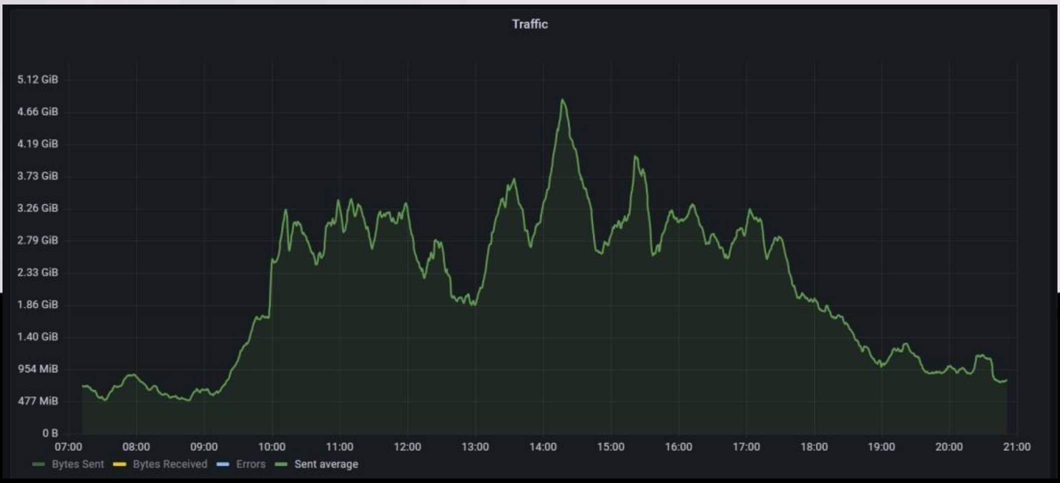
For Forbidden West we had a single server with 768GB RAM, 14TB of storage, and 80Gbps network that was serving around 350 people.

Another way of looking at this, is that instead of duplicating all converted content to every devkit and workstation, and having to invest in an extra 500GB SSD for every developer, we invest in 1 big machine with ‘only’ 14TB of storage.

CLICK We have since upgraded the machine and are now running a staggering 4TB of RAM.

CURRENT SOLUTION

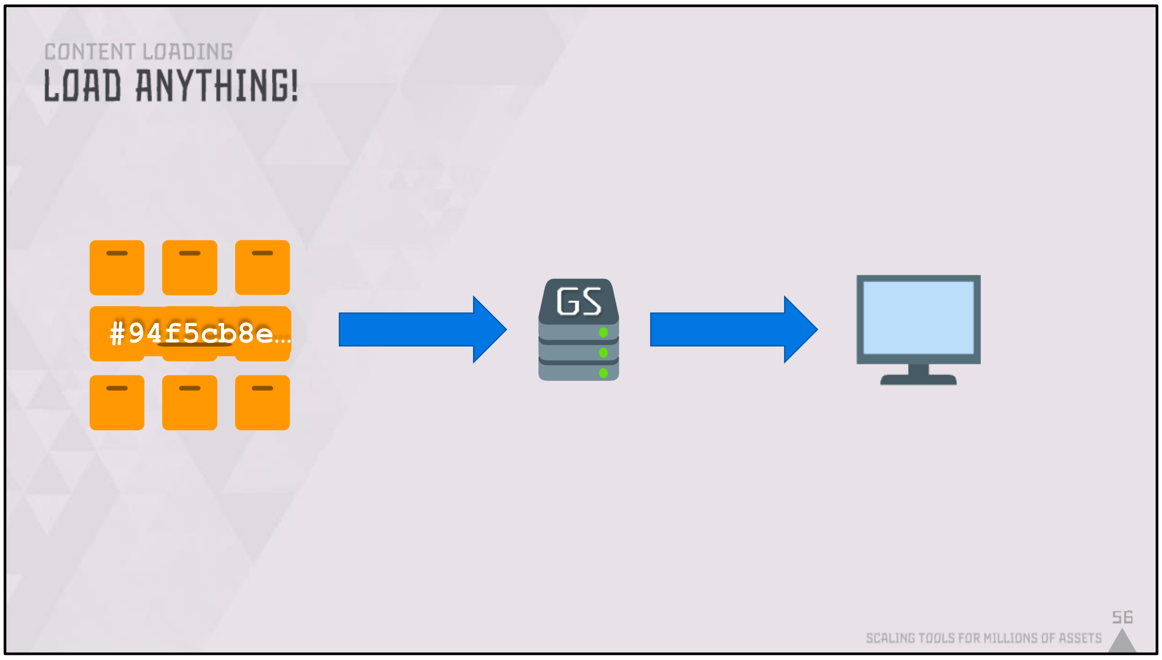
CONTENT LOADING



55

SCALING TOOLS FOR MILLIONS OF ASSETS

That single machine is able to serve the entire company and numerous build machines. Throughout the day we average around 2.5GB and 250K requests per second. As people come in to work, there is clear rise in traffic, with a dip around lunch and a roll off after 6pm. And it’s doing all of this with around 5% CPU load.



CONTENT LOADING

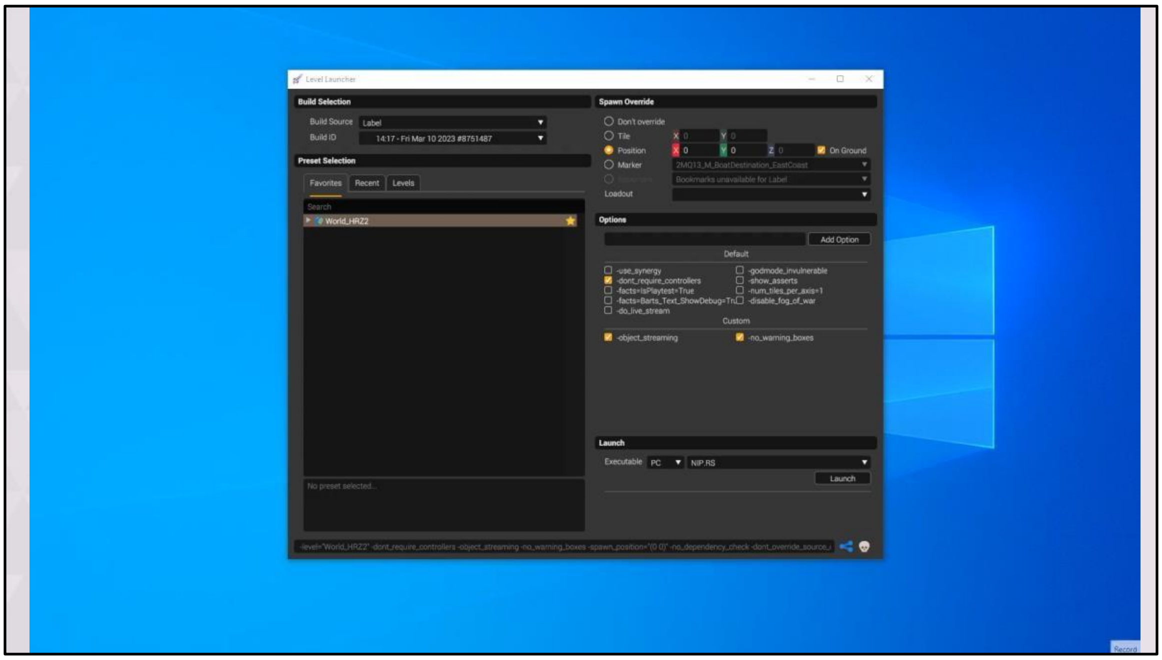
LOAD ANYTHING!

SCALING TOOLS FOR MILLIONS OF ASSETS 56

But this is where the fun begins! Now that we have a system where all data is streamed in from a central server, loading your local version of the game is no different from loading any other build.

A build is composed out of many assets. Every asset maps to a converted result in the Global Store. *CLICK* By creating a single mapping for all asset in a build and uploading this mapping itself to the Global Store, we can address the whole build with just a single identifier for that mapping.

Our build machines produces a new build every 10 minutes. You can load into any of these builds, just like that.



Level Launcher

Build Selection

Build Source: Label

Build ID: 14:17 - Fri Mar 10 2023 #8751487

Preset Selection

Favorites | Recent | Levels

Search

World_HRZ2 ★

No preset selected...

Spawn Override

- ○ Don't override
- ○ Tile X 0 Y 0
- ● Position X 0 Y 0 Z 0 ☑ On Ground
- ○ Marker 2MQ13_M_BoatDestination_EastCoast
- Bookmarks unavailable for Label

Loadout

Options

Add Option

Default

- ☐ -use_synergy
- ☑ -dont_require_controllers
- ☐ -facts=IsPlaytest=True
- ☐ -facts=Barts_Text_ShowDebug=Tru
- ☐ -do_live_stream
- ☐ -godmode_invulnerable
- ☐ -show_asserts
- ☐ -num_tiles_per_axis=1
- ☐ -disable_fog_of_war

Custom

- ☑ -object_streaming
- ☑ -no_warning_boxes

Launch

Executable: PC | NIP.RS

Launch

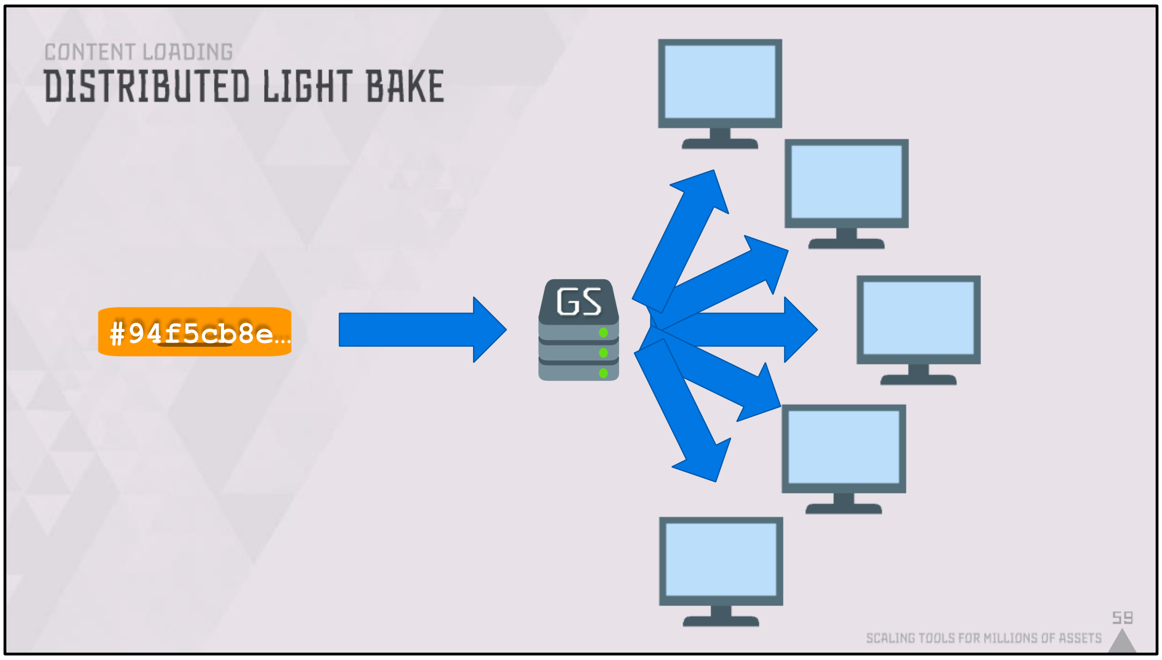
```
-level="World_HRZ2" -dont_require_controllers -object_streaming -no_warning_boxes -spawn_position="(0 0)" -no_dependency_check -dont_override_source_
```

To illustrate, here is our Level Launcher. Let's say you wanted to compare your changes against the same version without your changes. You would open the list of builds, select the revision you're synced to, and click launch. The launcher first downloads the executables, and then starts the game. You could also have picked a build from this morning or last week. Or a something a colleague is working on, which they can share with a simple link. All of this without having to sync or revert any of your changes.

And within 20 seconds we are loaded into something completely different from your local state.

Here's a little real-world anecdote to illustrate this:
After one of my test runs of this presentation, I was chatting with our lead technical producer. While chatting, he was using the level launcher to load into several

cinematics. During this conversation that took only about a minute he had managed to check 3 cinematics for performance and visual issues. When someone submitted a fix, he could select simply select a different build, and 20 seconds later he has verified the fix worked and can mark the item off his list. Seeing this, my thoughts were "you know, we could probably make this even faster.” But hey, at least it's already faster much faster than getting a coffee!

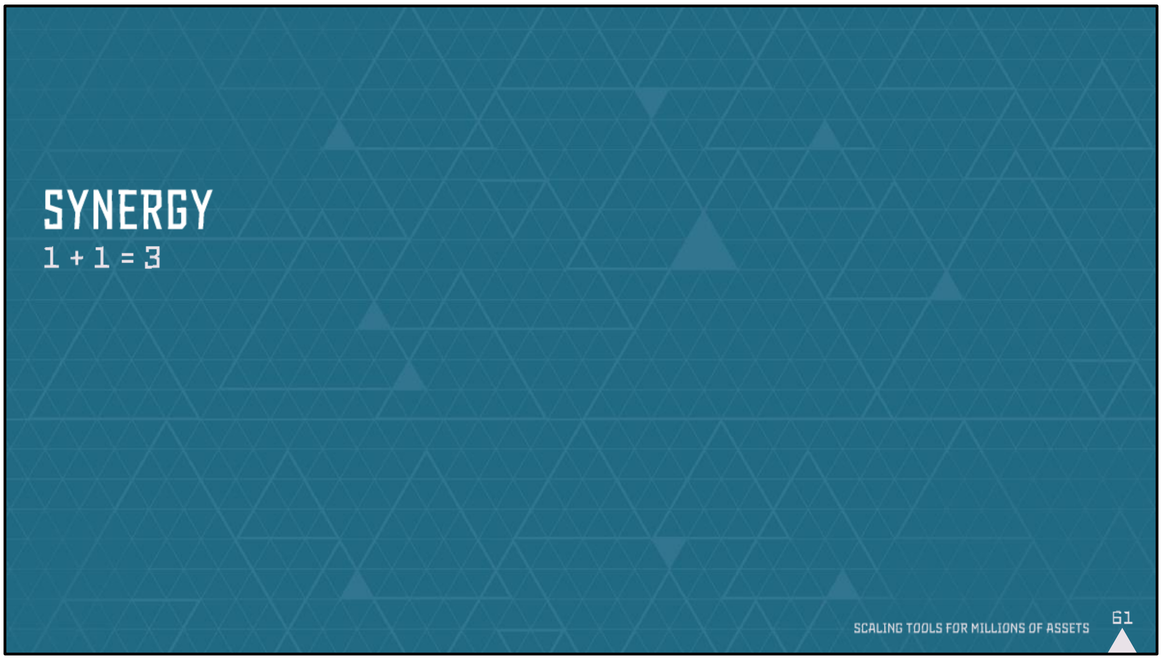


DISTRIBUTED LIGHT BAKE

Like I briefly mentioned, you can also share anything you are working on with your colleagues. By creating one of those identifiers for your local changes, you can create a link that can be shared with anyone in the company and with a single click, they can then launch your exact changes in 20 seconds. This is incredibly useful for getting feedback on a prototype or having QA test a change.
Another example of the power of this system is being able to distribute our light bakes. *CLICK* We can have every idle workstation in the entire office load into the same version of the game in just a few minutes.

The light bakes are an expensive process that requires running the full game logic, including GPU rendering. Previously our build machines would sync to a particular revision, unshelve someone's change, convert the content, and then start the lengthy bake process. This was putting significant stress on our server farm at the end of the project.

We have a lot of idle workstations during the night, but changing someone's sync state, reverting their changes, and applying someone's work is insanity and would definitely lead to problems. So with global store streaming, we can put a single



SYNERGY

1 + 1 = 3

SCALING TOOLS FOR MILLIONS OF ASSETS

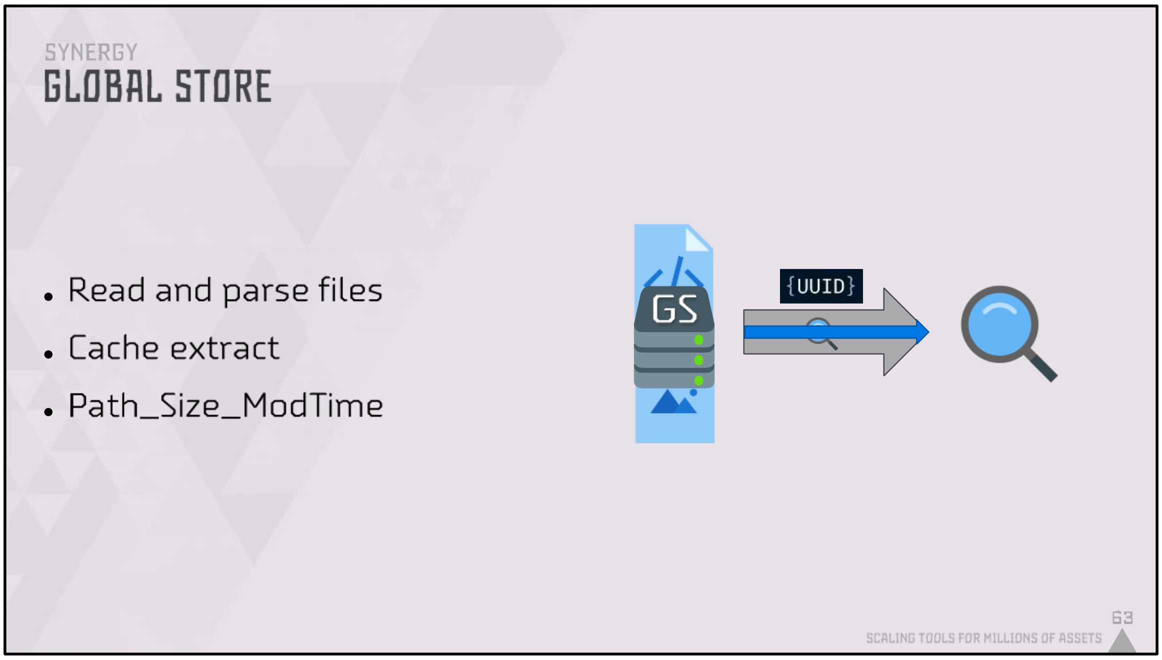
61

executable on their machine, and it simply streams in everything without having to change any of their local state.

This allows us to distribute the light bakes over all idle workstations, guaranteed they're baking the exact same thing without change users' state. This has unlocked a lot of compute power.
By going from monolithic packages to granular content loading, we have massively improved iteration times.

And by no longer storing duplicate data on every devkit or workstation, and instead streaming everything over the network, we have decoupled the size of the game from the data that you're working with.

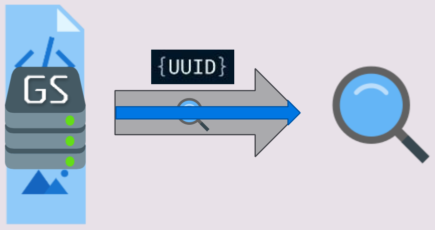
Loading your own local changes, last week's build, or the prototype your colleague has shared is all just as fast. Which enables completely new workflows.
We now have a content loading system that streams everything from a highperformance key value store, and an asset database that doesn't need to read any files. These systems synergize together in a quite a powerful way.



SYNERGY

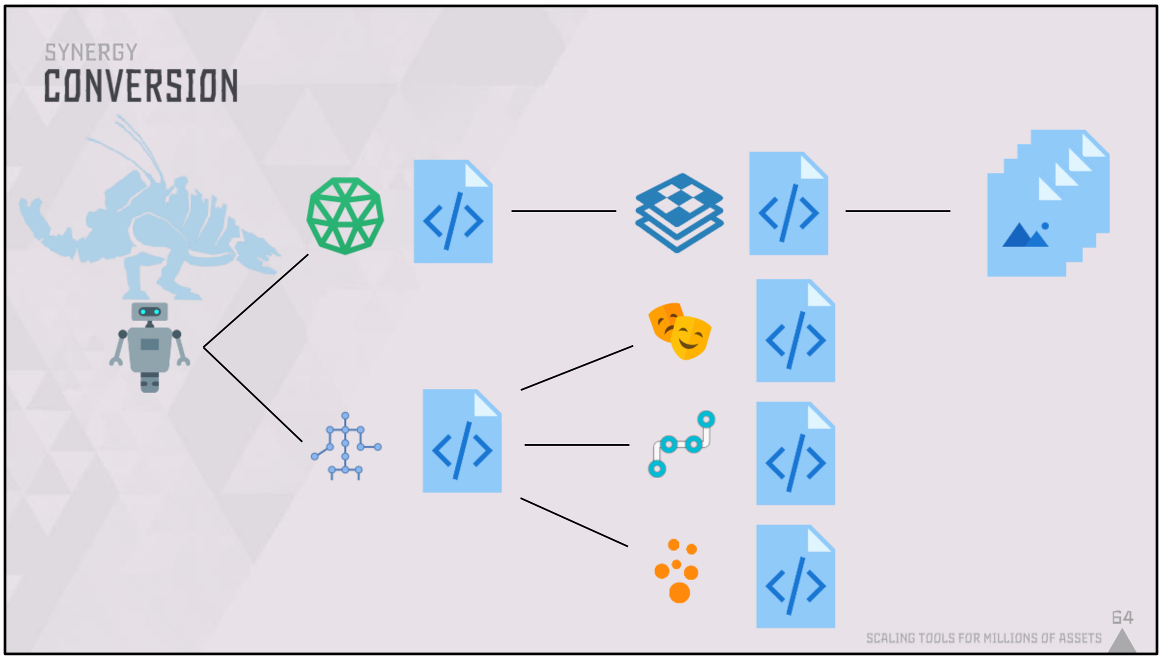
GLOBAL STORE

- Read and parse files
- Cache extract
- Path_Size_ModTime



As mentioned, we store our converted content in the GlobalStore. But that's just one example of the things we can cache. We also store executable artifacts from the build machines, so we don't need check them in to Perforce. Or a shader compiler cache which helps speed up our conversion. Because we have this high-performance store that is well integrating into our infrastructure, adding new caches is a just a few lines of code.
One great example of that is the AssetIndexer. It builds up its database from the files in Game Assets. To do this, it needs to read and parse all millions of files once to the extract the indexed data.

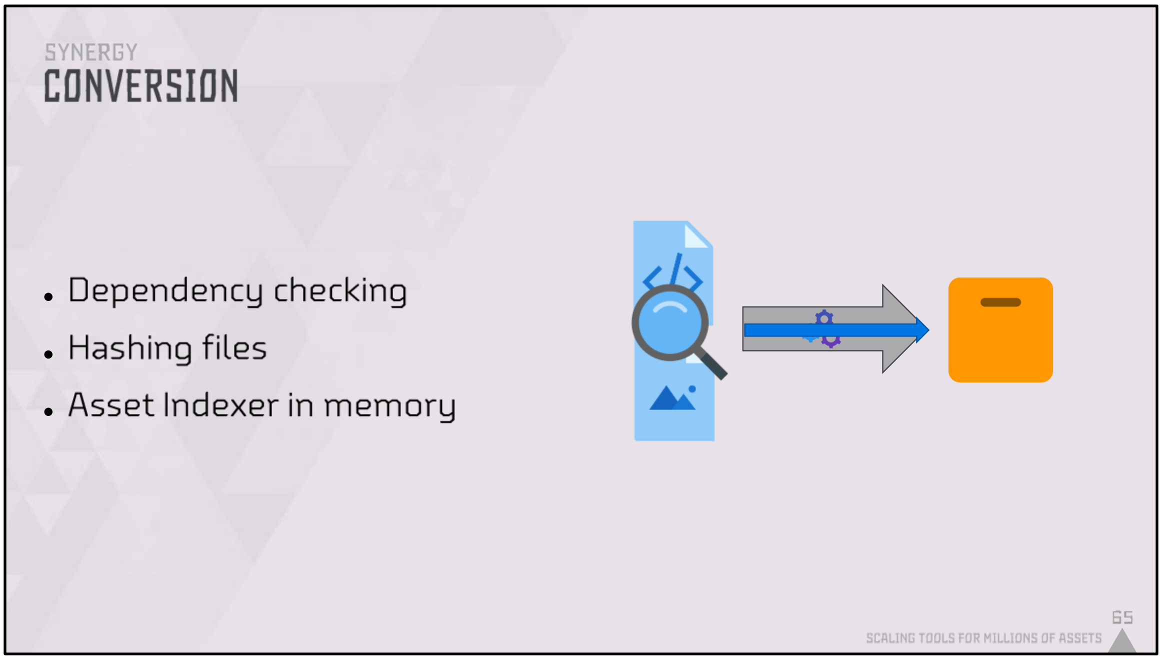
But every single workstation needs to do this and apart from a tiny number of local changes, they're all parsing the exact same files. *CLICK* We can store an extract of each file in the GlobalStore keyed on its metadata: file path, file size, and modification time. The AssetIndexer can then build its database by only listing files to get the metadata and downloading the extract from the GlobalStore, without even having to open any of the files. Because the GlobalStore is optimized for millions of small requests this is much faster than reading and parsing the files themselves. This means we can build up a database of every single file in the game in about 5 minutes, without opening any of them.



SYNERGY

CONVERSION

Another example of this synergy is during conversion. When we load something like a robot entity, *CLICK* we also need to load its mesh, textures, animations, etc. *CLICK* These might be stored in separate CoreText or binary files, so to find the converted results in the GlobalStore for all of these files, *CLICK* the convert process recursively follows the links between objects and files to create a graph of what needs to be loaded. This is a process we call dependency checking.



SYNERGY

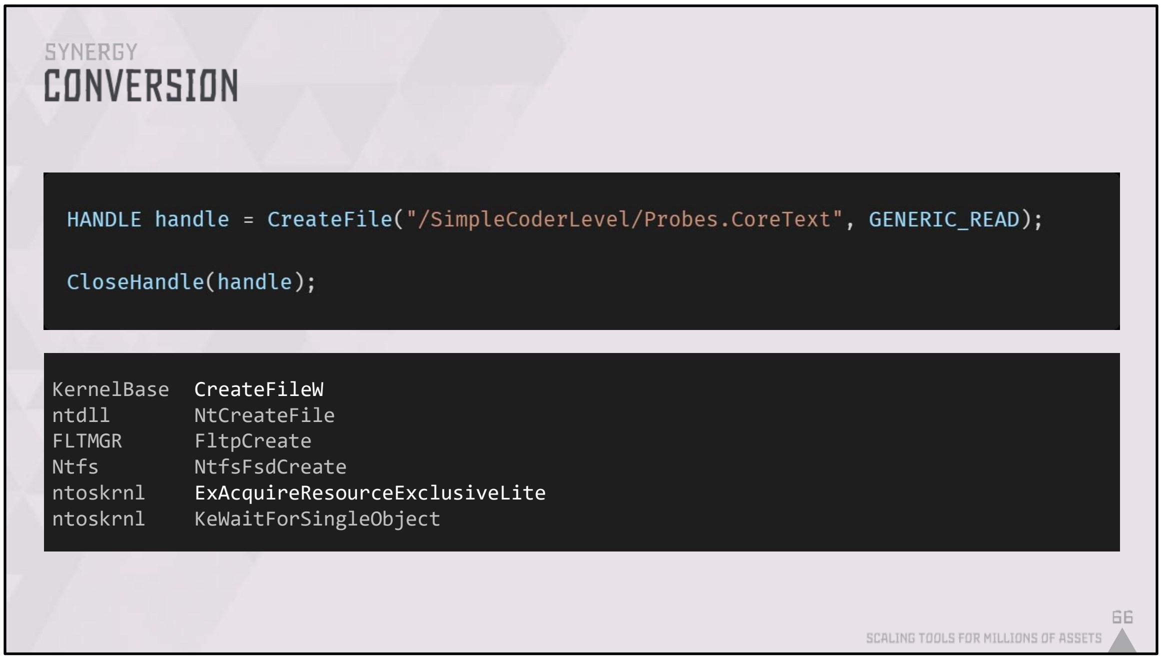
CONVERSION

- Dependency checking
- Hashing files
- Asset Indexer in memory

Previously we were reading the files from disk to figure out the link structure and to calculate hashes for comparison. This meant -you can probably guess by now- opening thousands of files every time the game tries to load something. *CLICK* But with

the AssetIndexer, we have all of that data available already in a memory-mapped database.

Let’s take this moment to dive a little deeper into why this is so much faster.



SYNERGY

CONVERSION

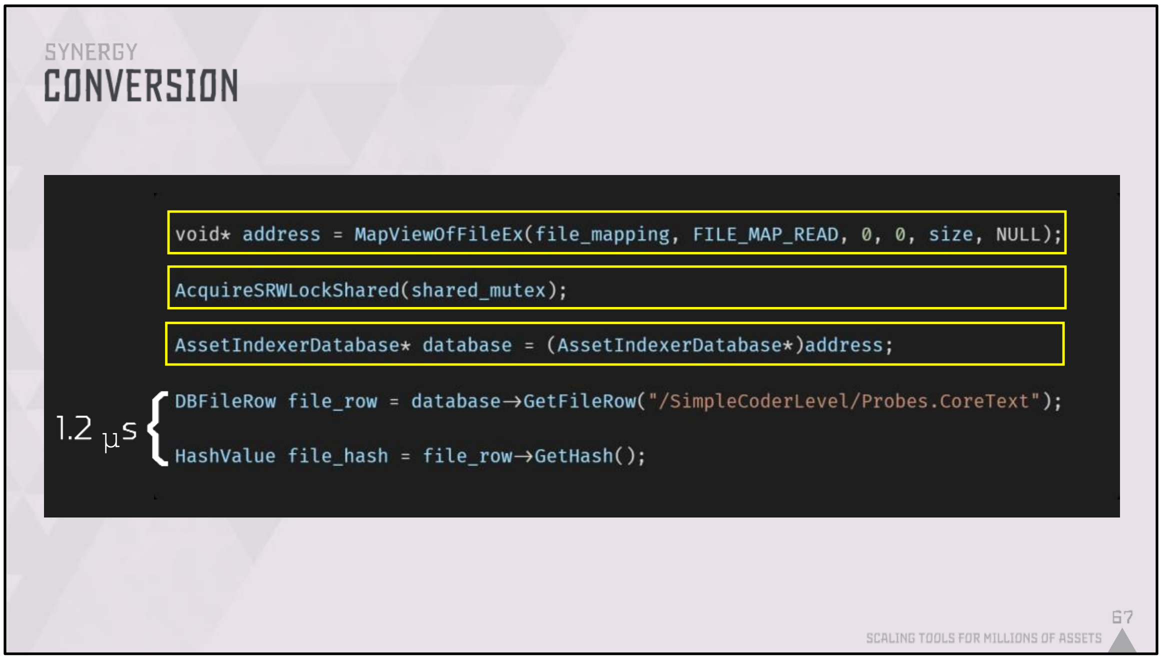
```
HANDLE handle = CreateFile("/SimpleCoderLevel/Probes.CoreText", GENERIC_READ);

CloseHandle(handle);
```

```
KernelBase  CreateFileW
ntdll       NtCreateFile
FLTMGR      FltpCreate
Ntfs        NtfsFsdCreate
ntoskrnl    ExAcquireResourceExclusiveLite
ntoskrnl    KeWaitForSingleObject
```

66

SCALING TOOLS FOR MILLIONS OF ASSETS

The problem with opening many files is that for every file, you need to do a kernel call to get the handle, and another kernel to close it again. Even without any other disk access the CPU needs to transfer into kernel mode and back into user mode twice for every file. The file system also has to provide guarantees about consistency of files between processes so most of the calls get serialized on internal locks.



SYNERGY

CONVERSION

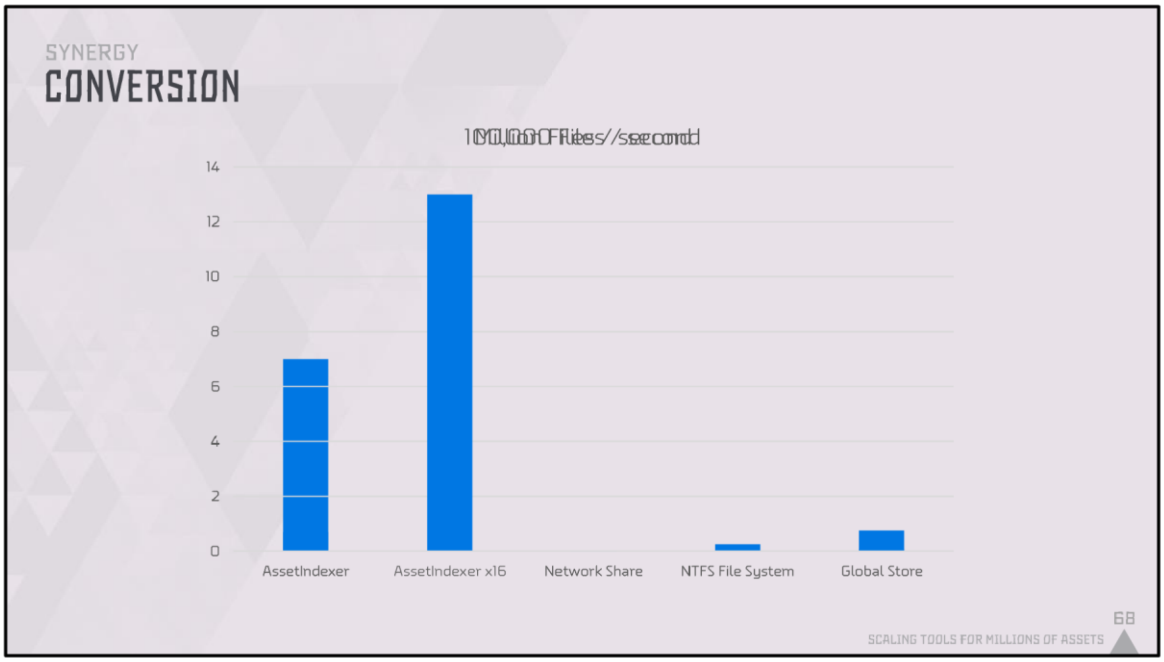
```
void* address = MapViewOfFileEx(file_mapping, FILE_MAP_READ, 0, 0, size, NULL);
AcquireSRWLockShared(shared_mutex);
AssetIndexerDatabase* database = (AssetIndexerDatabase*)address;
DBFileRow file_row = database→GetFileRow("/SimpleCoderLevel/Probes.CoreText");
HashValue file_hash = file_row→GetHash();
```

1.2 µs

67

SCALING TOOLS FOR MILLIONS OF ASSETS

For our shared memory, *CLICK* we only need to do one kernel call to map the Asset Indexer database into the converter's process space, *CLICK* and one more call to get a reader lock on it. *CLICK* With 2 kernel calls we can now access the indexed information about all 4 million files like any other data in process memory. *CLICK* After mapping it, looking up the hash for a file only takes 1.2 microseconds.



SYNERGY

CONVERSION

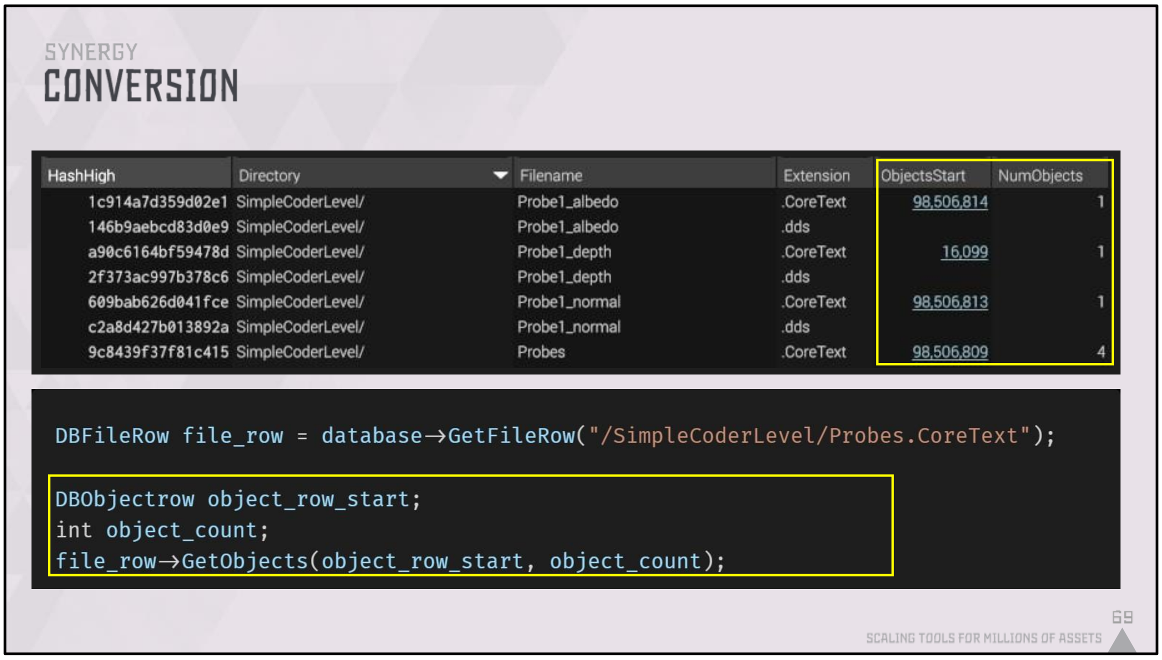
68

SCALING TOOLS FOR MILLIONS OF ASSETS

So using shared memory we can look up 700,000 hashes per second. Compare that to our previous numbers

- The network share did 700 files/s, which isn't even visible on this chart.
- NTFS file system was 25,000 files/s
- The GlobalStore did 75,000 files/s

But it gets better! That's just on a single thread. *CLICK* While holding our read-only lock, the Asset Indexer scales perfectly to parallel operations. Spreading this out over 16 cores I get 13,000,000 files/s. That means we can look up the hash of every single file in our game in 1/3 of a second. Compare that to the 2.5 minutes you would spend just opening the files.


SYNERGY

CONVERSION

HashHigh	Directory	Filename	Extension	ObjectsStart	NumObjects
1c914a7d359d02e1	SimpleCoderLevel/	Probe1_albedo	.CoreText	98,506,814	1
146b9aebcd83d0e9	SimpleCoderLevel/	Probe1_albedo	.dds		
a90c6164bf59478d	SimpleCoderLevel/	Probe1_depth	.CoreText	16,099	1
2f373ac997b378c6	SimpleCoderLevel/	Probe1_depth	.dds		
609bab626d041fce	SimpleCoderLevel/	Probe1_normal	.CoreText	98,506,813	1
c2a8d427b013892a	SimpleCoderLevel/	Probe1_normal	.dds		
9c8439f37f81c415	SimpleCoderLevel/	Probes	.CoreText	98,506,809	4

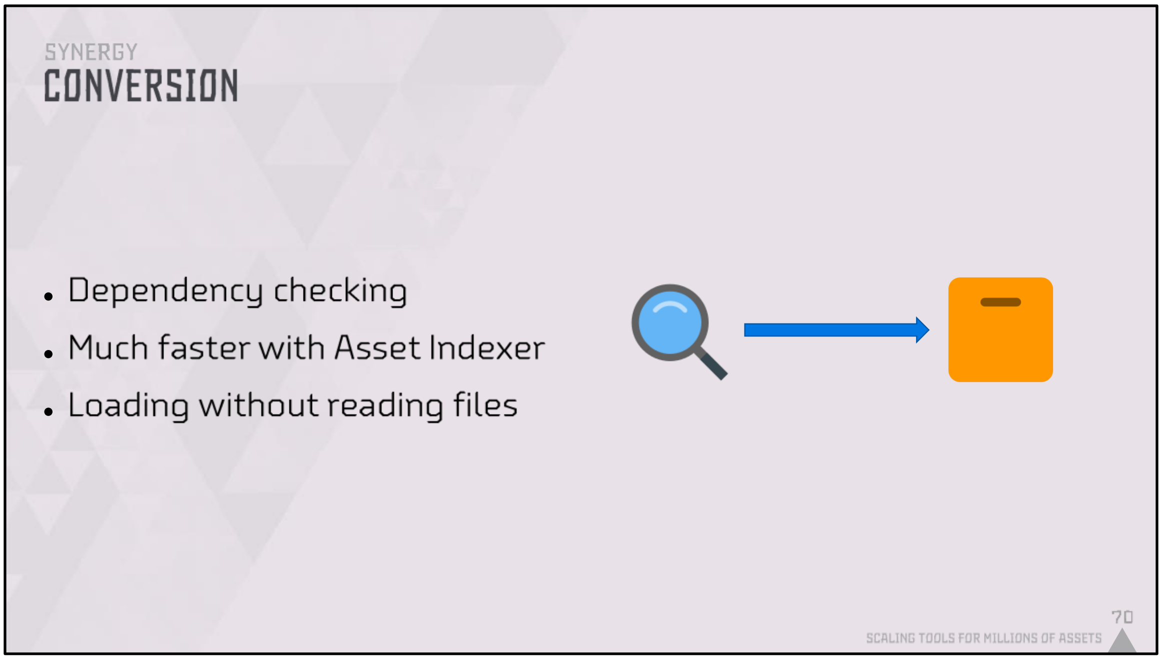
```
DBFileRow file_row = database→GetFileRow("/SimpleCoderLevel/Probes.CoreText");

DBObjectrow object_row_start;
int object_count;
file_row→GetObjects(object_row_start, object_count);
```

69

SCALING TOOLS FOR MILLIONS OF ASSETS

And when I say memory-mapped database, that’s just not a matter of memorymapping a SQLite file and performing queries on it. The database is an actual datastructure that has been designed for our content.

Files and objects are stored in separate tables for efficient storage, but navigating the data structure is still highly performant. *CLICK* A file’s entry has a direct index into the object table, pointing to contiguous rows that containing the objects for that file. No need to run complicated queries or iterate over all rows to find the related data. And the same thing goes for links.



SYNERGY

CONVERSION

- Dependency checking
- Much faster with Asset Indexer
- Loading without reading files

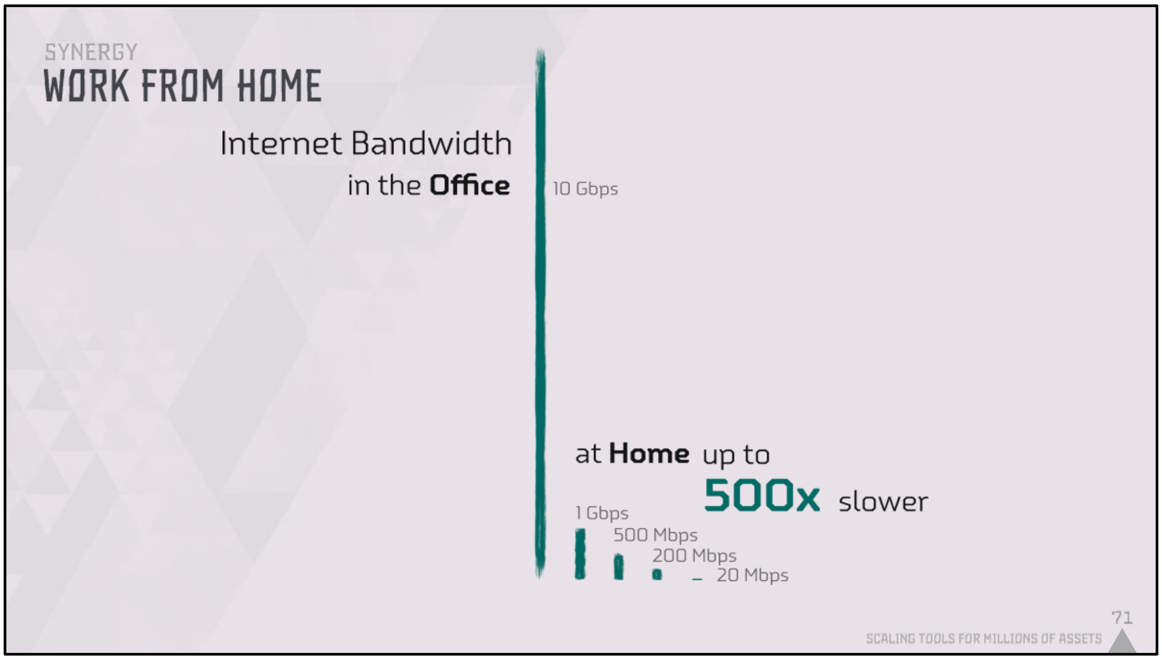
SCALING TOOLS FOR MILLIONS OF ASSETS 70

That means we can build up the link graph, compare hashes, and do some other checks and in 40 seconds we know for all million coretext files in the game if they have been converted.

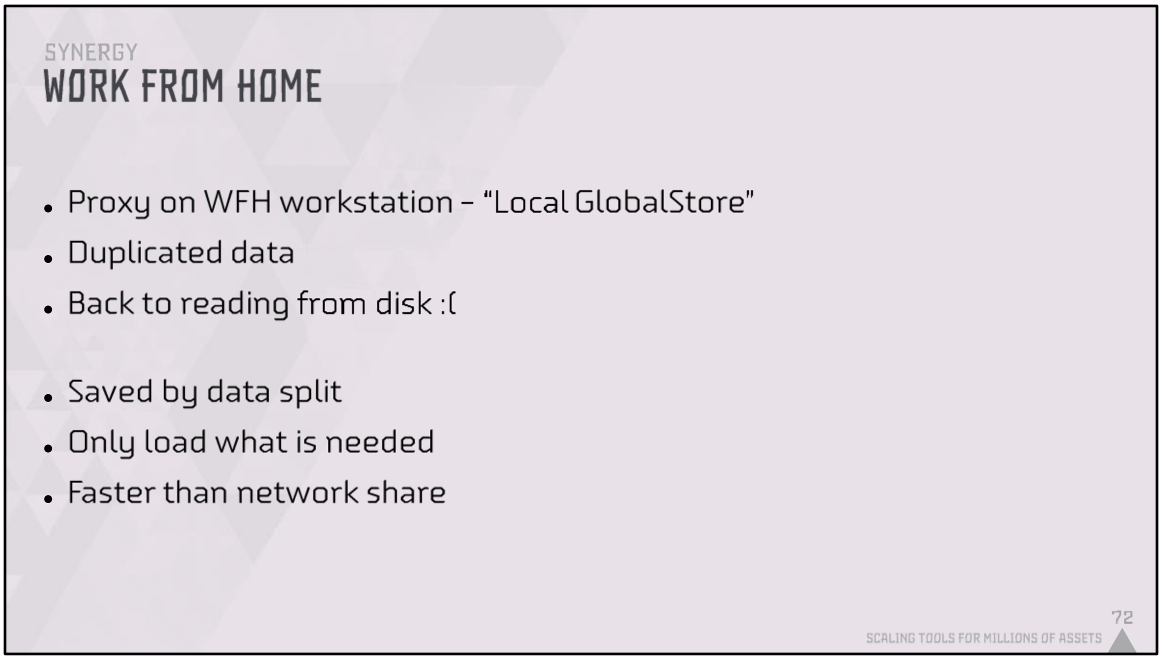
So by switching dependency

checking to use the Asset Indexer, the whole process got significantly faster. But maybe more importantly, we can load into the game without reading a single CoreText file from disk. We do the dependency checking against the database and stream the converted content from the Global Store. And this will become even more relevant later on.

Because almost exactly 3 years ago, March 2020, everyone suddenly had a very strong desire to start working from home.



All of these designs were made with our office network in mind. Which is 10Gbps and less 1ms latency to every workstation. We had decided we were going to send complete workstations home, which meant some people were working through a 20Mbps connection in the worst cases. That’s 500x slower than our office. Downloading a 95GB package on such a connection would have taken an entire workday. But obviously streaming over the internet as I’ve been describing also wasn’t going to work.


SYNERGY

WORK FROM HOME

- Proxy on WFH workstation – “Local GlobalStore”
- Duplicated data
- Back to reading from disk :(

- Saved by data split
- Only load what is needed
- Faster than network share

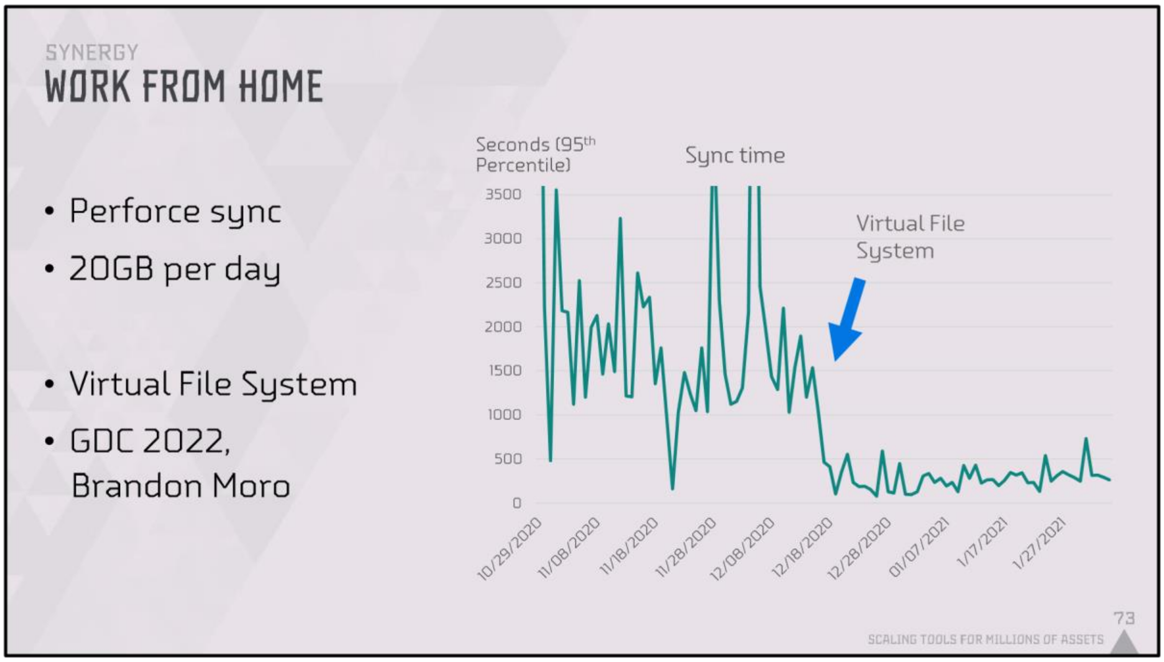
72

SCALING TOOLS FOR MILLIONS OF ASSETS

So we set up GlobalStore proxies on every WFH pc. In a fit of irony this meant we now had Local
GlobalStores. This wasn’t ideal as we’re back to every workstation containing duplicated data that needs to be read from disk.

But all of the work we did was still incredibly valuable. By having the data split into smaller chunks and only loading what is needed, people didn’t have to download a full 95GB every day. Instead, after they get the latest revision, they only download the delta of converted content since the

last sync and only in the area where they are working. And our own GS proxies achieve much higher throughput than the network file share for these small requests.



SYNERGY

WORK FROM HOME

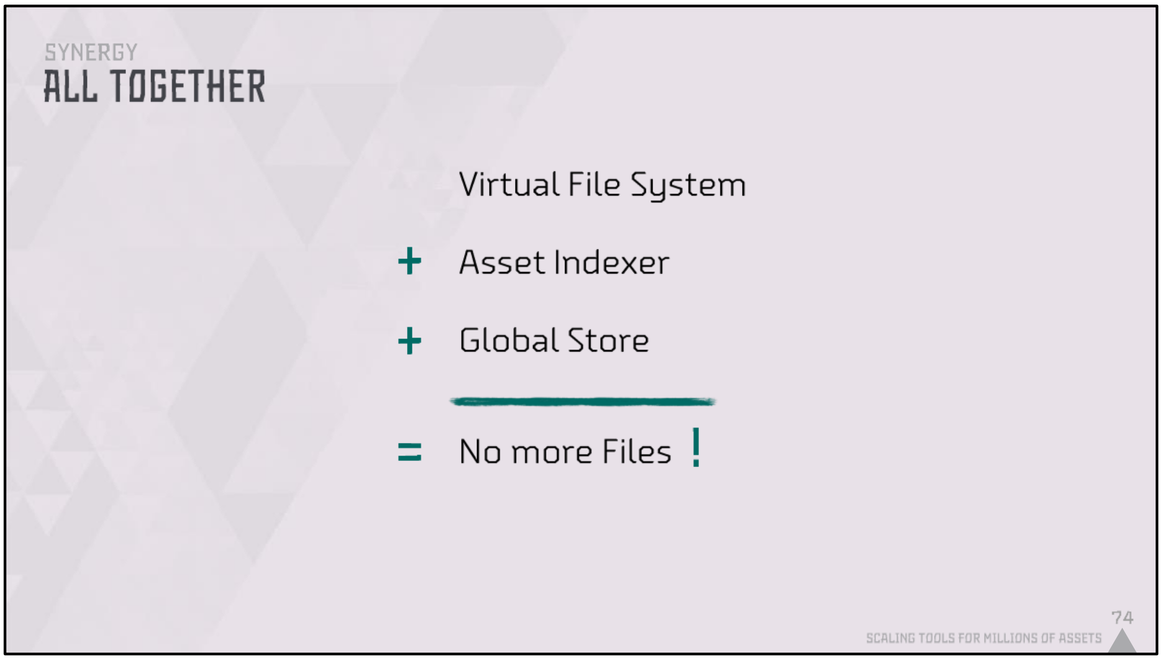
- Perforce sync
- 20GB per day
- Virtual File System
- GDC 2022, Brandon Moro

73

SCALING TOOLS FOR MILLIONS OF ASSETS

Our biggest bottleneck at this point was syncing the game assets from Perforce. Users had to download 10-40GB everyday, but similar to the converted content, most of that wasn't actually used by that person. Luckily we were able to use a most excellent Virtual File System by some industry friends. *CLICK* In summary, the file appears to be there to the user or any program, but the content only gets downloaded when the file is actually read. For more details, Brandon Moro had a great talk on the subject at GDC last year. This was rolled out to everyone and made syncing for people at home go

from hours to minutes and even improved it for those in the office.



SYNERGY

ALL TOGETHER

Virtual File System

+ Asset Indexer

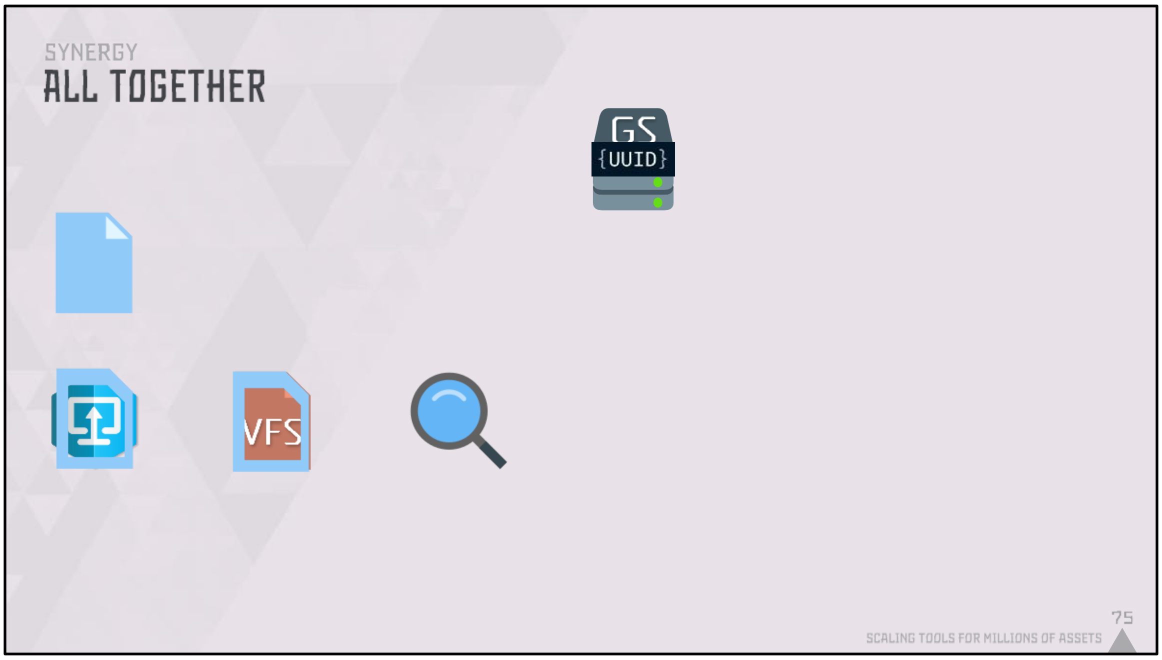
+ Global Store

= No more Files !

SCALING TOOLS FOR MILLIONS OF ASSETS 74

The Virtual File System was the final piece of our triforce. By combining it with the AssetIndexer and the Global Store, we can start working on the game without having a single piece of content on disk.

Only the content that is needed is downloaded on demand. To put it all together, let's go through the lifetime of a file.



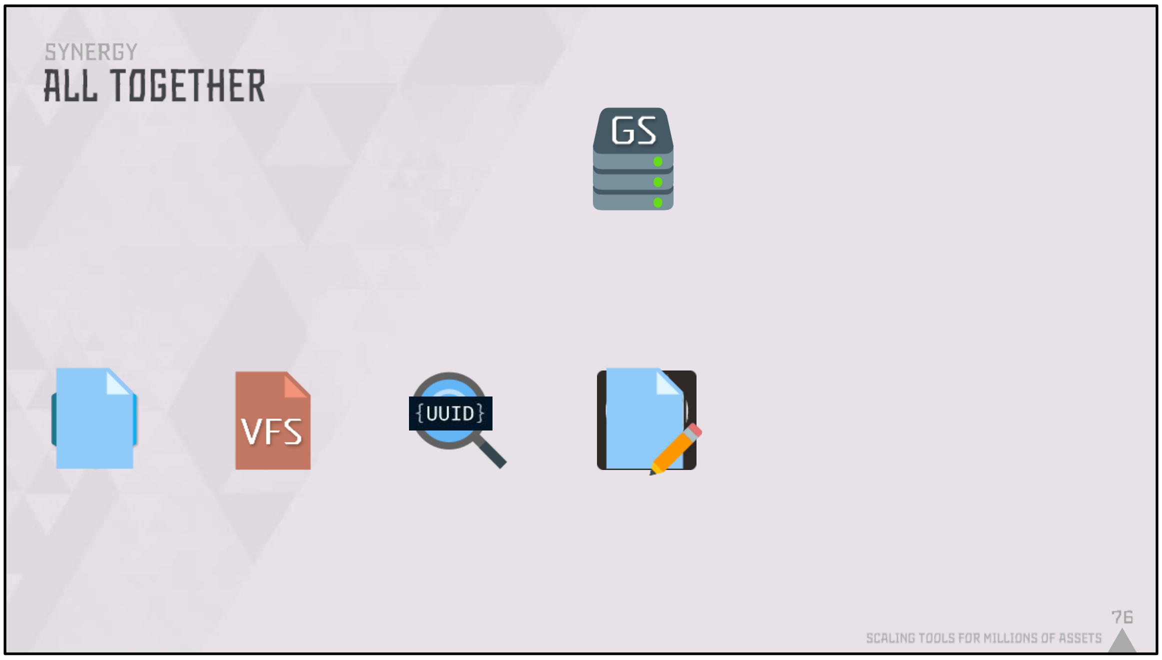
SYNERGY

ALL TOGETHER

It starts with someone submitting a file to Perforce.

Someone else then syncs the file using the Virtual File system. Perforce provides enough metadata for it to pretend the file is there.

The Asset Indexer sees this new file, and based on the metadata pulls an extract with objects, UUIDs, etc from the Global Store.



SYNERGY

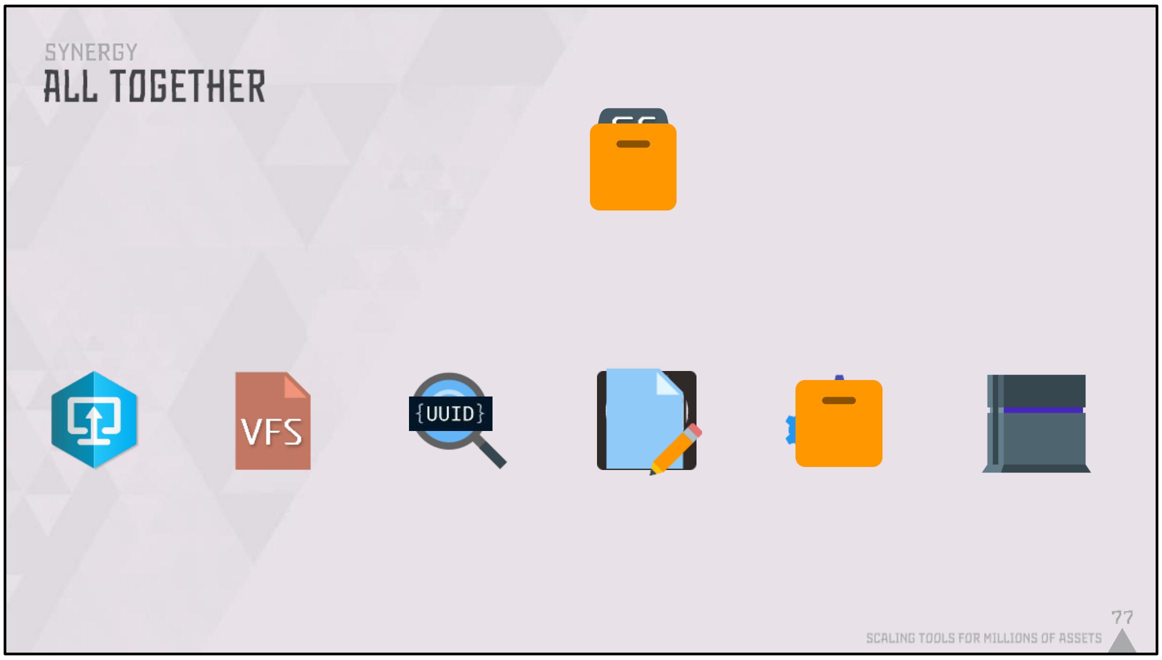
ALL TOGETHER

The user then opens the Asset Browser in the Editor, which asks the Asset Indexer for the list of objects, which now includes the new file.

If the user opens the file, the full contents are fetched on-demand

from Perforce by the Virtual File System.

The user can then make some edits, and on save the Asset Indexer re-scans the file with the new changes.


SYNERGY

ALL TOGETHER

To tests the changes the user starts the game, which also starts a conversion process. This asks the Asset Indexer for the state of the disk.

Noticing there are local changes that haven't been converted yet,

it converts those and uploads them to the Global Store.

Finally the game then streams those new assets.

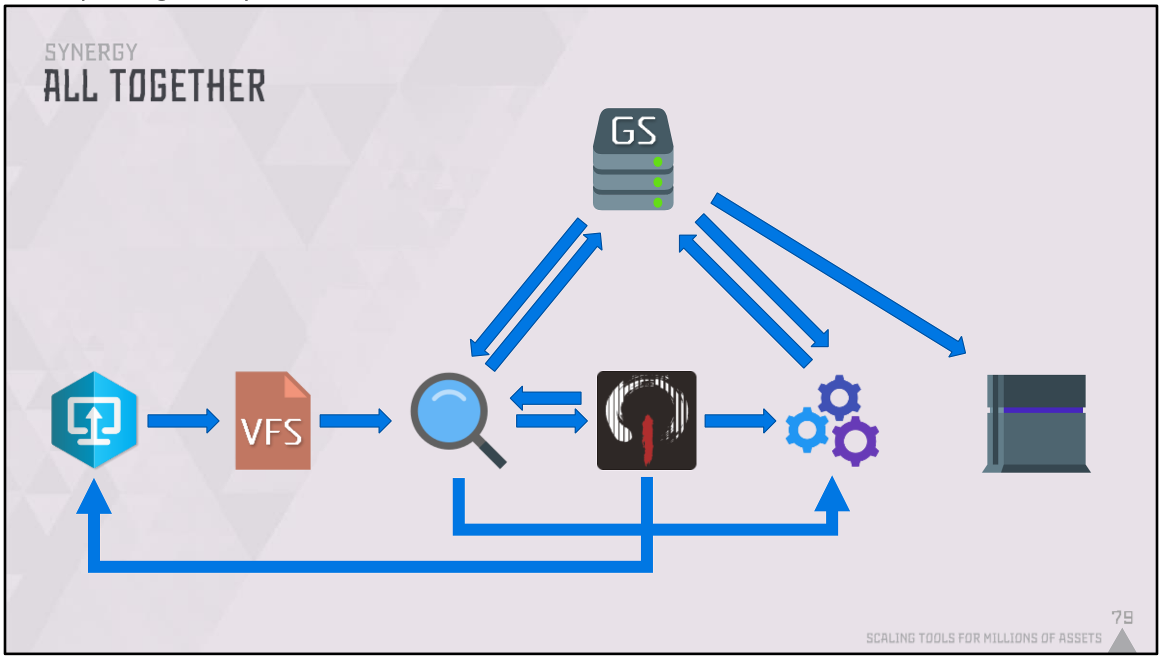
SYNERGY

ALL TOGETHER

78

SCALING TOOLS FOR MILLIONS OF ASSETS

And being happy with their changes, they submit their file back to source control, completing the cycle.


SYNERGY

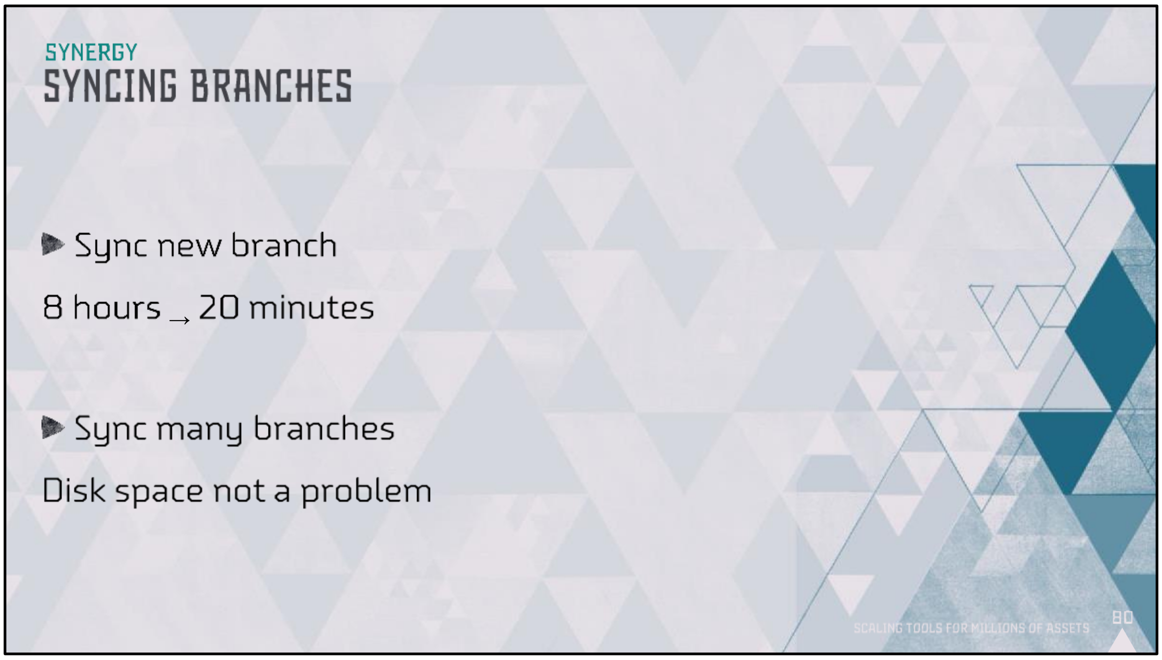
ALL TOGETHER

79

SCALING TOOLS FOR MILLIONS OF ASSETS

Even this spaghetti of a chart is still somewhat simplified, but all of these systems working together means we locally only ever store the data that we absolutely need.

DRINK



SYNERGY

SYNCING BRANCHES

- Sync new branch

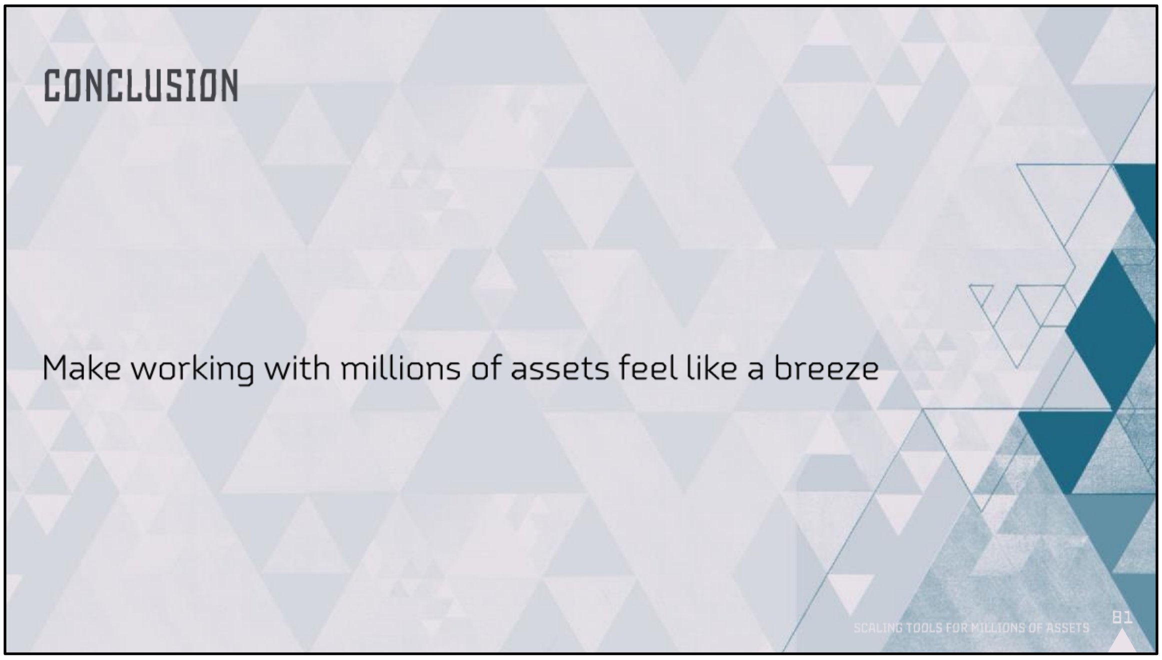
8 hours → 20 minutes

- Sync many branches

Disk space not a problem

SCALING TOOLS FOR MILLIONS OF ASSETS 80

Thanks to this, syncing and starting the game on a new branch no longer takes more than 8 hours, but only 20 minutes.
And because you no longer need all game assets and converted content, you can easily have multiple branches synced without running out of storage space, so switching between projects is just a few seconds.



CONCLUSION

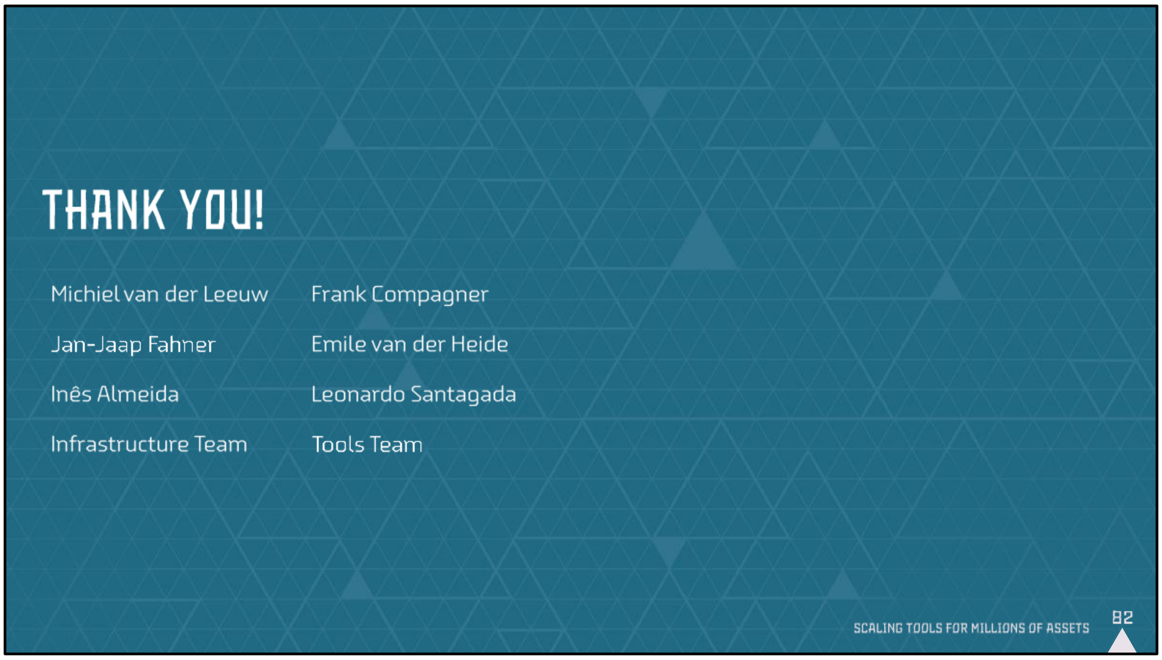
Make working with millions of assets feel like a breeze

SCALING TOOLS FOR MILLIONS OF ASSETS 81

So in conclusion, by having a local Asset Database that can rapidly provide information about all the game's content,
a loading system that streams all data over the network,
And by combining these 2 systems in smart ways.

We can greatly reduce the time spent waiting around and make working with millions of assets feels like a breeze.

Hopefully the coffee machine doesn't start feeling too lonely.



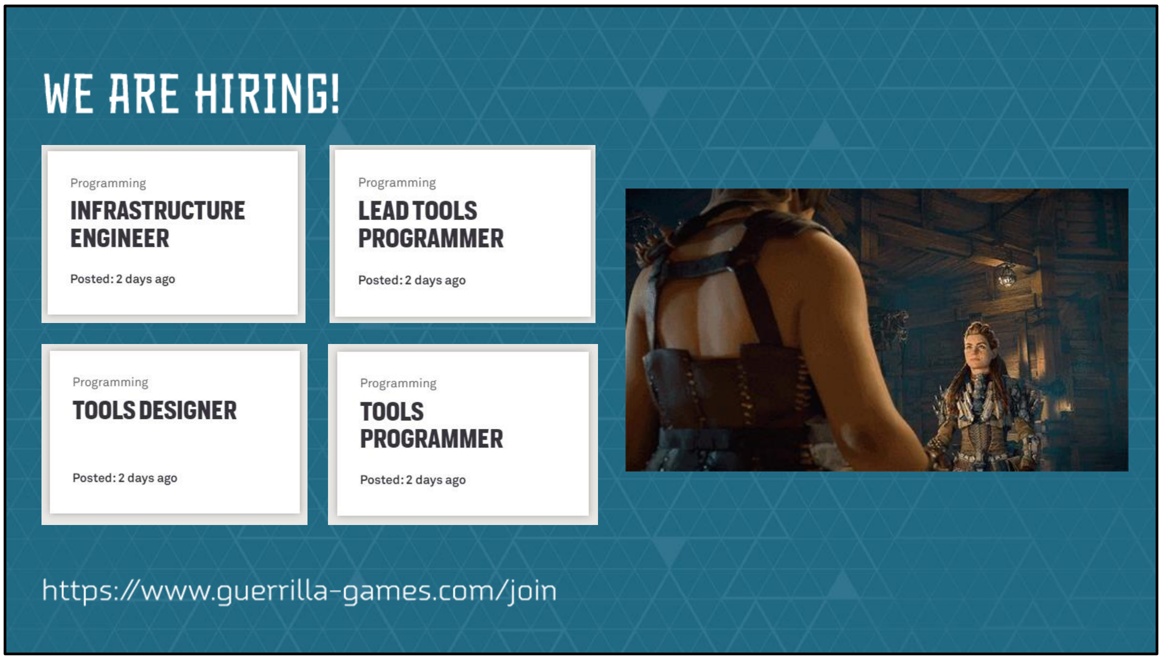
THANK YOU!

Michiel van der Leeuw	Frank Compagner
Jan-Jaap Fahner	Emile van der Heide
Inês Almeida	Leonardo Santagada
Infrastructure Team	Tools Team

I want to give a big shout out to all my amazing colleagues at Guerrilla. In particular to these people for their support of this presentation, all of their smart ideas, and actually making a lot of the things that I've talked about.

And thank you all for your attention!

One more note:



WE ARE HIRING!

Programming
INFRASTRUCTURE ENGINEER
Posted: 2 days ago

Programming
LEAD TOOLS PROGRAMMER
Posted: 2 days ago

Programming
TOOLS DESIGNER
Posted: 2 days ago

Programming
TOOLS PROGRAMMER
Posted: 2 days ago

https://www.guerrilla-games.com/join

Here's the obligatory "We are hiring" slide! We still have a lot of ideas, so if you're interested in joining us to push this tech even further, come talk to me or apply on our website!

QUESTIONS?

david.marcelis@guerrilla-games.com

david-marcelis

A reminder to please fill out the questionnaire.

With that, are there any questions?