



Streaming Sunset Overdrive's Open World

Elan Ruskin
Insomniac Games

twitter: [@despair](https://twitter.com/despair)
slides: bit.ly/1EMgHUv

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015



Let's talk about Sunset Overdrive, last year's title out of Insomniac Games.



Sunset is about fast traversal and varied missions across seamless regions, (mostly) without load screens.

View this video at: <http://youtu.be/6cG3LOerGlc>

Today's storytime

- Insomniac's experience with turning a linear engine into an open-world game.
 - Not everything we built was clever.
 - Not everything worked the first time.
 - This is how it worked for us.

This talk is a narrative of our own experience making this game. There have been a lot of GDC talks about building streaming engines and open-world games; we aren't the first and we won't be the last. But everyone seems to have done it differently, so I'm going to tell you how it went for us.

Lots of the talks that have been done here before discussed engines that were built for streaming, built from the ground up to support open worlds. That wasn't our engine. So this is really a story about *our* experience of taking an engine that had been built for linear games and adapting it, one step at a time, for an open world environment. We hit a lot of hitches and did a lot of learning along the way.

Overview

- Our engine before Sunset Overdrive
- How we adapted it
- Problems we discovered by stepping on them

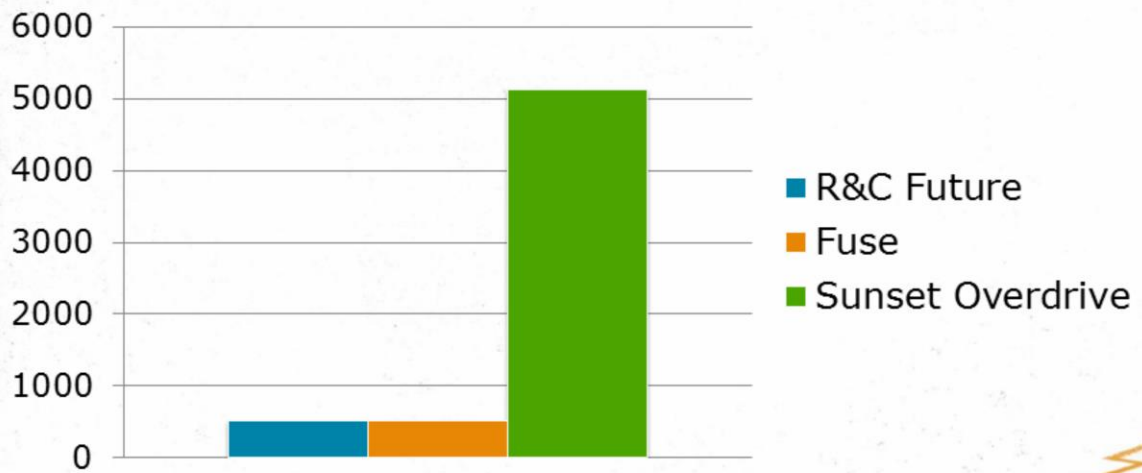
Our story in three acts: where we started, how things went, and the things that surprised us.

Our Engine's Evolution



SO's tech is an evolution from what we had in *Ratchet*, *Resistance*, and primarily *Fuse*.

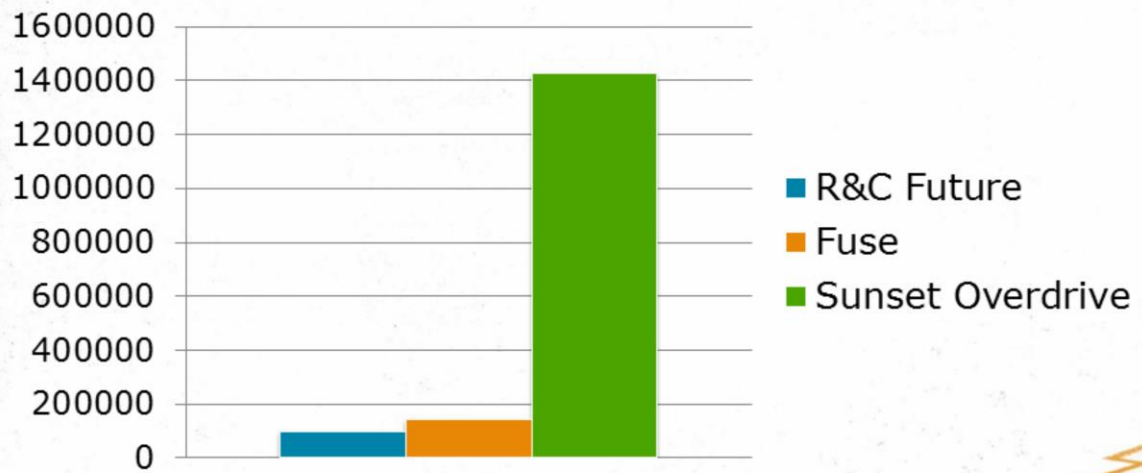
Available RAM (mb)



Sunset overdrive is way bigger in every respect than our previous games.

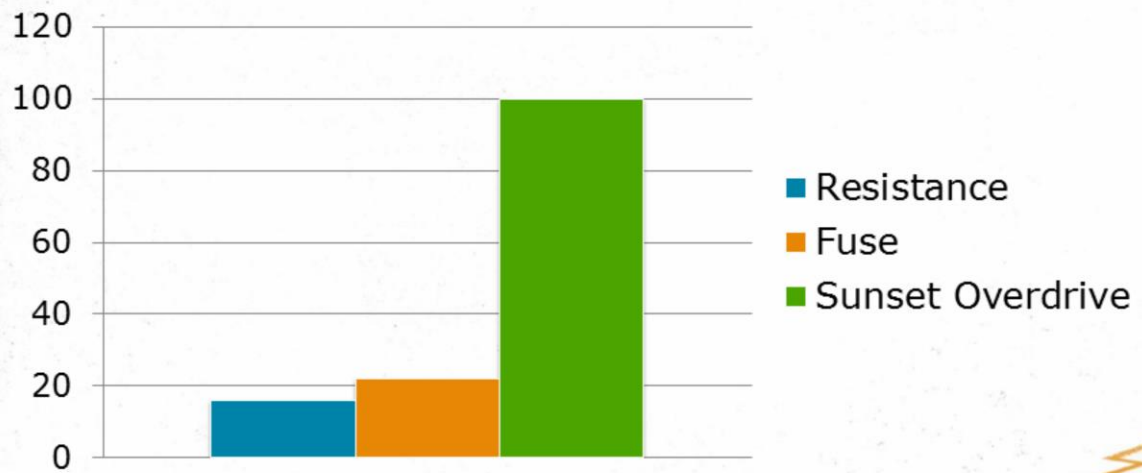
The platform it runs on, the Xbox One, has ten times the memory (and more CPU and more everything) than the Playstation 3 and the Xbox 360.

assets on disk



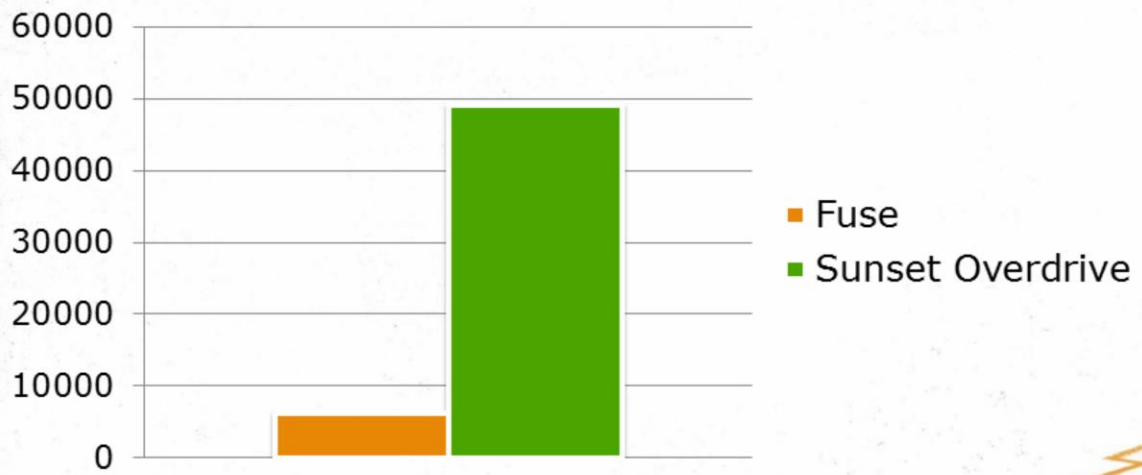
We had ten times the *stuff* in our world than either of our previous biggest games.

Simultaneous NPCs



Five times the active enemies and other AIs.

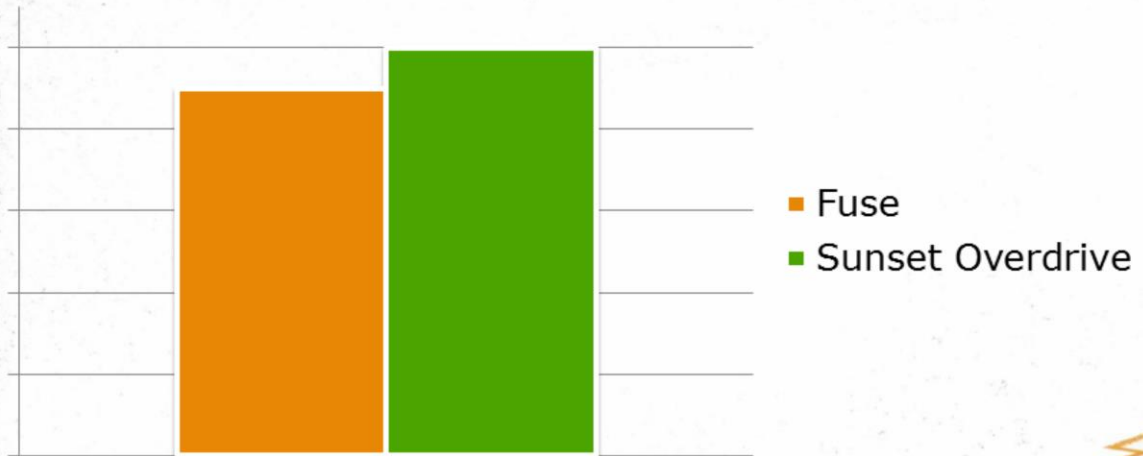
Actors



The engine was simulating ten times as many simultaneous things.

The one thing that wasn't bigger in Sunset...

of developers



What we did *not* have 10x as much of was people. Sunset Overdrive's team was only slightly larger in size than Fuse's.

Ratchet & Clank

- Each planet is a “level”



Novalis.level



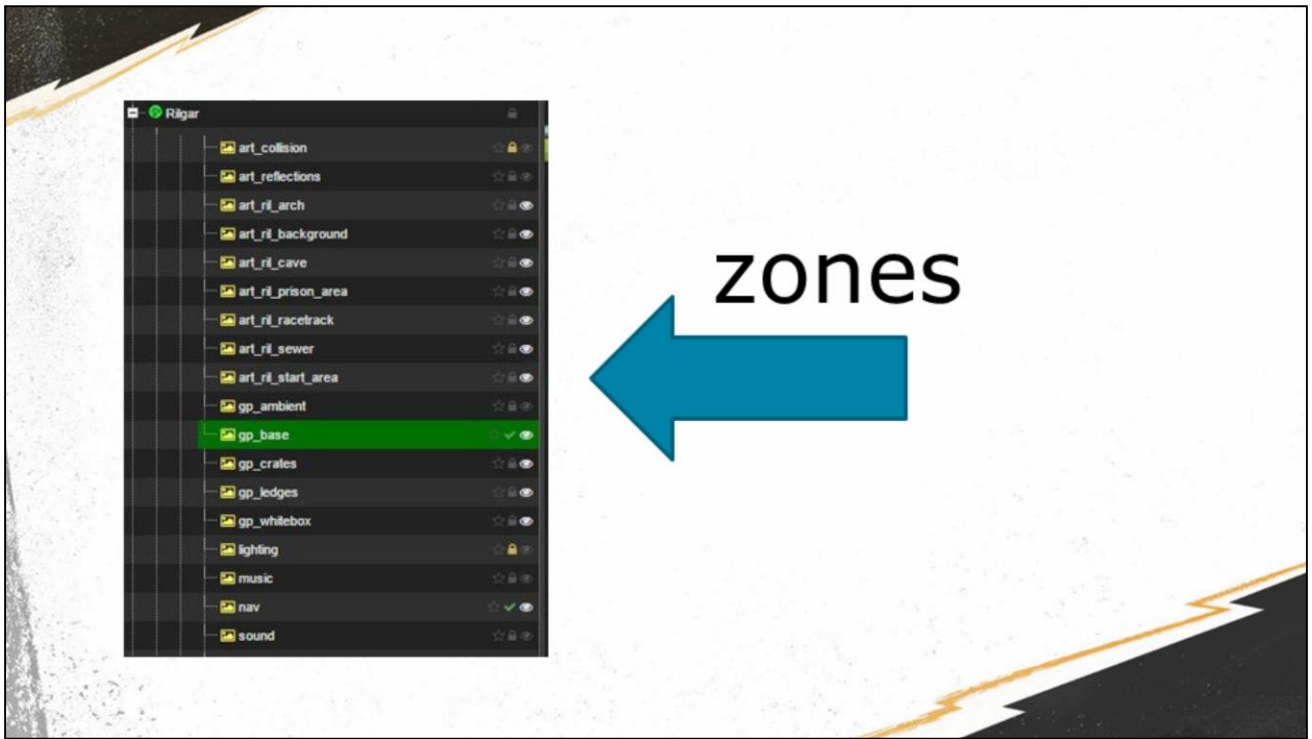
Rilgar.level



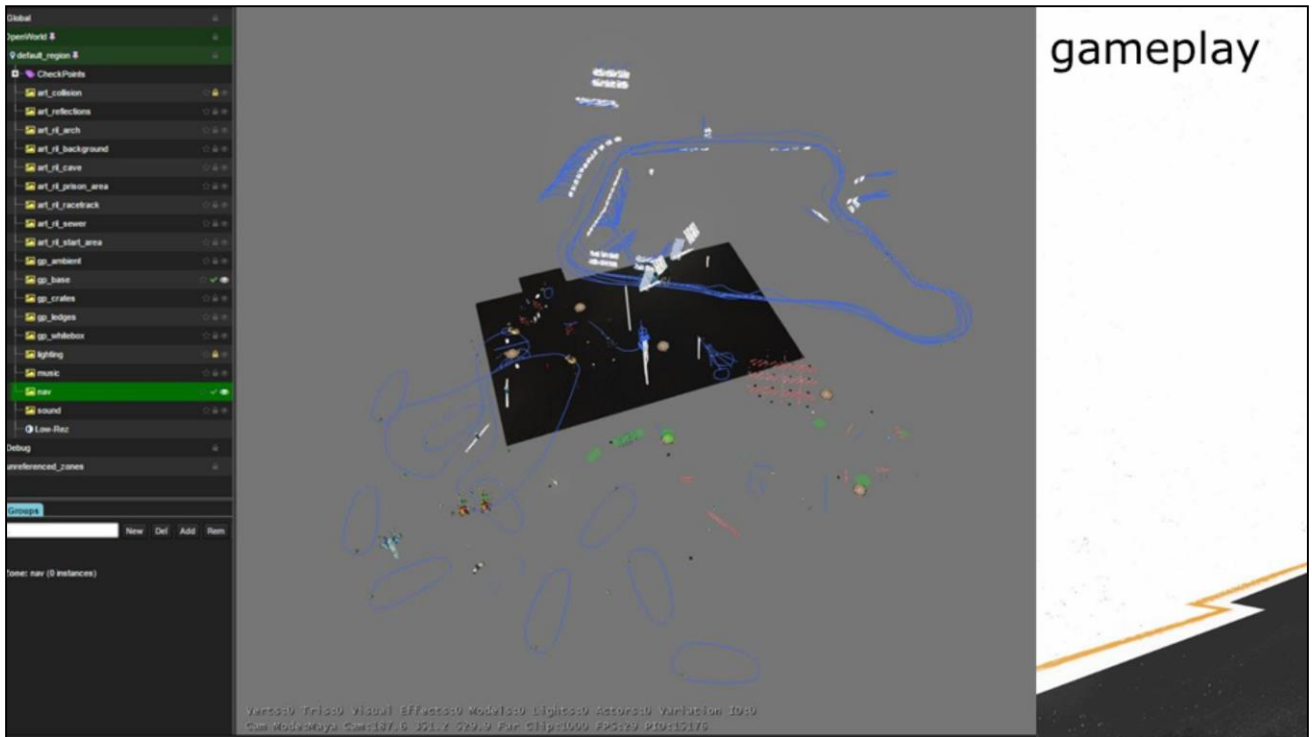
Gaspar.level

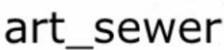
In *Ratchet* each planet was a “level”. All its assets were loaded at once.

When you went from one level to another, we evicted all of its data and loaded the next. We covered the transition with a little animation of Ratchet flying his spaceship from one planet to the next.

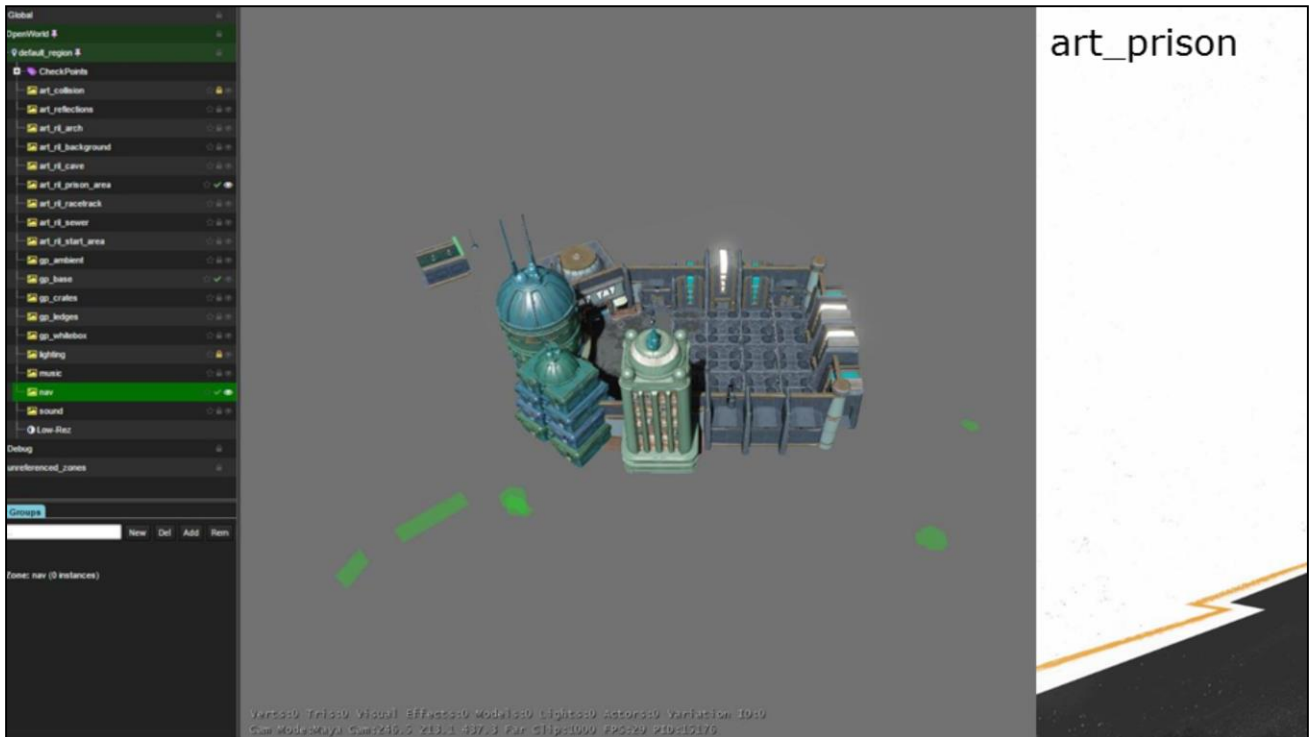


We split levels into terrain/lighting/set-dressing/script zones to allow multiple people to work concurrently on the same planet. If we'd had just the one level file, then only one person could have worked on it at a time (or else we would have needed some mergeable data type and a lot of complexity). So we split the levels up into layers and people worked on the layers simultaneously.

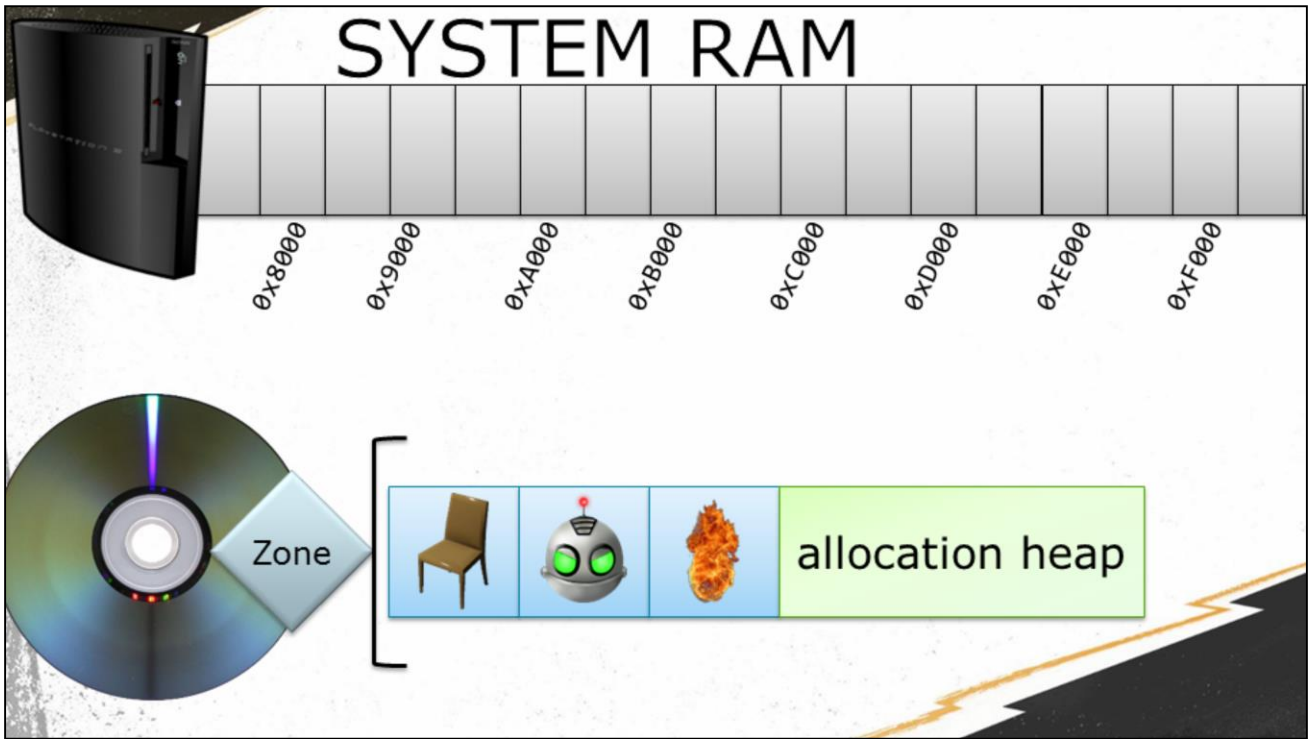




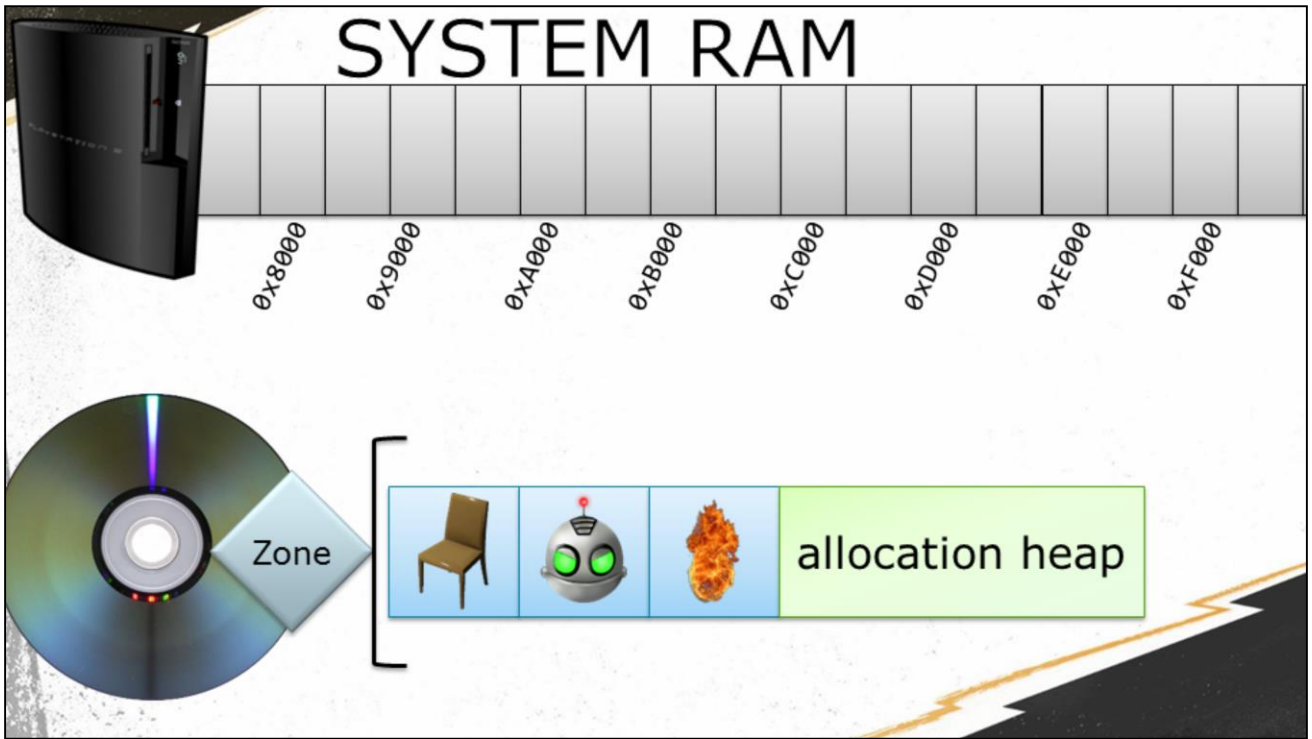
17



Another on this,



In these games, the zones contained *all* the data necessary for running them, including the actor heaps – we’d map out the slack space in memory that the dynamic allocator would use. This was possible (and nice) because we could know, deterministically, all the stuff that a level needed in order to work. We could determine that by looking at the assets on disk and their references to each other. We could also budget ahead of time how many actors we could have because we knew how much memory there was for them.



You loaded a zone basically by memcopying it. When you unload a zone, all the stuff in that zone – *including* the actors – gets immediately unloaded also. It's a very straightforward system, but it relies on that complete build-time determinisim.

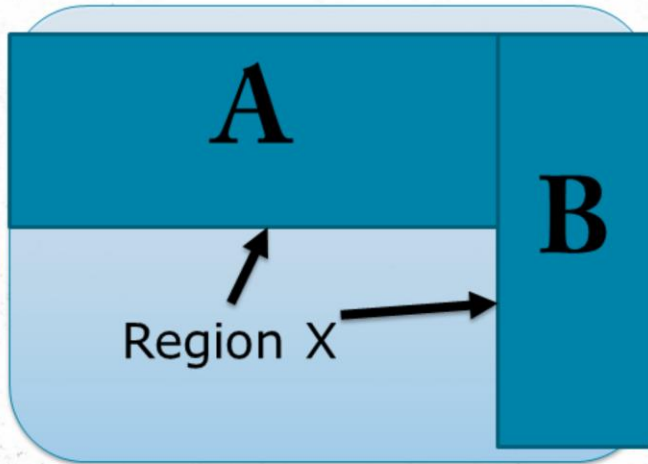


As we transitioned towards making linear shooters, we wanted the player to have a continuous experience without needing load screens. We used airlocks for this. Here's an example from Fuse. It's a traditional technique: you divide the world into regions, and have adjacent regions share some overlapping area, like an elevator or a corridor. When the player steps into the corridor, you lock the door behind her and start unloading the region she just left, then load the next region into memory.

You either make the corridor long enough so that the next zone is loaded by the time you've left the last one, or you come up with some interaction to detain the player long enough to cover the changeover.

View this video at: <http://youtu.be/wxIIX-3bg5A?t=48m23s>

Streaming before Sunset: airlocks

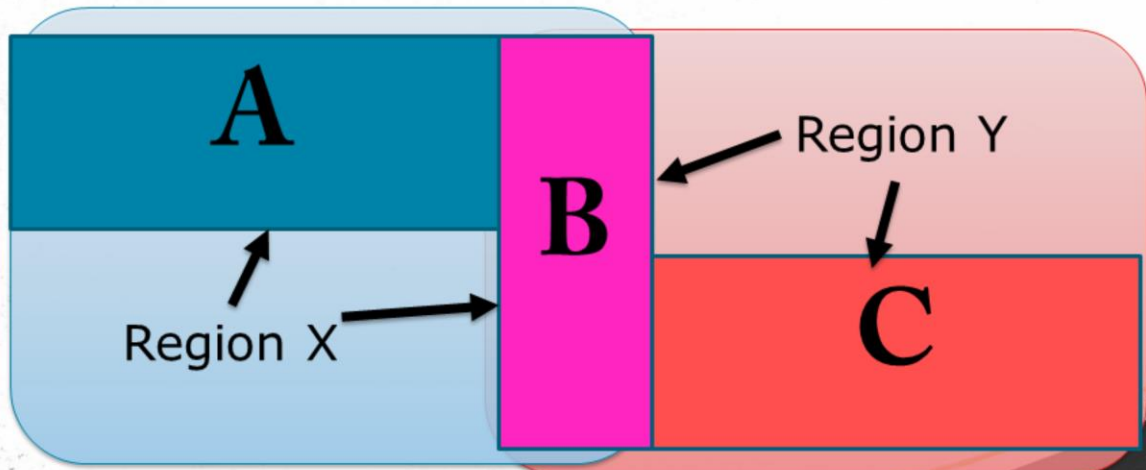


Here's Fuse's implementation.

A "region" is a list of zones. In the figure, region X contains references to zones A and B. Region Y contains references to zones B and C.

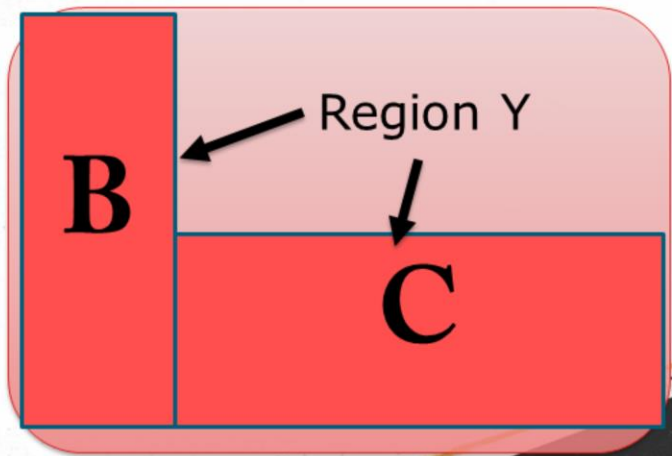
So both of these regions have zone B in them; that is the shared corridor.

Streaming before Sunset: airlocks



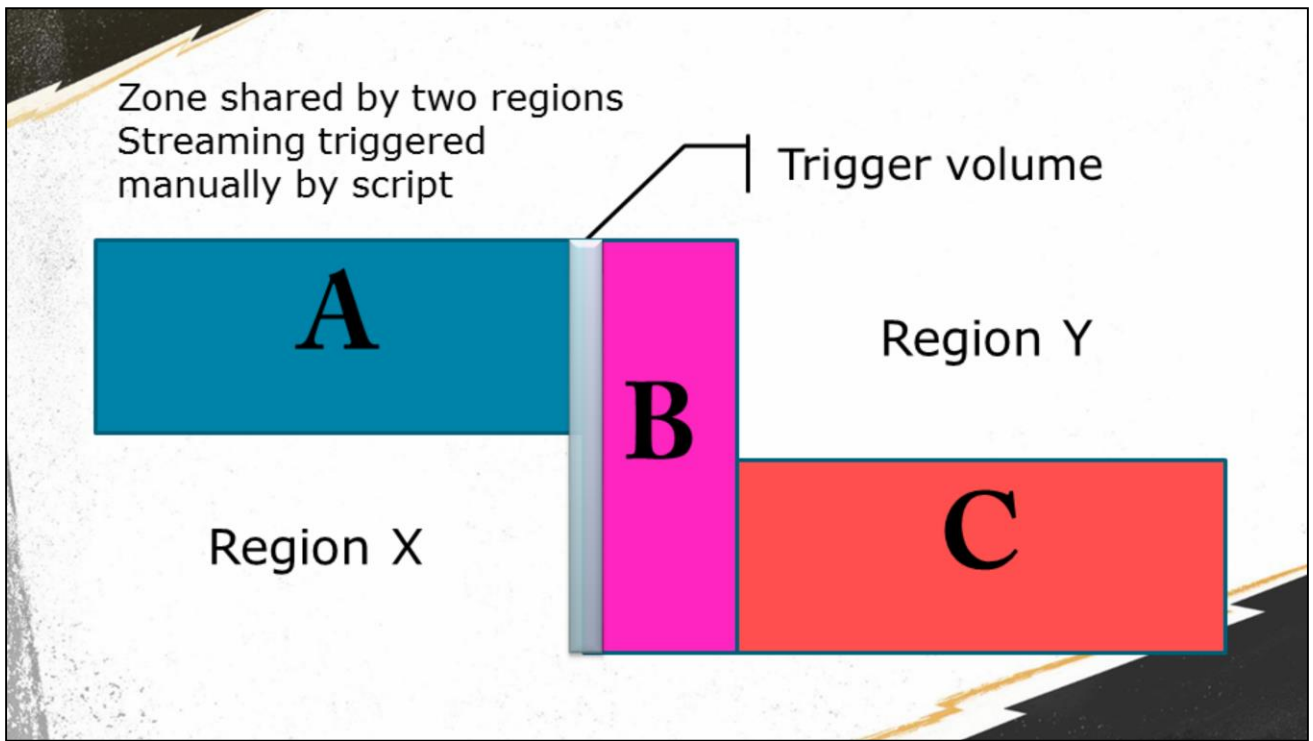
When the player walks from zone A to B, he enters a trigger volume which fires a script that loads region Y. Zone B now has two references to it, one from each Zone.

Streaming before Sunset: airlocks

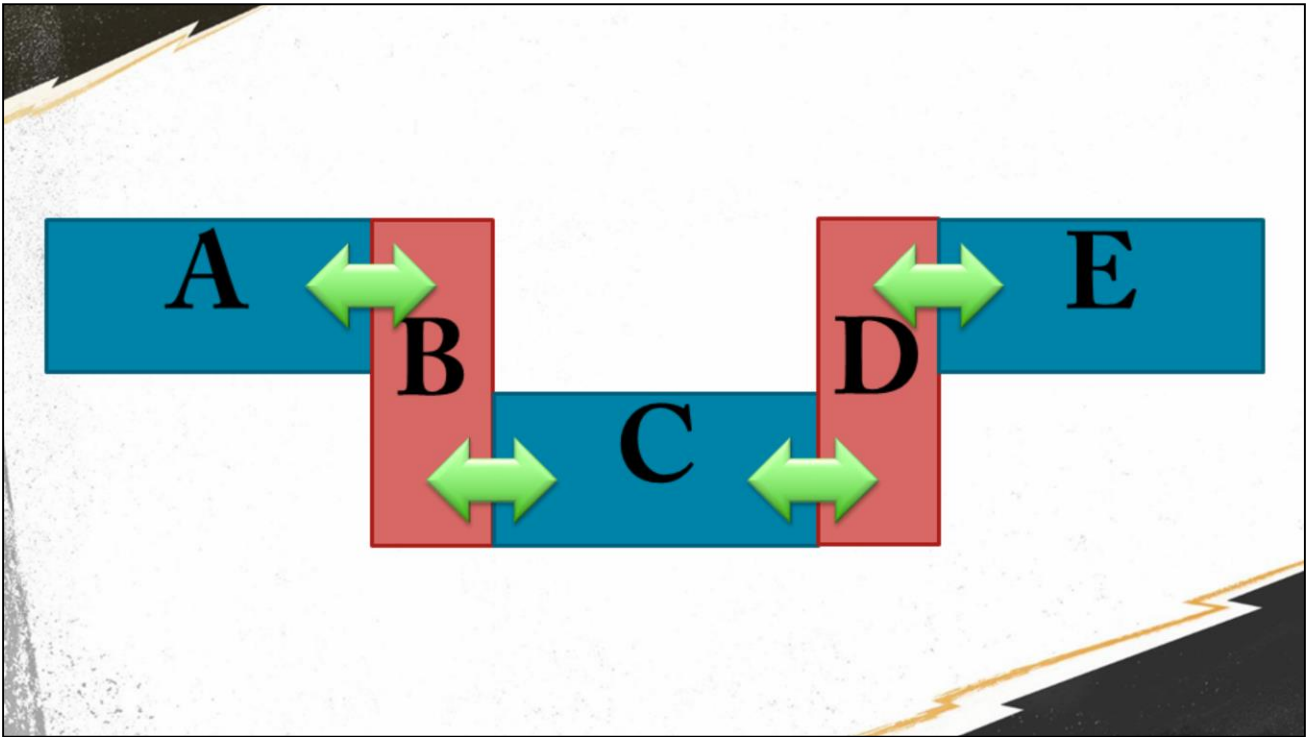


When the player moves from zone B to zone C, another script trigger tells region X to unload. Zone A unloads from memory, but B still has one reference to it (from Y) so stays loaded in case the player turns around.

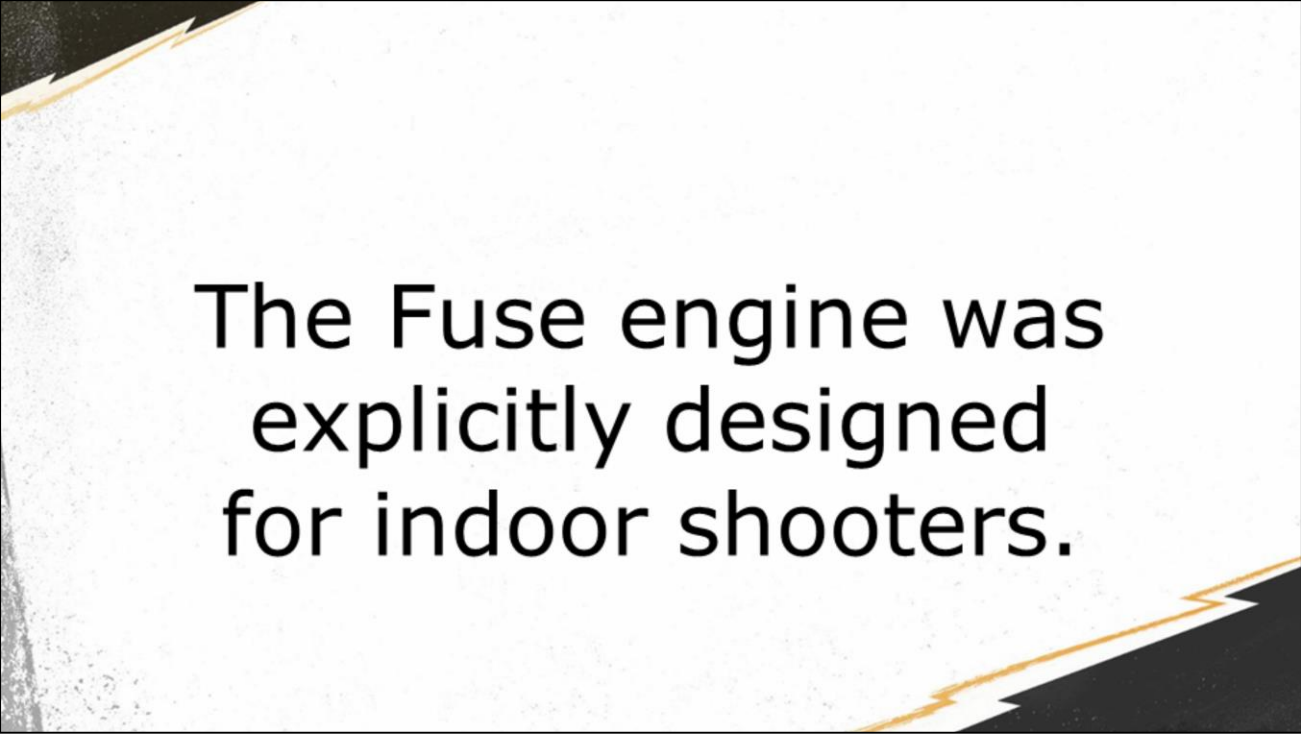
This is all very standard.



In Fuse, we did this explicitly, by having trigger volumes that ran scripts which told the engine exactly which zones to unload and load.



And that worked here because these games were pretty linear. From each region there was usually only one or two other regions you could move into (forward and sometimes backward).



The Fuse engine was explicitly designed for indoor shooters.

At the time, we were focused on linear games, so we optimized our engine to work really well at indoor environments. We knew that it wouldn't need to simulate any continuous outdoor environments, and built it because we weren't going to make any continuous open worlds.



Sunset Overdrive began
on Fuse's engine.

Until we did.



Sunset Overdrive wanted a completely open world, a living city where you could go in any direction from anywhere.

Trigger volumes obviously wouldn't cut it.

But we couldn't build an entirely new engine; our engine team was focused on simply porting the engine forward to the new console generation. We also didn't have time to rewrite major parts of the engine. So we approached the prototype one step at a time – figuring out how we could build adapters that let us test Sunset's gameplay within the way our engine already handled loading.



There's a temptation, when striving for something that's not been done before (at least, not by you), to engineer everything the best possible way. You want the smartest tech you can think of, because you want to do the best job you can. But you don't always need to shoot for the moon.

Image credit: NASA



The best technology is expensive, in time and money and effort. Those are scarce. And often it's overkill. You don't always need the most expensive technology; what matters is what you need to get the job done. Sometimes you just need to get your weather instruments into the stratosphere.

Image credit: Wikimedia Commons,
http://commons.wikimedia.org/wiki/File:High_Altitude_Hydrogen_Balloon.JPG



Keep it simple.

We climbed this mountain one step a time. We kept things simple, and at each stage built just what we needed to implement the game's design.



How to divide a city?

So the first question was, how would we cut up our city into small mouthfuls that we can hold in memory? How many pieces would we have in memory at once?

Picture from Wikimedia Commons:
http://commons.wikimedia.org/wiki/File:Mapping_L.A._City_neighborhood_boundaries.svg

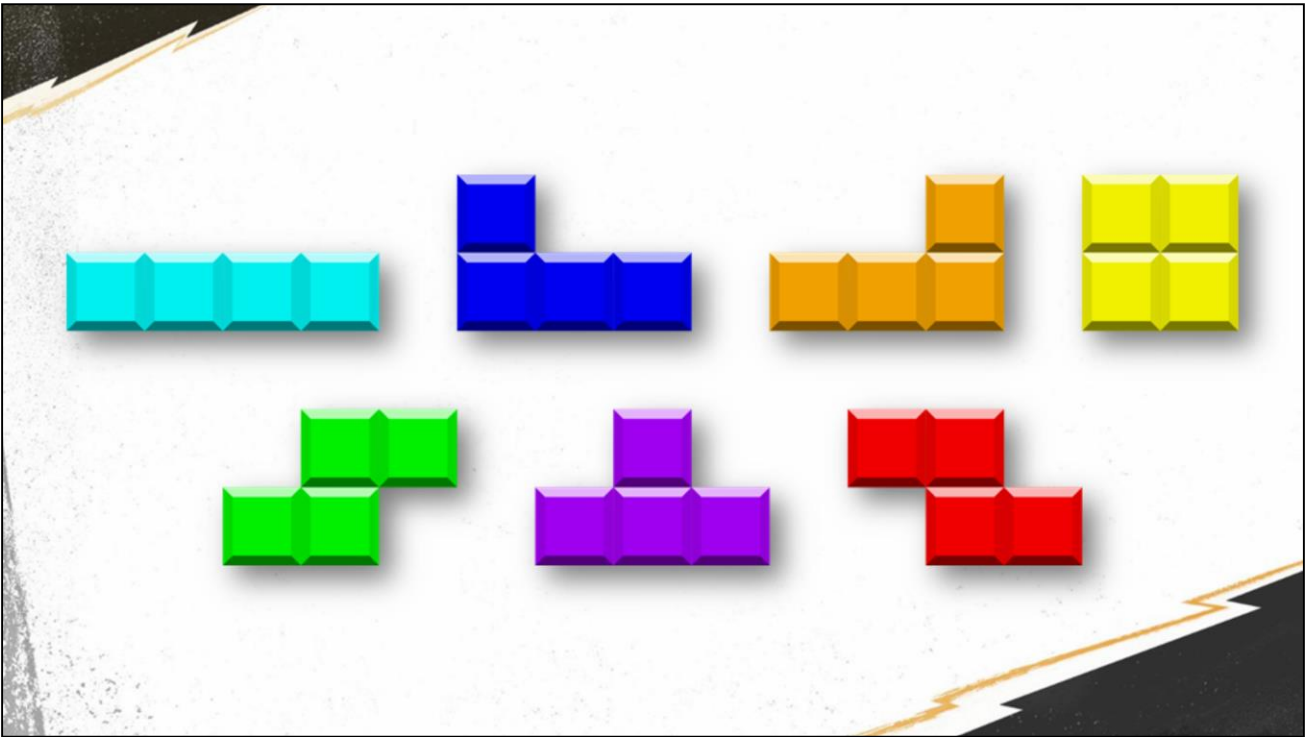


There are lots of ways to tessellate!

Mathematicians call cutting up a map into pieces “tessellation,” and there’s lots of different ways you can tessellate a map.



The pieces can be irregularly shaped.

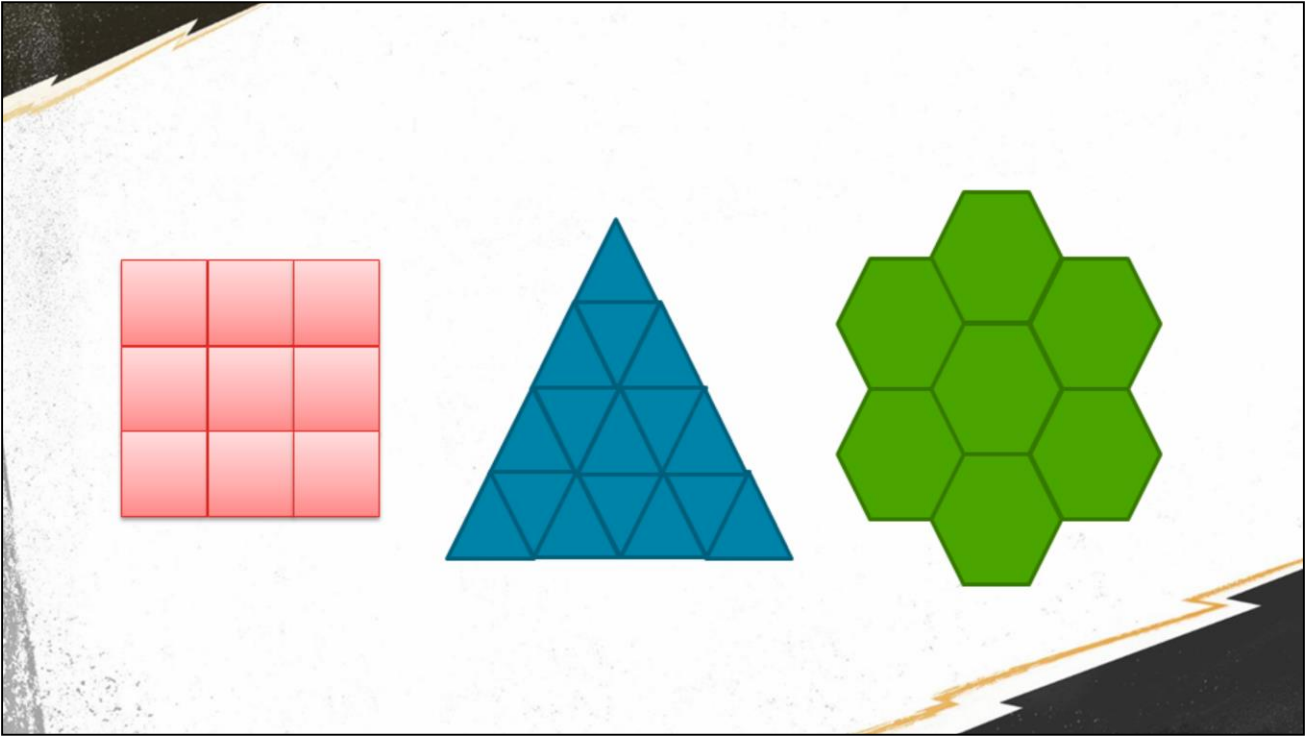


Or semi-regularly shaped.

Often we'd prefer to cut up our map into pieces that all have the same shape. But even if you restrict yourself to shapes that can be repeated to fill an entire surface...



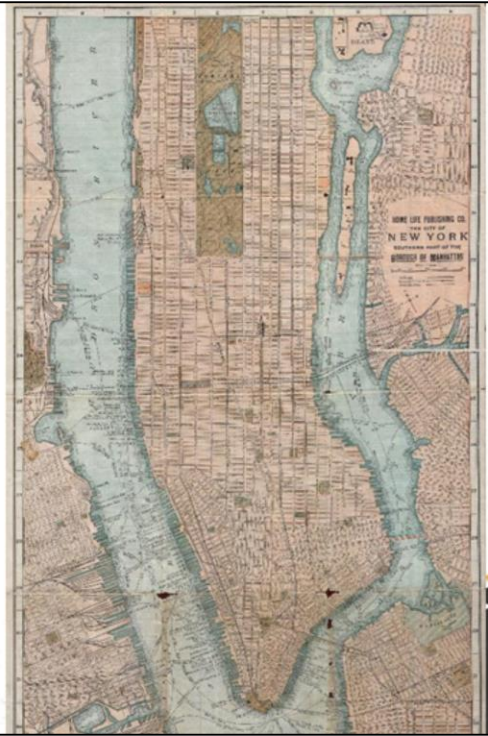
...there's still an infinite number to choose from. You're not really limited to using the regular polygons.



Still, for convenience's sake we wanted to have all our regions be of uniform size, and uniform shape. In 2d, that means triangles, squares, or hexagons.



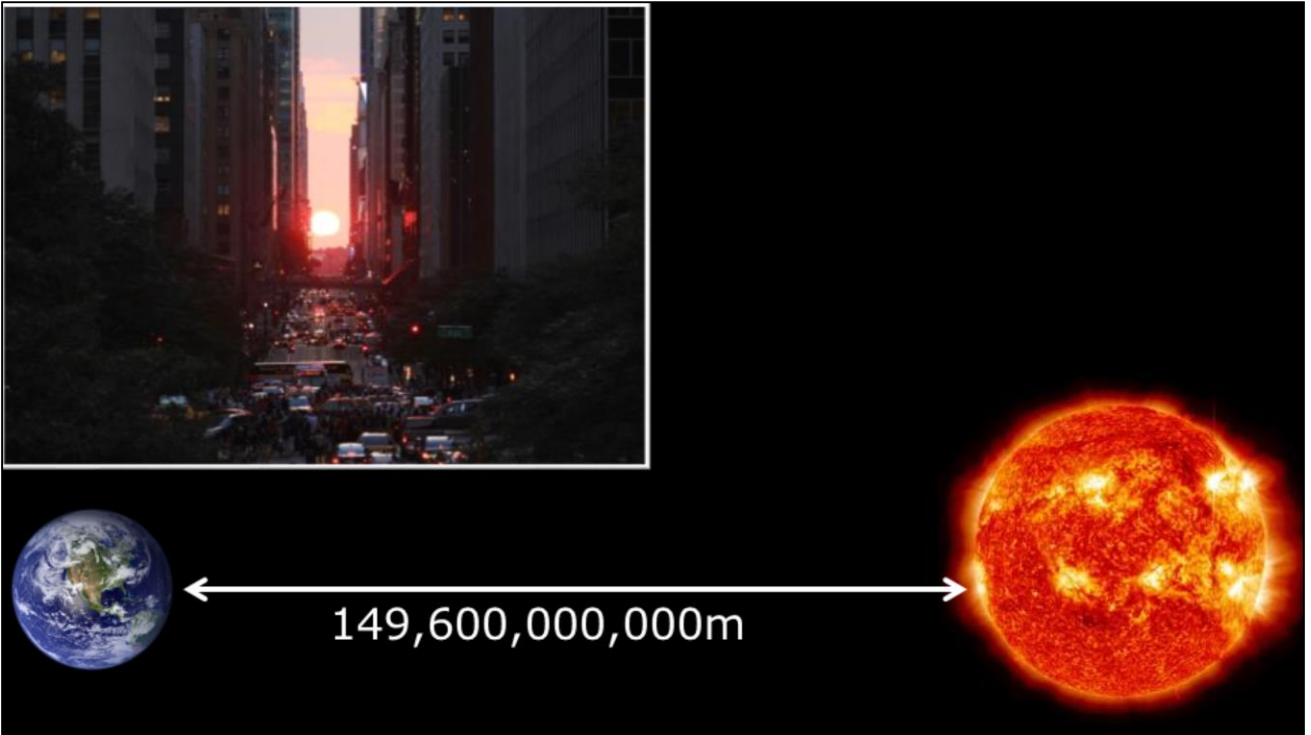
There's a number of things that make hexagons convenient for tiling maps, most of them familiar to tabletop gamers. But there was one additional advantage they had over rectangles for us.



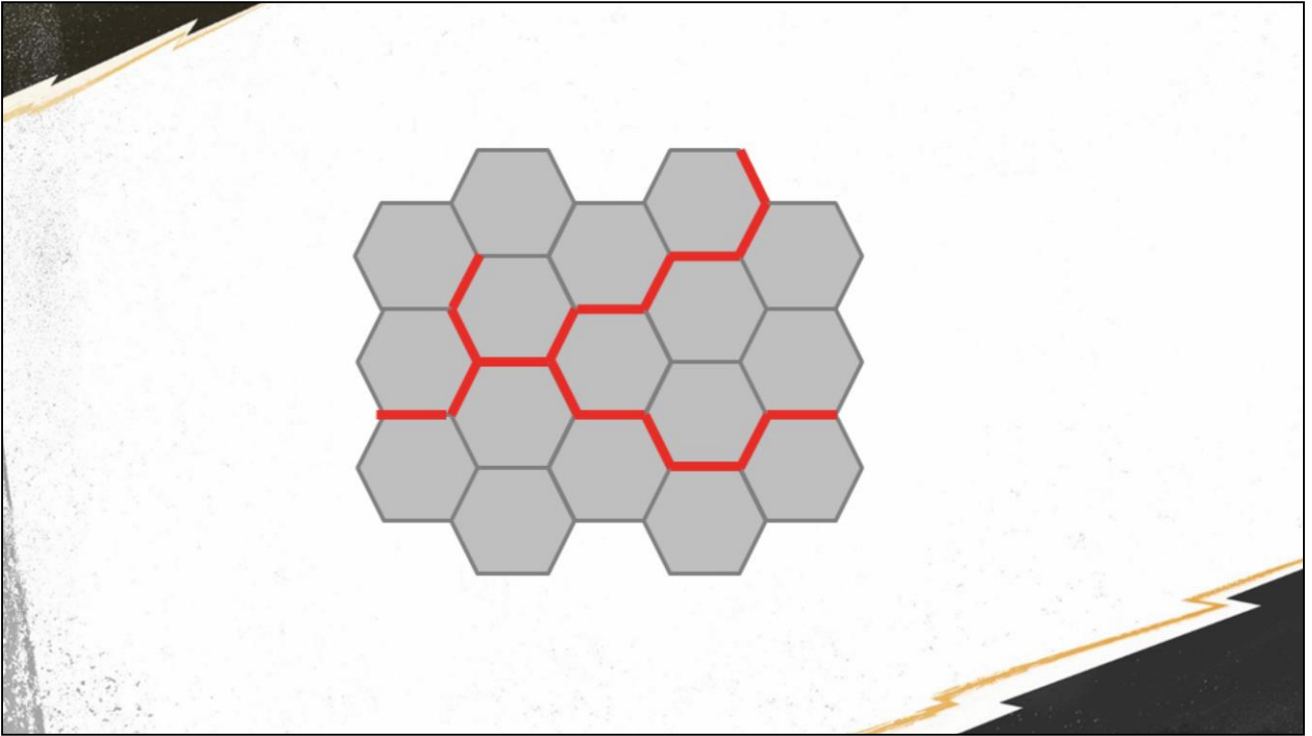
Regular square- or rectangular-shaped city blocks can line up such that you have long, uninterrupted sightlines down linear streets. That means long draw distances and a lot of work for the renderer.

Manhattanhenge photo credit: C'Est La Vie, Annie:
<https://cestlavieannie.wordpress.com/2013/05/30/manhattanhenge/>

Manhattan map via Wikimedia Commons:
[https://commons.wikimedia.org/wiki/File:1899_Home_Life_Map_of_New_York_City_\(Manhattan_and_the_Bronx\)_-_Geographicus_-_NYC-HomeLife-1899.jpg](https://commons.wikimedia.org/wiki/File:1899_Home_Life_Map_of_New_York_City_(Manhattan_and_the_Bronx)_-_Geographicus_-_NYC-HomeLife-1899.jpg)



That's Manhattanhenge, the day on which the setting sun aligns with New York's grid streets and shines from one end of the city to the other. Effectively you've got an object 93 million miles away in your line of sight, or a 150 million kilometer draw distance. Wouldn't fit in a 32 bit float.

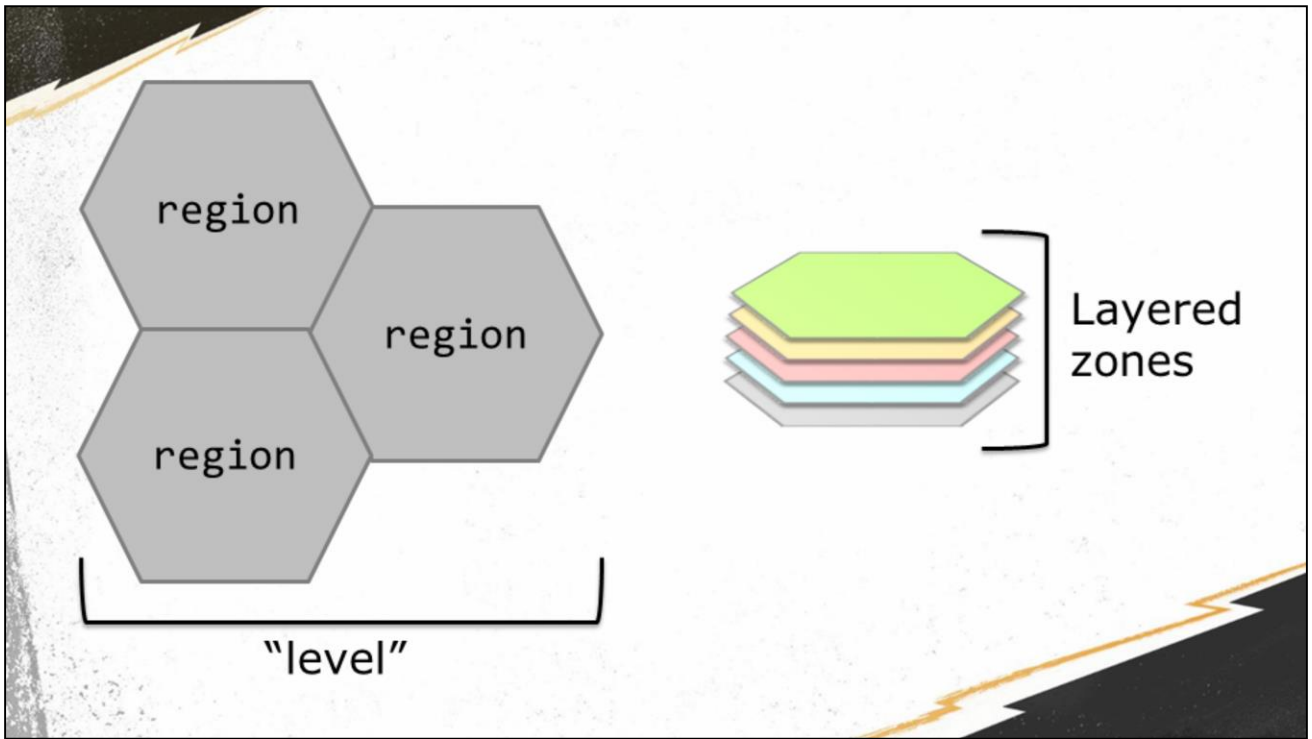


If people build structures to a hexagonal map, the natural boundaries will tend to be roughly aligned to hexagonal angles as well, and you'll naturally limit sightlines as people put 60 degree angles in everywhere. It'll still look nice and fit together organically.



Engine Overview

Here's a quick overview of the game runtime tech.



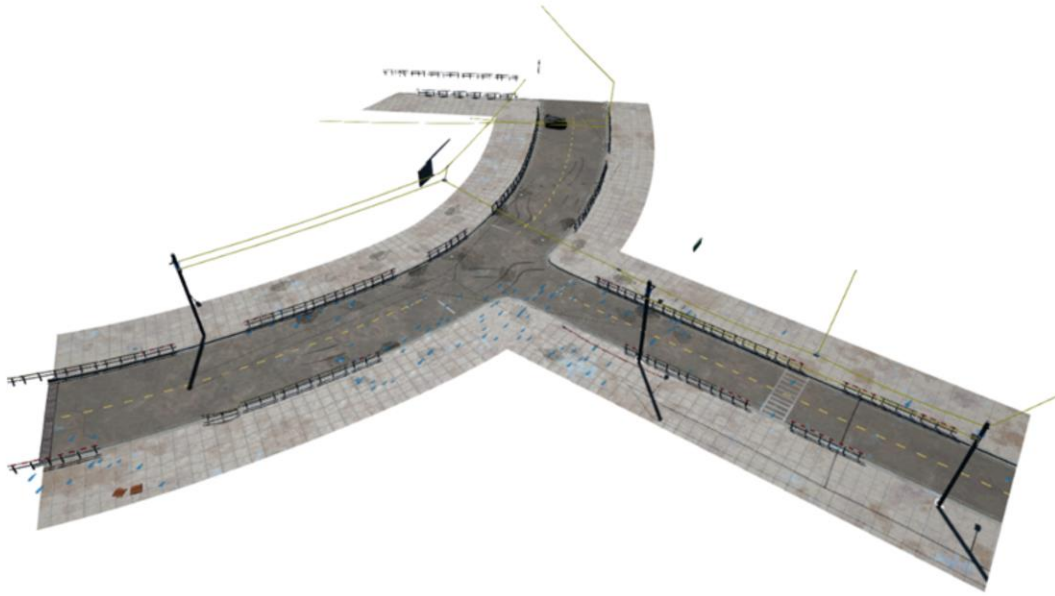
Our engine treats "levels" in such a way that only one can be loaded at a time. So for Sunset, we only had one level ever – `sunset_city.level` – and then we divided that into hexes, or "regions" (we use the term interchangeably).

Like in Ratchet, our world was built from overlapping zones, which would be built up to represent a locale in space like a stack of pie plates or animation cels.



Let's take a quick look at how a hex gets built up

hex210_ground.zone



Verts:0 Tris:0 Visual Effects:0 Models:0 Lights:0 Actors:0 Variation ID:0
Cam Mode:Way Cam:253.3 59.7 -43.6 Far Clip:1000 FPS:12 PID:22100

Here's the ground zone.

hex210_art_1.zone



Verts:541 Tris:620 Visual Effects:0 Models:1 Lights:0 Actors:0 Variation ID:0
Cam Mode:Way Cam:186.1 21.4 27.3 Far Clip:1000 FPS:29 PID:22180

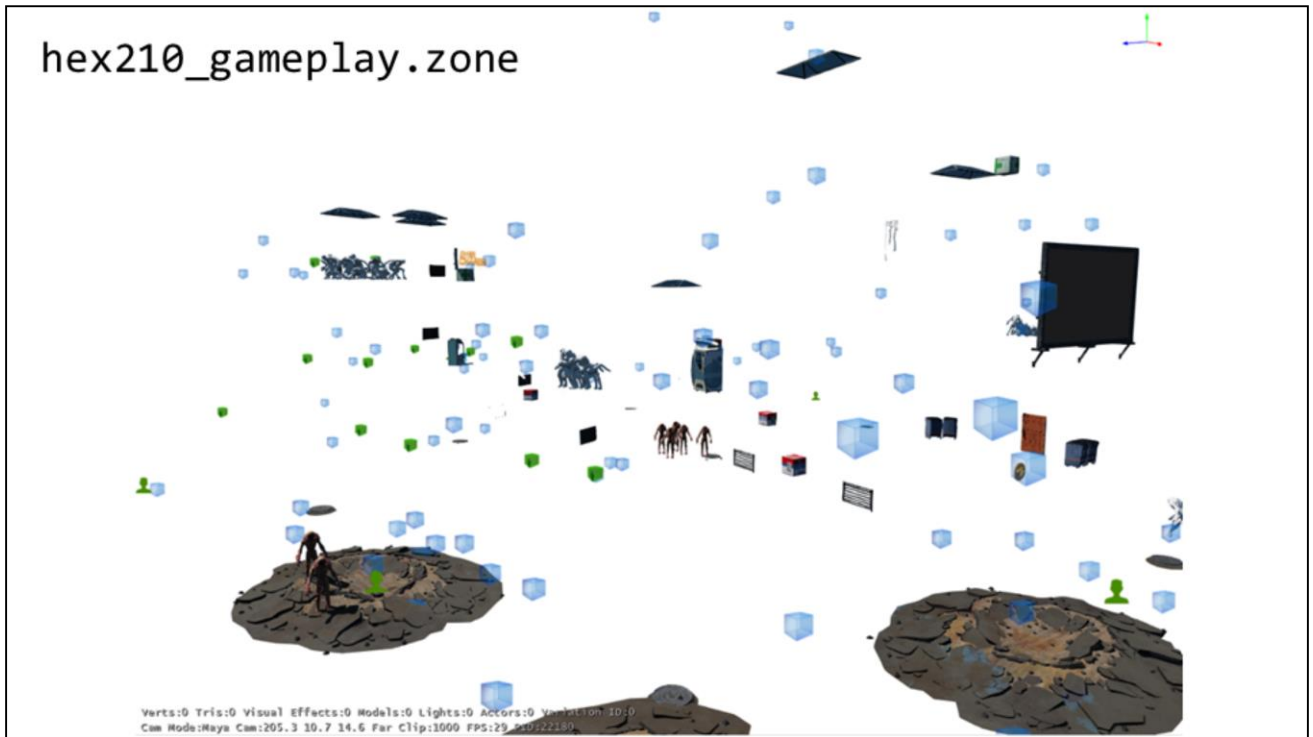
Art goes into its own zone.

hex210_art_4.zone



Verts:0 Tris:0 Visual Effects:0 Models:0 Lights:0 Actors:0 Variation ID:0
Cam Mode:None Cam:218.5 11.9 -16.8 Far Clip:1000 FPS:29 PID:22100

We can have several art zones in a hex, each of them adding another layer.



And here's the gameplay zone. You can see it has enemies in it. The blue boxes represent the origins of abstract things, like spawners and volumes and script objects.

Gameplay zones contain:

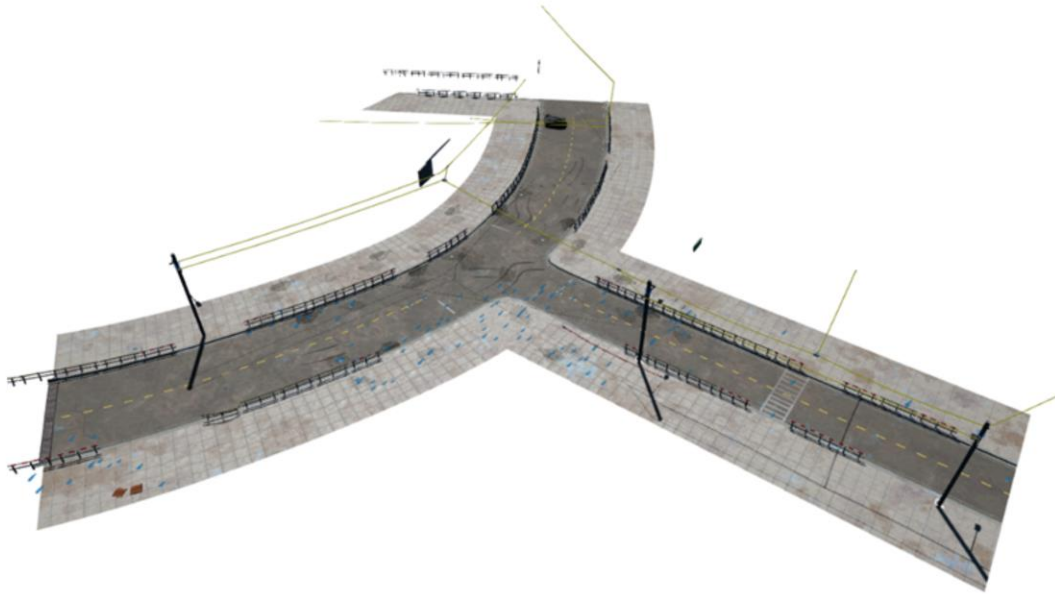
- Baseline enemies
- Item spawners
- Interactive objects
- Script
- Volumes
- Anchors for dynamic encounters



Gameplay zones contain a lot of what makes the world seem alive. They have the logic for spawning NPCs and pigeons and interactable objects. When you run into a region, the gameplay zone for that region is what fills the streets with OD.

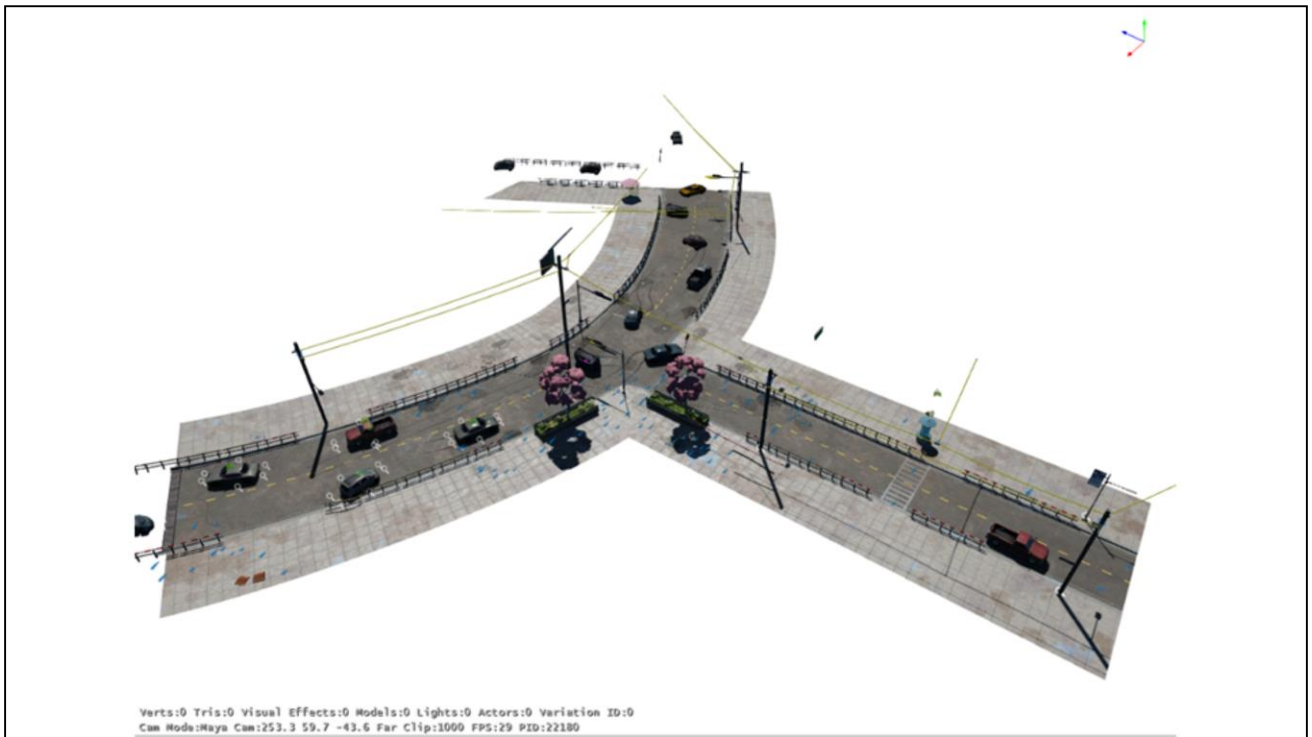
They also contain markup that global systems can hook into. For example, we have a global system that is looking around for places to put dynamic encounters – small mini-setups with unique content. It iterates over gameplay zones as they're loading, and looks for "dynamic encounter goes here" nodes that it can place the encounters into.

hex210

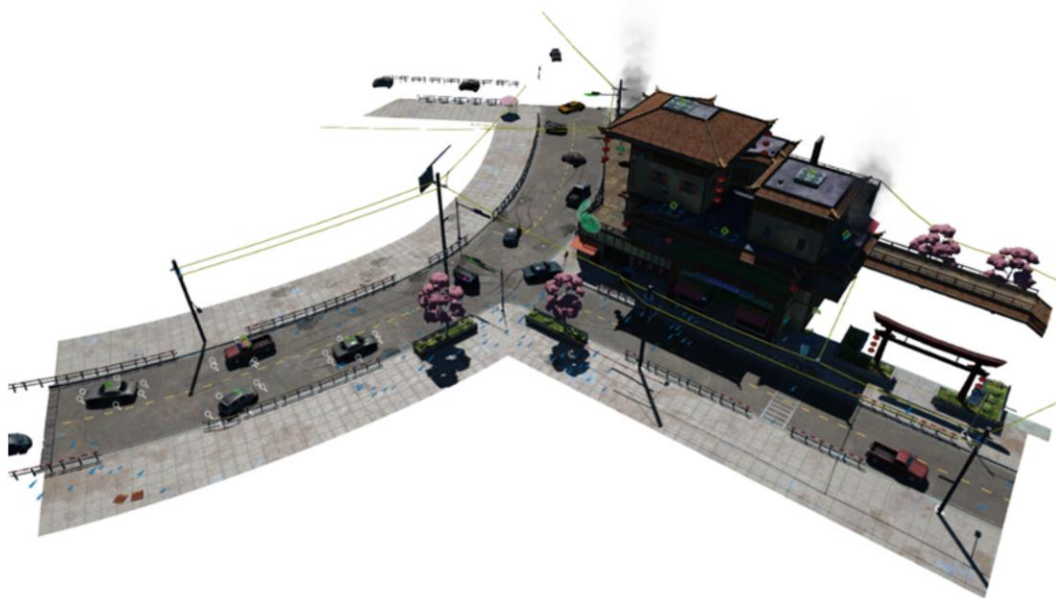


Verts:0 Tris:0 Visual Effects:0 Models:0 Lights:0 Actors:0 Variation ID:0
Cam Mode:Way Cam:253.3 59.7 -43.6 Far Clip:1000 FPS:12 PID:22100

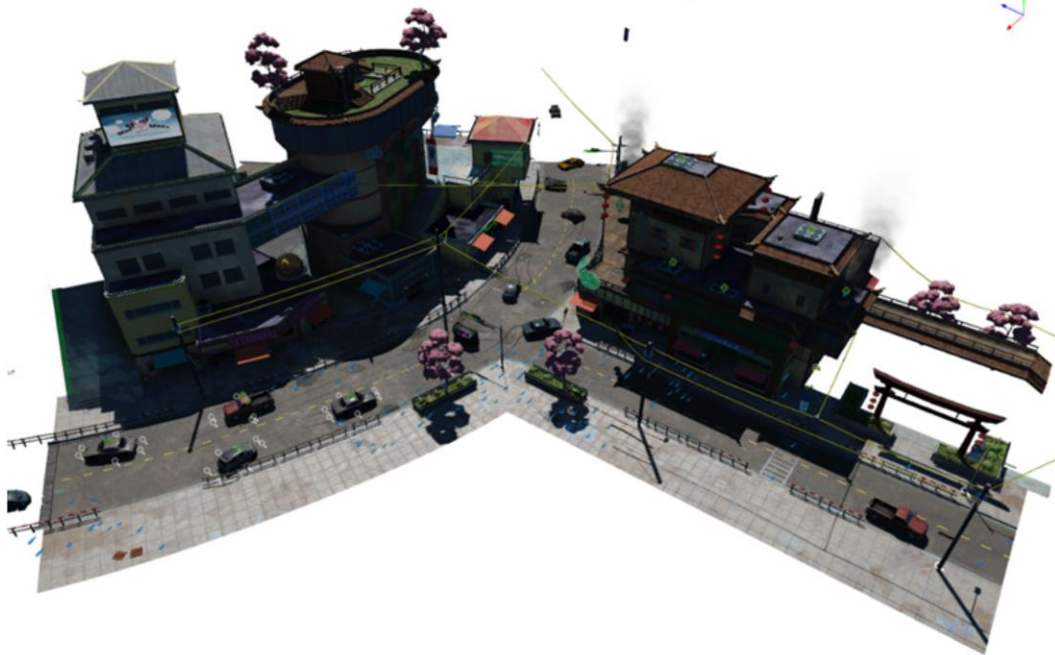
A region has many zones; so all the stuff in the zones gets added together on load.



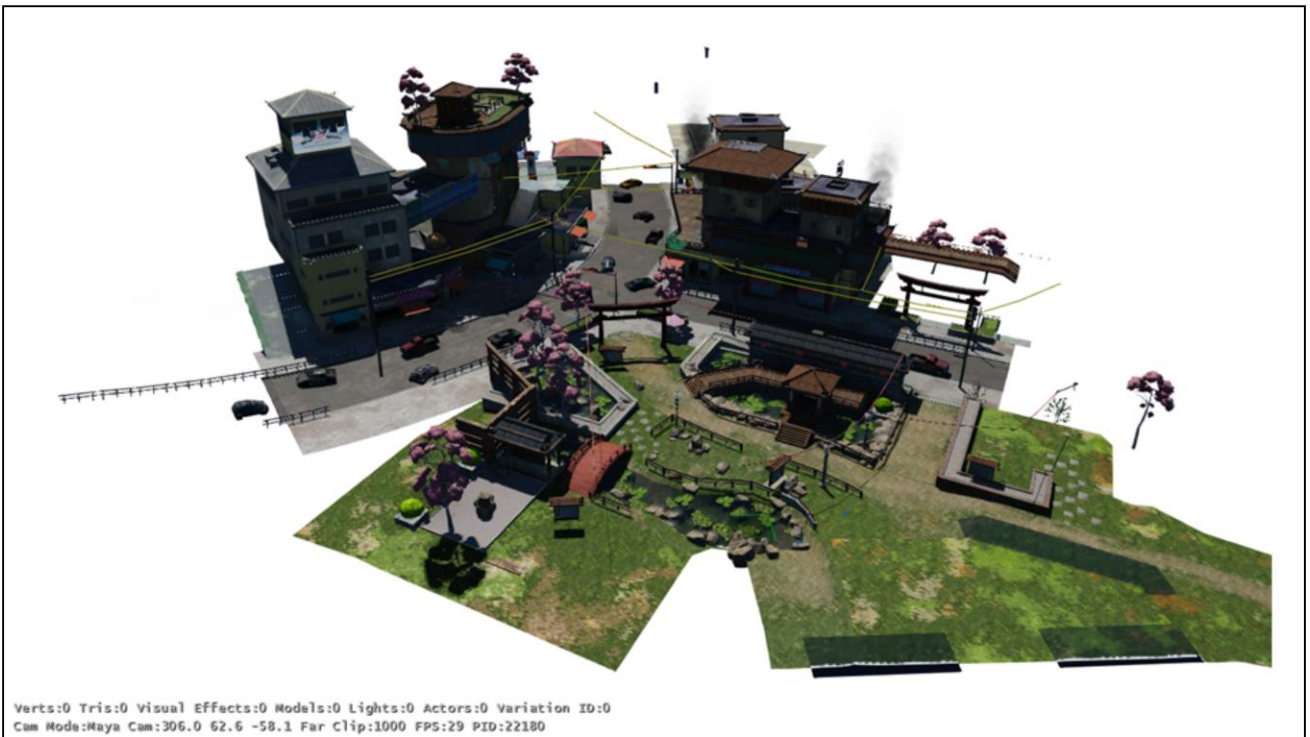
So putting it all together one layer at a time...



Verts:0 Tris:0 Visual Effects:0 Models:0 Lights:0 Actors:0 Variation ID:0
Cum Mode:Way Cam:253.3 59.7 -43.6 Far Clip:1000 FPS:29 PID:22100



Verts:0 Tris:0 Visual Effects:0 Models:0 Lights:0 Actors:0 Variation ID:0
Cam Mode:Way Cam:253.3 59.7 -43.6 Far Clip:1000 FPS:16 PID:22180

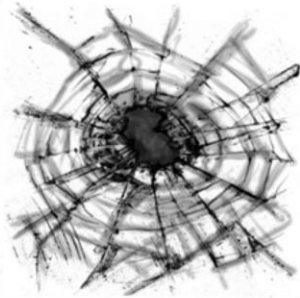


You can see here that one of the zones actually sticks a little bit outside the hex – it's like a flange to cover the seam between zones. Zones don't actually have to be constrained to a region; any zone can contain an object at any point in the world. That's useful for mission-specific behavior and some other purposes – I'll get back to that soon. But even for ordinary geographical streamed zones, having a bit of them stick into neighboring hexes like this is a convenient way to cover seams.

Scene objects



Actor



Decal

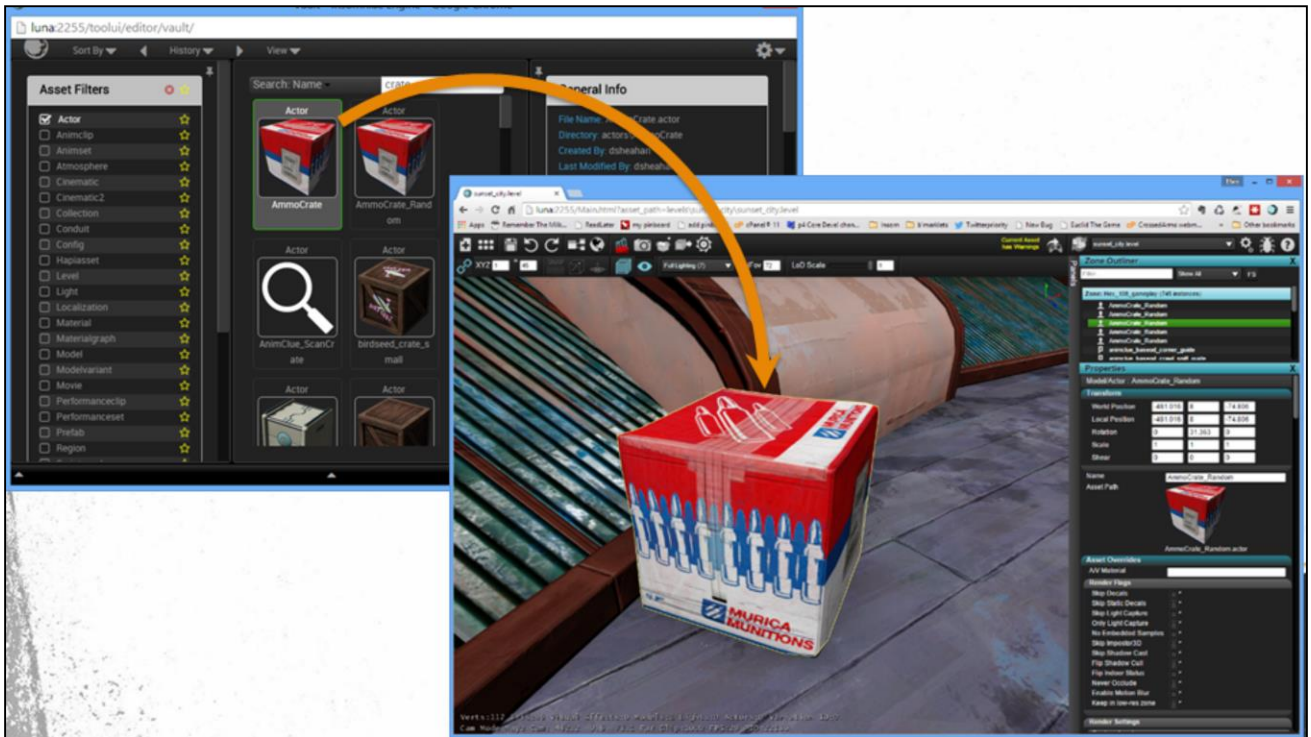


Model

Visual
Effect

Our engine's basic atom of simulation is the "scene object."

These are: actors, lights, volumes, model instances, visual effects, stand-ins, script entities, decals, etc. Anything that has a position, an asset, and needs to engine to tick it.



Content producers add things to the world by selecting an object type and dragging it into the world in the level editor. This creates a "node" in the zone, which is a generic representation of an object.

```

Hex_108_gameplay.zone:
{
  "SceneNodes":
  {
    "0x801031e346e3149b":
    {
      "AssetPath":
        "actors/AmmoCrate/AmmoCrate_Random.actor",
      "AssetType": "kModel",
      "Id": "0x801031e346e3149b",
      "LocalTransform":
      {
        "EulerRotation":
        { "Y": 90 },
        "Position":
        {
          "X": -4.92114655e+02,
          "Y": 1.60002728e+01,
          "Z": -2.00374374e+01
        }
      }
    },
    "Name": "AmmoCrate_Random"
  }, ...
}

```

Properties

Model/Actor : AmmoCrate_Random


Transform

World Position	-481.016	8	-74.806
Local Position	-481.016	8	-74.806
Rotation	0	31.363	0
Scale	1	1	1
Shear	0	0	0

Name

AmmoCrate_Random

Asset Path



AmmoCrate_Random.actor


Asset Overrides

Actor Overrides

AssetType

kModel

AssetPath



A node is just a dictionary of keys and values. Some of the fields contain simple data, like position. When you place an object into the world, you set keys and override values to describe the object. For example, you set its position.

Some fields are references to other assets. This scene object references an actor.

[CLICK] The actor in turn has some components on it, and we could override configuration values on the components on this instance if so desired.

The zone is essentially a list of such nodes – it doesn't contain assets directly; it contains nodes, and those nodes may refer to assets.

```

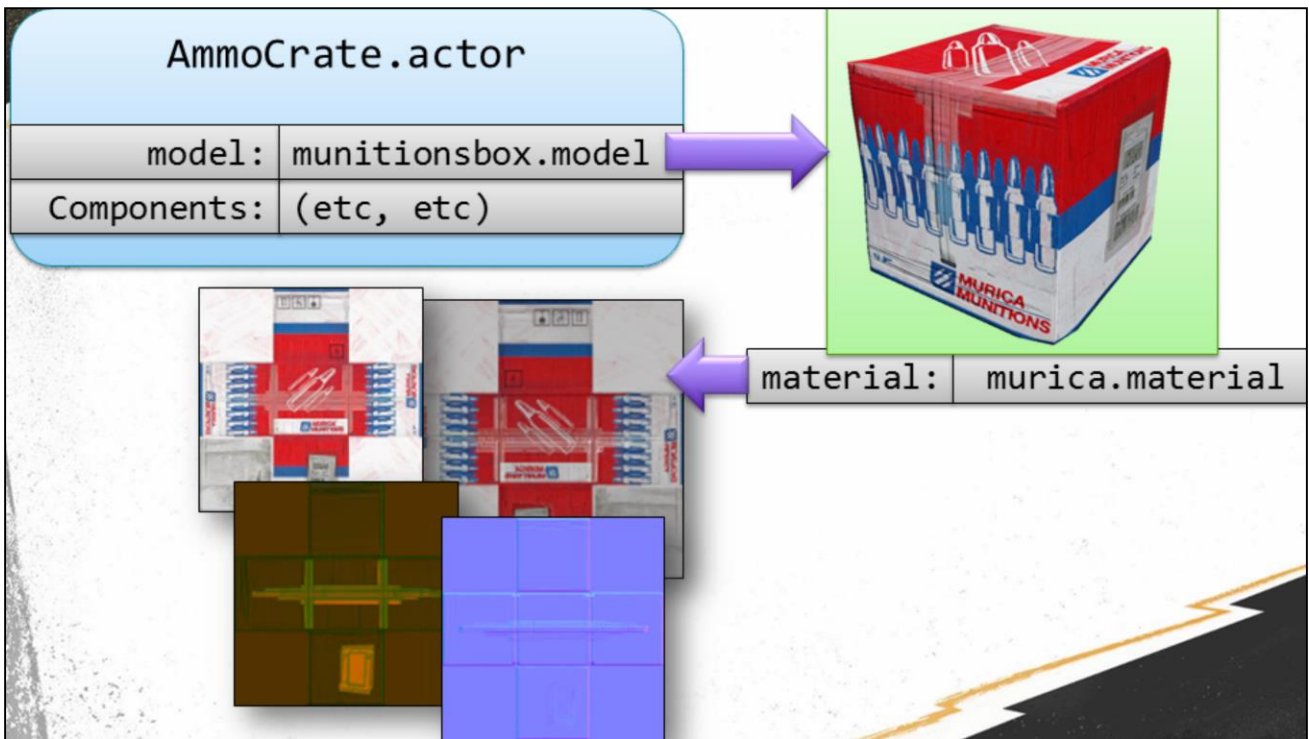
AmmoCrate_Random.actor:
{
  "AssetPath":
    "objects/manmade_single/gameplay/np_muricaMunitions_box/np_muricaMunitions_box.model",
  "AssetType": "kModel",
  "Components":
  [
    {
      "Overrides":
      {
        "AreaDamage": 0,
        "AreaDamageRadius": 0,
        "ChunkExitVelocity": 3.50000000e+00,
        "ChunkLifeTimeMax": 8,
        "ChunkLifeTimeMin": 4,
        "DebrisAngularVel": 5,
        "DebrisExitVelocity": 6,
        "DebrisPerChunk": 3,
        "DeleteOnDestroy": true,
        "DestroyedChunkModels":
        [
          "objects/manmade_single/gameplay/np_muricaMunitions_box/np_muricaMunitions_box_chu
          "objects/manmade_single/gameplay/np_muricaMunitions_box/np_muricaMunitions_box_chu
          "objects/manmade_single/gameplay/np_muricaMunitions_box/np_muricaMunitions_box_chu

```

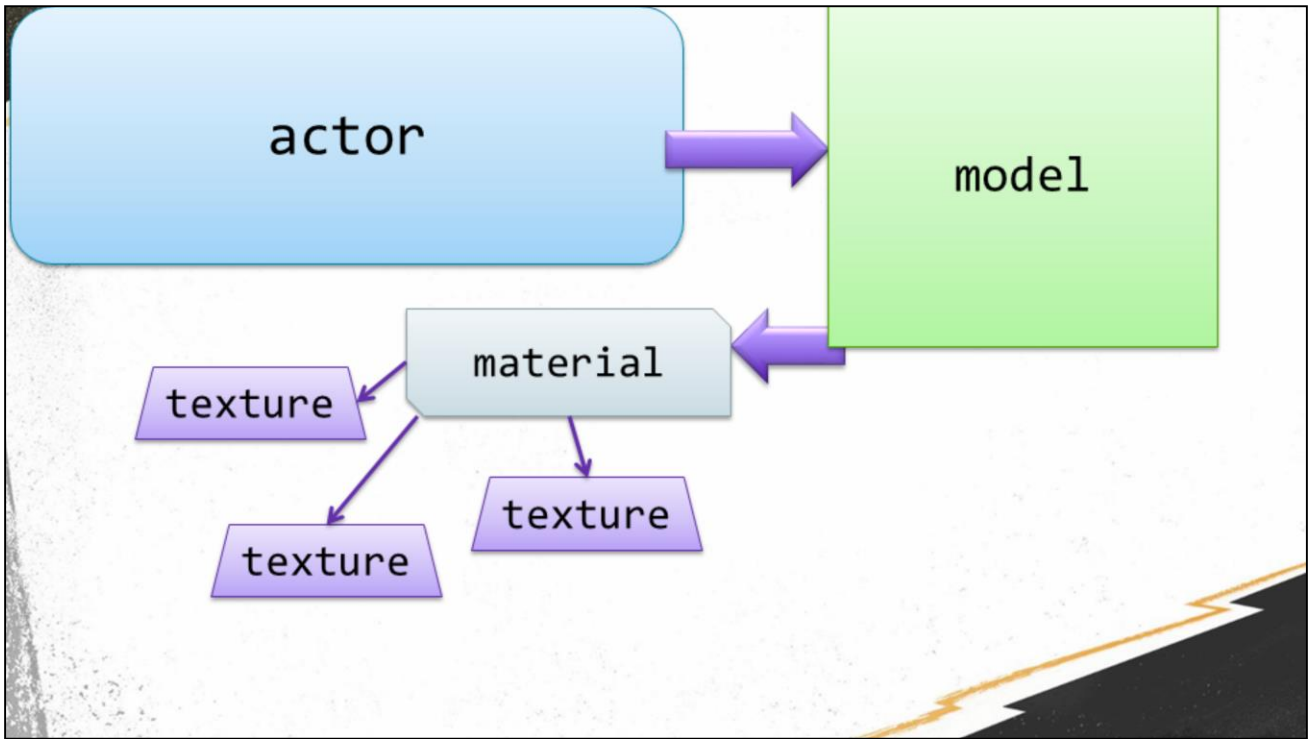
That .actor asset in turn is itself a blob of key-value data. Some of those values may reference other assets...

```
np_muricaMunitions_box.model:
{
  "DaeFilename":
    "objects/manmade_single/gameplay/np_muricaMunitions_box/np_muricaMunitions_box.dae",
  "MaterialMap":
    [
      {
        "AssetMaterialName": "",
        "SourceMaterialName": "lambert1"
      },
      {
        "AssetMaterialName": "materials\\np_fizzieFresh_box\\np_muricaMunitions_box.material",
        "FadeOutDist": 100,
        "SourceMaterialName": "np_muricaMunitions_box"
      }
    ],
  "NavProperties":
    { "SkipNavGeneration": true },
  "SkelLocators":
    [
      { "Name": "igLoc_chunk01" },
      { "Name": "igLoc_chunk02" },
      { "Name": "igLoc_chunk03" },
      { "Name": "igLoc_chunk04" }
    ]
}
```

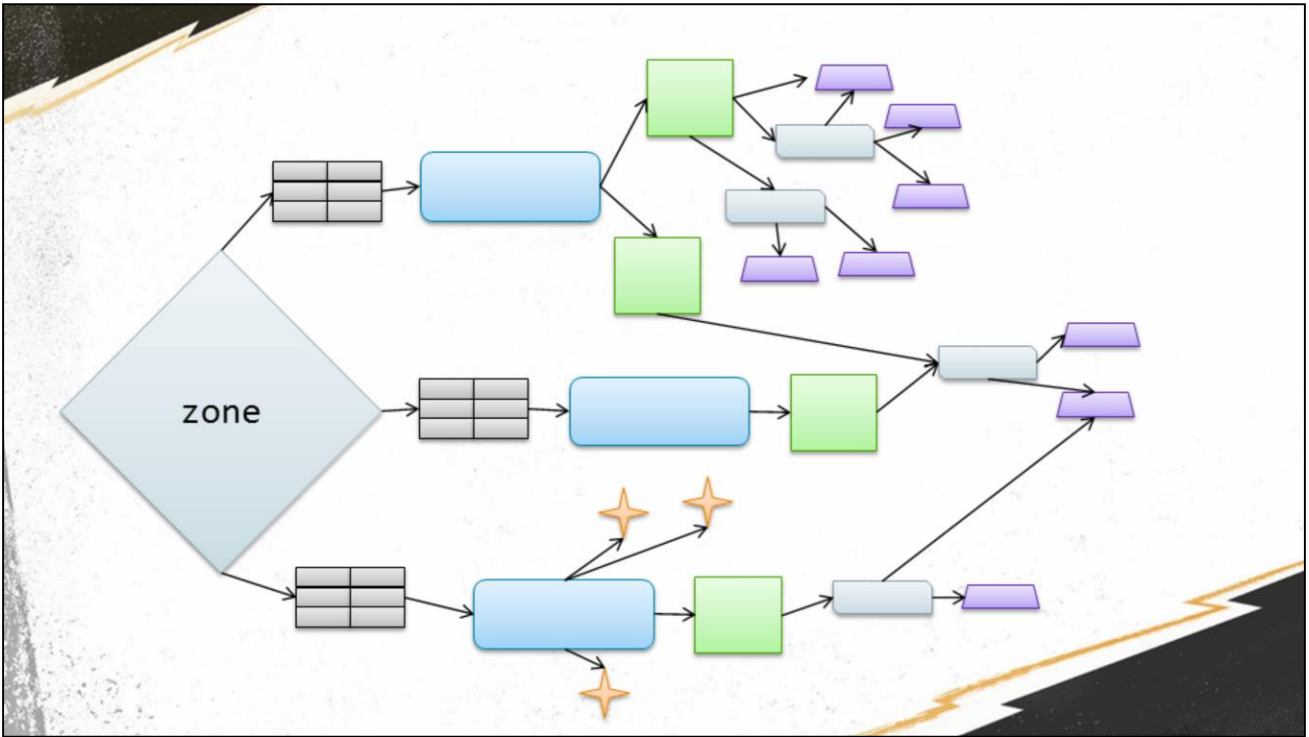
..which in turn have their own values, and so on.



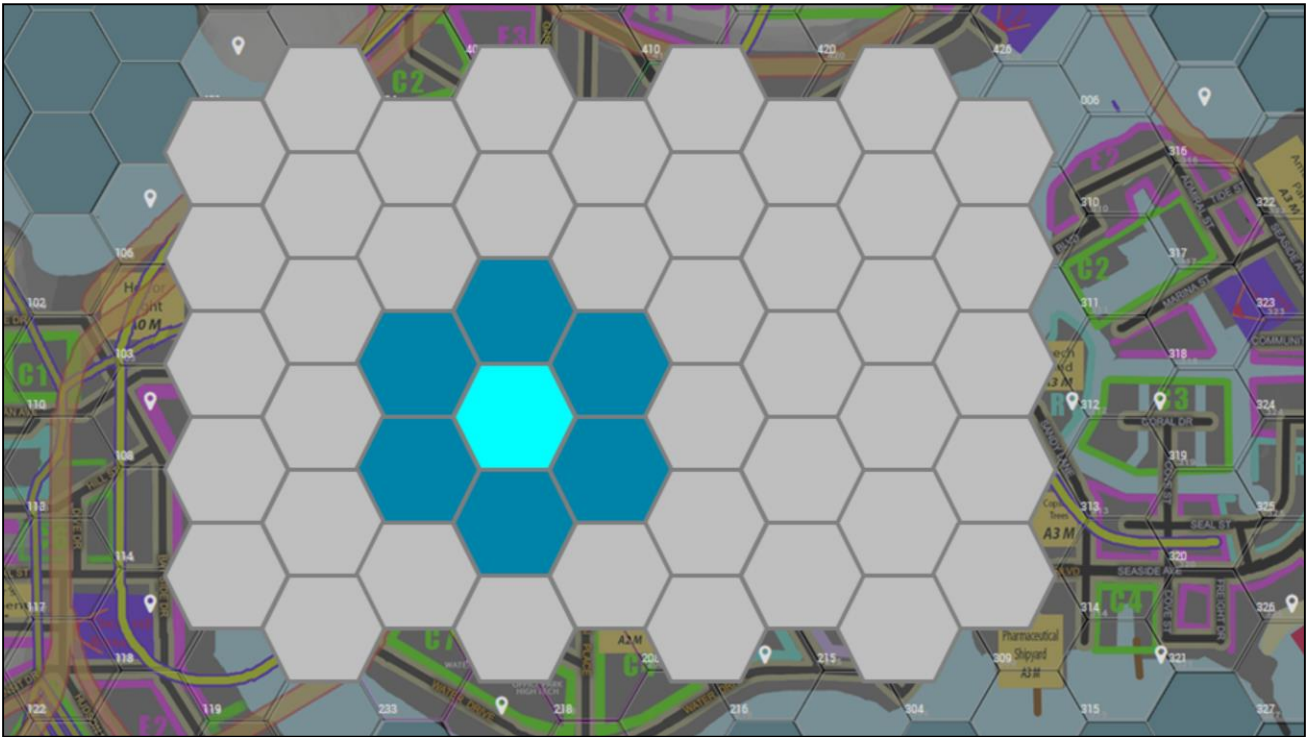
The important thing is that all our objects are built in a consistent way: **[CLICK]** everything is a JSON blob of dictionaries; keys point to values, to other dictionaries, or to assets. Eventually some assets will have fields pointing to binary data, like textures.



(show for a second and skip past)



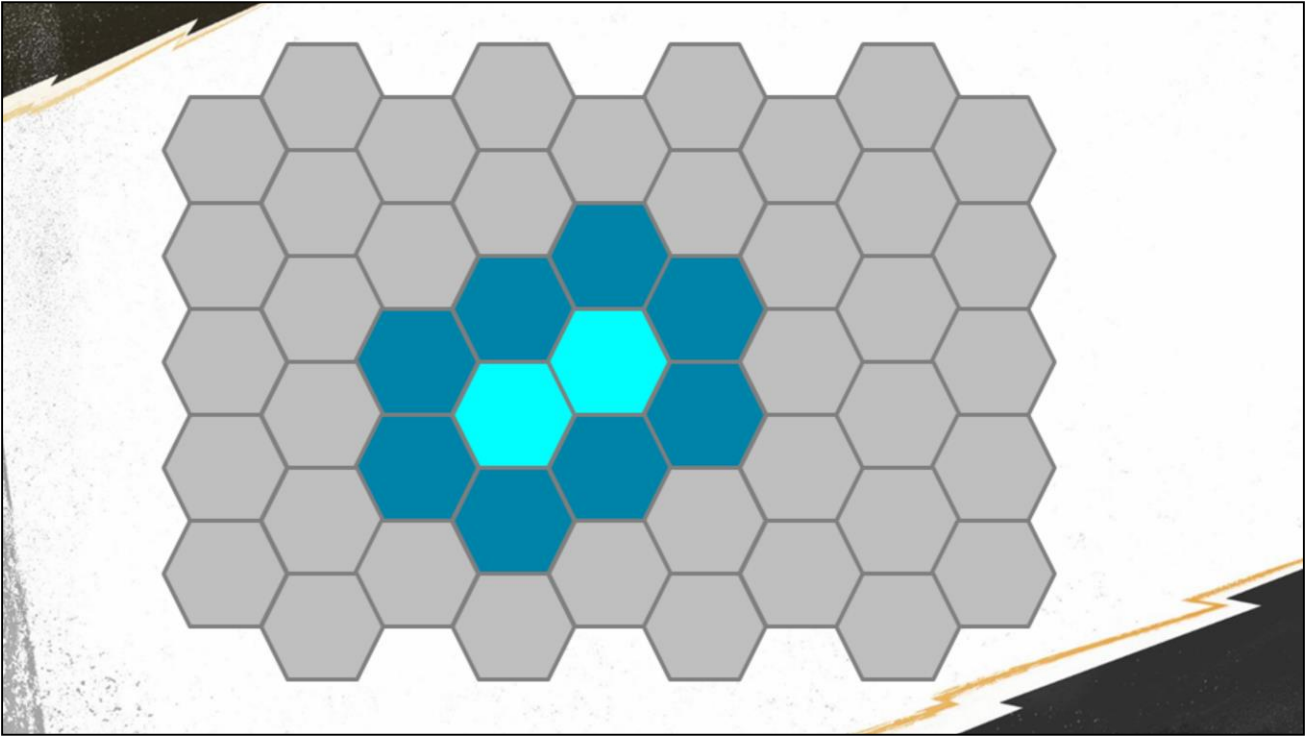
So we have a uniform way of asking, for each asset, what other assets it uses. Thus each zone forms the head of a directed acyclic graph where some things can be linked to multiple times via different paths. The zone file itself doesn't reference base asset types. It stores a list of instances, the data that makes each instance unique, and the instances in turn reference their assets.



The easiest way to ensure that the player can move into any adjacent hex at any time is to have them loaded. That's also the easiest way to make sure you're surrounded by high resolution assets. We load the hex in which the player stands, along with the surrounding 6 hexes, so that there are always at least 7 hexes loaded at a time.

When the player moves from one hex to the next, we load in all of the new neighboring hexes, and evict the ones left behind.

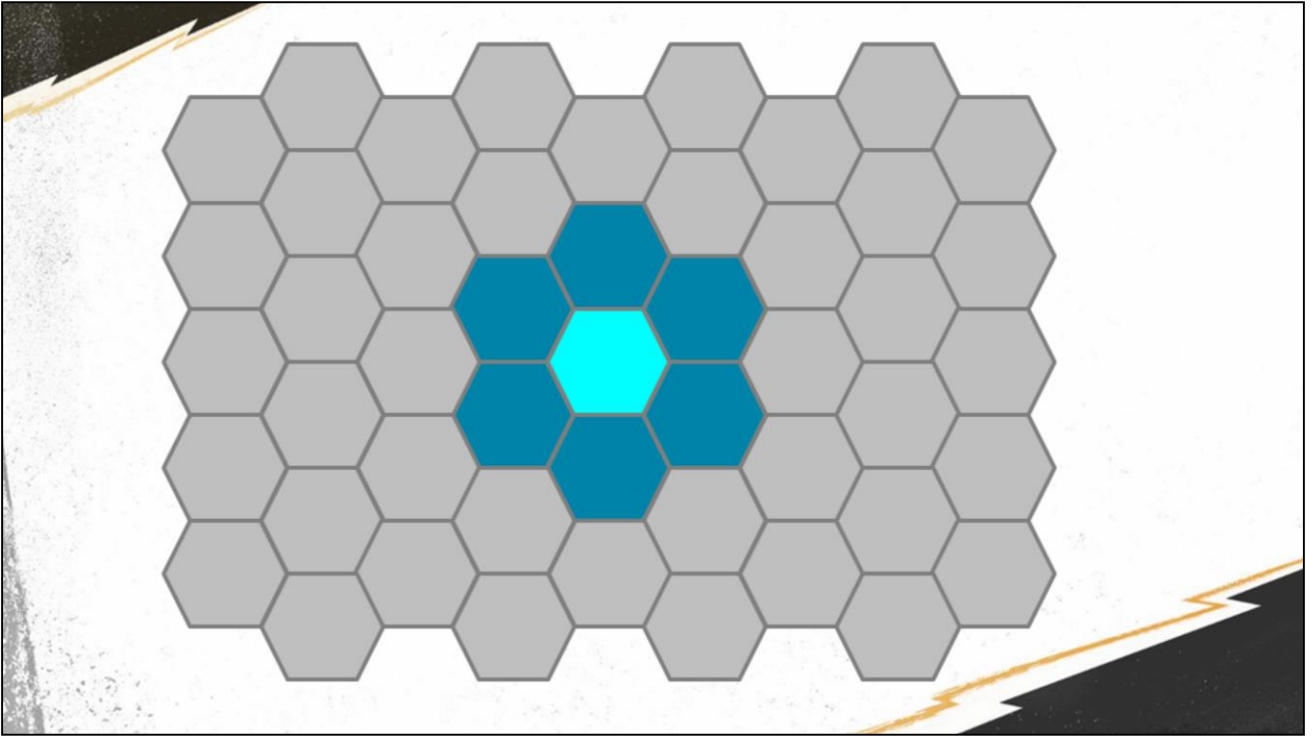
This means you can temporarily have 10 hexes loaded at once (and while moving continually, you usually do).



We load the hex in which the player stands, along with the surrounding 6 hexes, so that there are always at least 7 hexes loaded at a time.

When the player moves from one hex to the next, we load in all of the new neighboring hexes, and evict the ones left behind.

This means you can temporarily have 10 hexes loaded at once (and while moving continually, you usually do).



We load the hex in which the player stands, along with the surrounding 6 hexes, so that there are always at least 7 hexes loaded at a time.

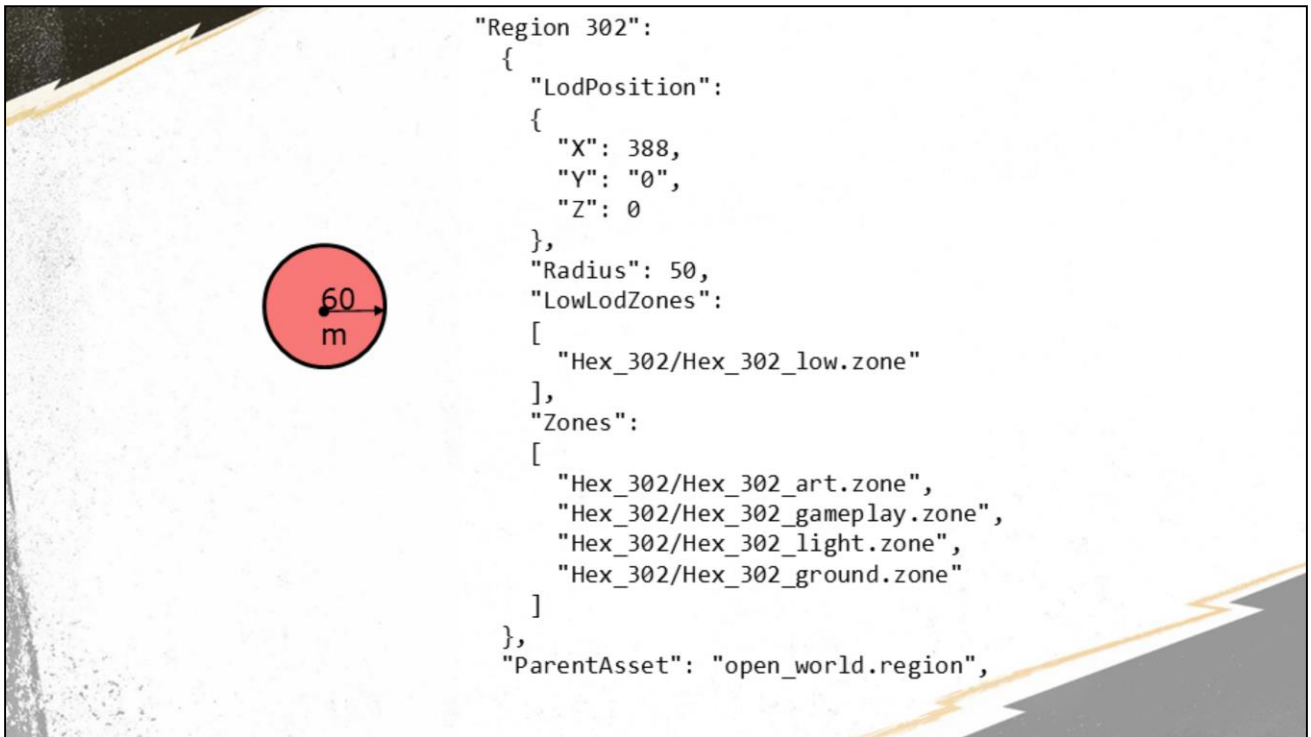
When the player moves from one hex to the next, we load in all of the new neighboring hexes, and evict the ones left behind.

This means you can temporarily have 10 hexes loaded at once (and while moving continually, you usually do).



(actually, they're not hexes)

Well, actually... hexes don't exist in the game runtime. They are purely an editor construct. The game just works with zones.



A region (hex) is simply a point, a radius, and a list of zones. When the player steps into that radius, we load the zones. When the player steps outside the radius, we unload the zones. This was a small step from the way things worked on Fuse. In Fuse, a "region" was also a list of zones; but regions got loaded and unloaded via script. So all we did here was to replace that manual script with a system that checked the player's location relative to...



... all of the overlapping circles in the world, and load the relevant zones. The entire world streaming rests on this very simple amendment to our linear streaming setup.

(actually, the stuff doesn't need to fit in the radius)

The radius just specifies when the zone loads.

Objects in the zone can have any position in the world.



This is useful for missions, where we often have a single global zone representing all the objects necessary for a mission regardless of their position in the world. It also helps us hide seams.

Photo credit: Wikipedia and Leonard G. :
https://en.wikipedia.org/wiki/Cantilever#/media/File:Pierre_Pflimlin_Bridge_UC_Adjusted.jpg



So what does it mean when part of the city is “not loaded”? Obviously we need to have *something* there so that you can see the skyline in the background; we can’t have blocks just wink out of existence.

To portray the city outside the currently loaded region, we have a low-level-of-detail version of every hex.

View this video at: <http://youtu.be/6L8FHxSaYJE>

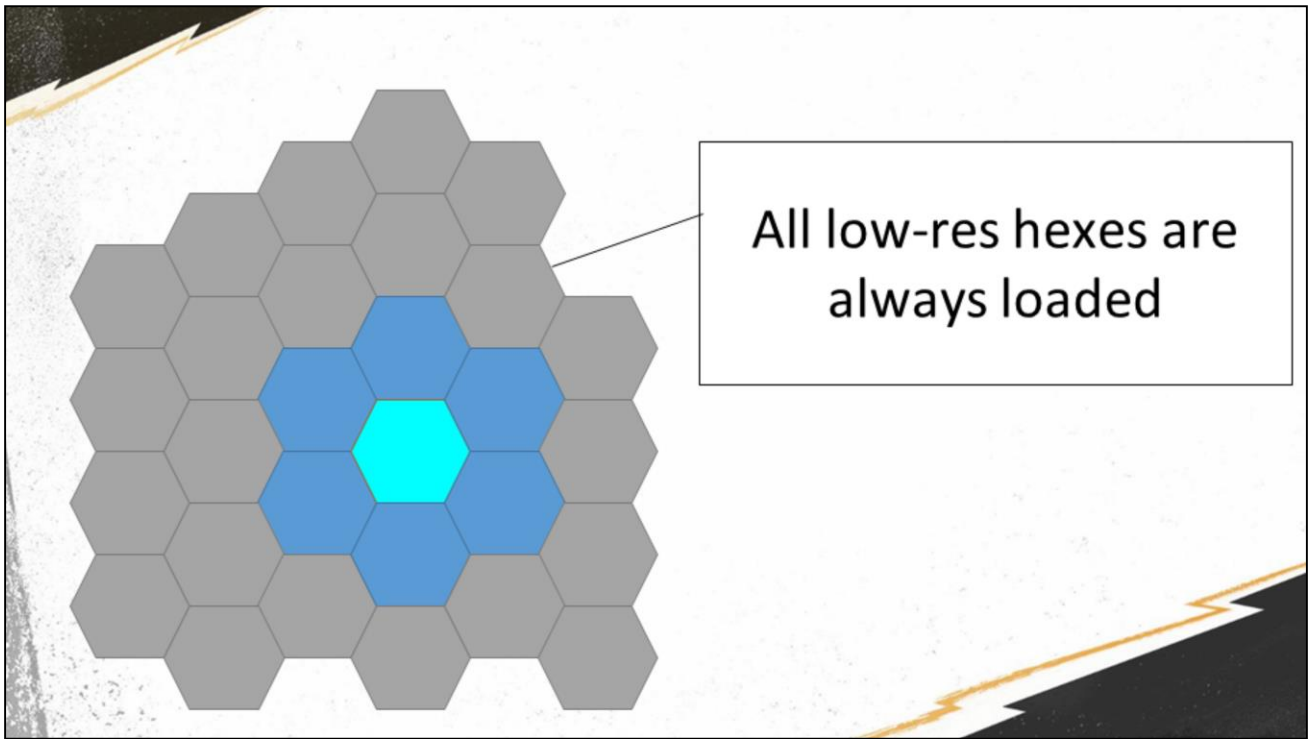


For more information:
"Procedural Techniques
in Sunset Overdrive"

David Santiago
305 South Hall
Thursday, 5:30pm



These "low-res" zones primarily low-LOD models ("stand-ins") automatically generated from the baseline level geometry via Simplygon. If you're curious about that process, David will give you the skinny on it tomorrow, in the room next door.

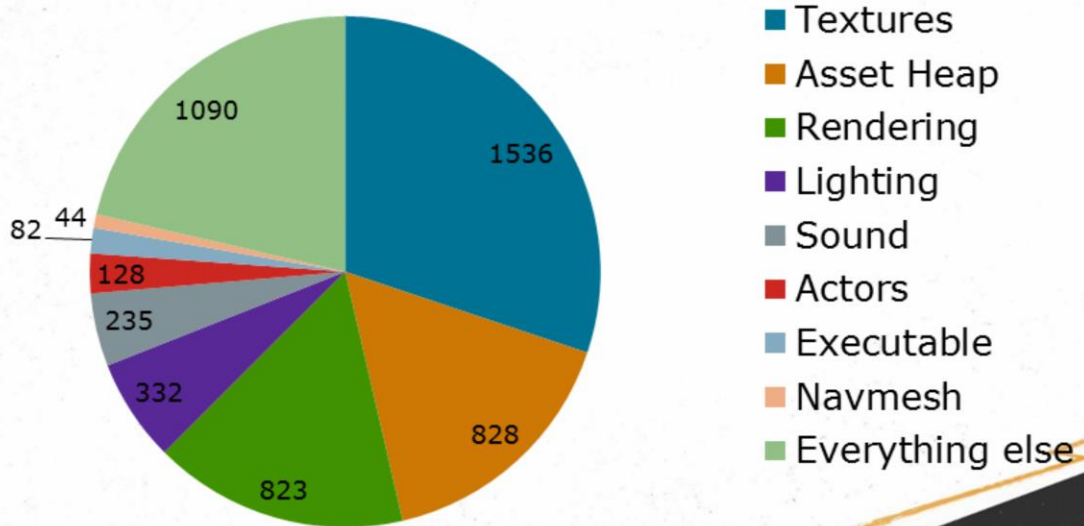


All the low-res zones are always loaded. Each model in the world knows its equivalent stand-in geometry. When the model is beyond a certain distance, we automatically swap it to render the stand-in instead. This way we don't need to stream stand-ins in and out.

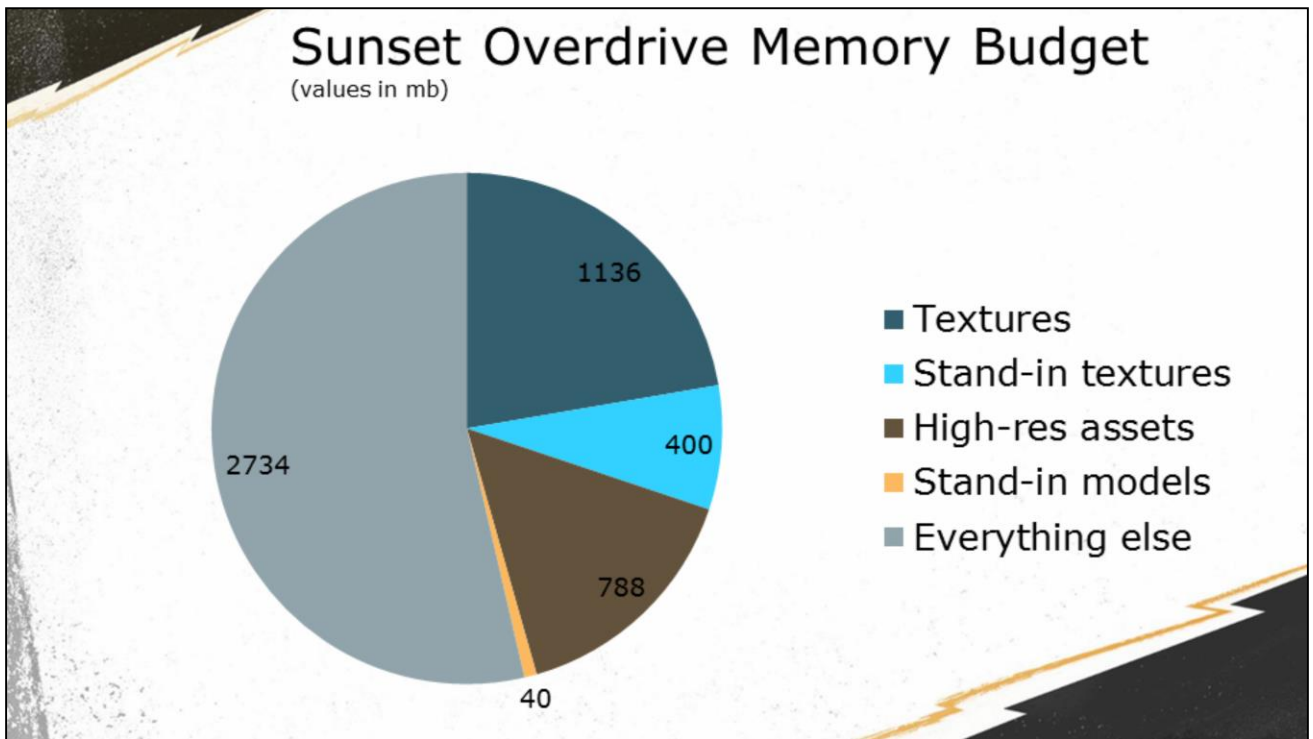
The always-loaded low-res zones was a development time tradeoff – we didn't have time to build a more elaborate system of medium- and low-LOD hexes, and contend with hexes so far away they can be dumped entirely. So, what about the memory cost of keeping all that stuff loaded always?

Sunset Overdrive Memory Budget

(values in mb)



Here's how our memory looks. Textures are the biggest single category, but we are always streaming some textures in and out, parallel to the general streaming system. The highest MIP-levels of textures are huge, too huge to keep even all of them for the objects in just a single zone. So we proximity-stream those for specific models just like every other engine does. Sound does this too.



For a world the city of Sunset City, all the low-res zones and their textures fit into 500mb, and that was good enough for us to simply keep it always resident. It put a little bit of extra pressure on the texture streamer since we had to be juggling high-MIPs more aggressively.

We could have put a lot of work into making four levels of LOD, and some tiered streaming system, but



you don't have to go putting pineapples and teriyaki sauce on a pizza just because you can. Simple is better. Simple is fine. Simple is good enough to be tasty.

... I saw a place in Los Angeles that puts avocado on pizza. *Avocado*. It's a sin.

Photo from Wikipedia – Ewan Munro -
[https://en.wikipedia.org/wiki/Pizza#/media/File:Fox,_Dalston,_London_\(3634023284\).jpg](https://en.wikipedia.org/wiki/Pizza#/media/File:Fox,_Dalston,_London_(3634023284).jpg)

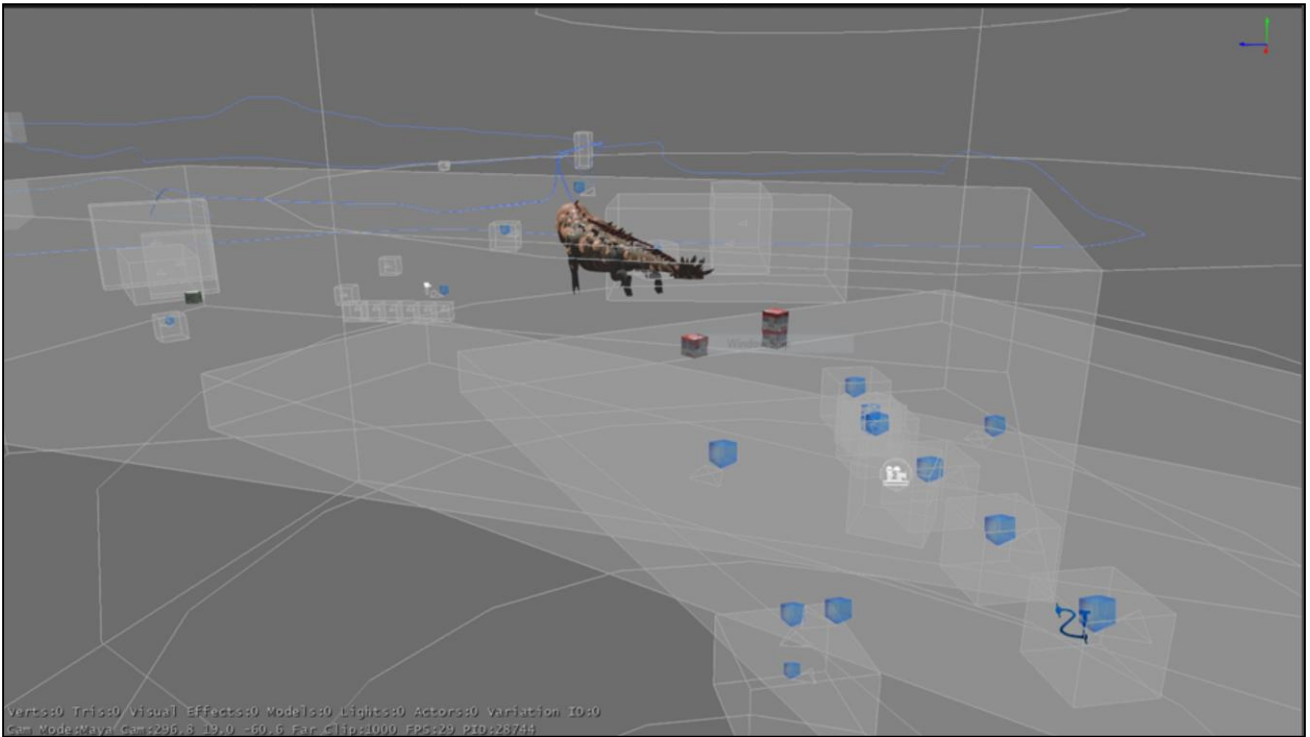


Missions and Overlays

Ours is a mission-based game; there's a plot, and a lot of content that is unique to each point in that plot. We needed a way to add that unique content to parts of the world only when the relevant part of the game was active, and then remove them after. We learned pretty quick that we couldn't just have all mission-specific content be part of the ordinary streamed hexes. We could turn spawning and behavior on and off depending on which missions were active, but all of the spawners and scripts and volume take up a lot of memory, memory that we don't need to be consuming unless that mission is actually underway.



So the simplest next step we could take was to say, each mission gets a zone of its own too. Zones can overlay anything in to the world while they are loaded – actors, effects, script, even entire buildings. So, we'll say the mission gets a "global" zone. It is loaded and stays resident the entire time the mission is active; and it has all the actors and so forth that makes the mission go.



Here's an example of a global mission zone for a boss battle. You can see it has the boss, and curves and spawners and volumes and script objects.

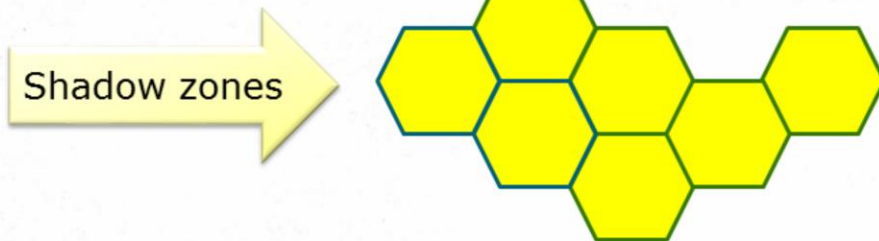
Having a single additional global zone for the mission like this is convenient, because it means all of your data in one place. But it's also problematic – for one thing, this zone may take up a lot of memory, especially if it has custom assets or level geometry.

There isn't a convenient way to link stuff in this global zone to anchor points in the underlying world streaming zones, like we had with the dynamic encounters. Sometimes you'd want to spawn a couple dozen enemies into the street when something happens; and there's already a lot of enemy spawners in the world, so you might want to just tell the existing spawners "dispense ten enemies." Or sometimes you might need to turn them off so they don't contend with all your custom enemies. But it's inconvenient to refer to things in specific world hexes from a zone that's not part of the world.

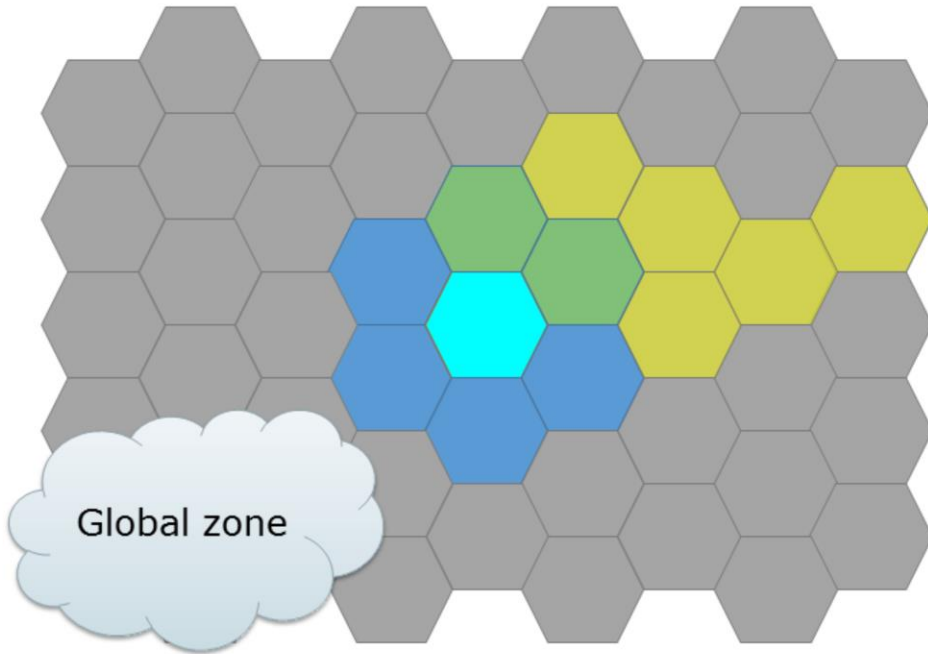
--

Also, it makes it more difficult to keep mission-specific things lined up with level geometry. Let's say your mission adds enemies to a particular zone, and you want those enemies to spawn perched on ledges. If the environment artist moves those ledges, it'll break the mission. It's helpful to have some clear linkage between a geographical zone, and any mission overlay zones that touch it, so that environment artists know what they have to go touch up when they change level layout.

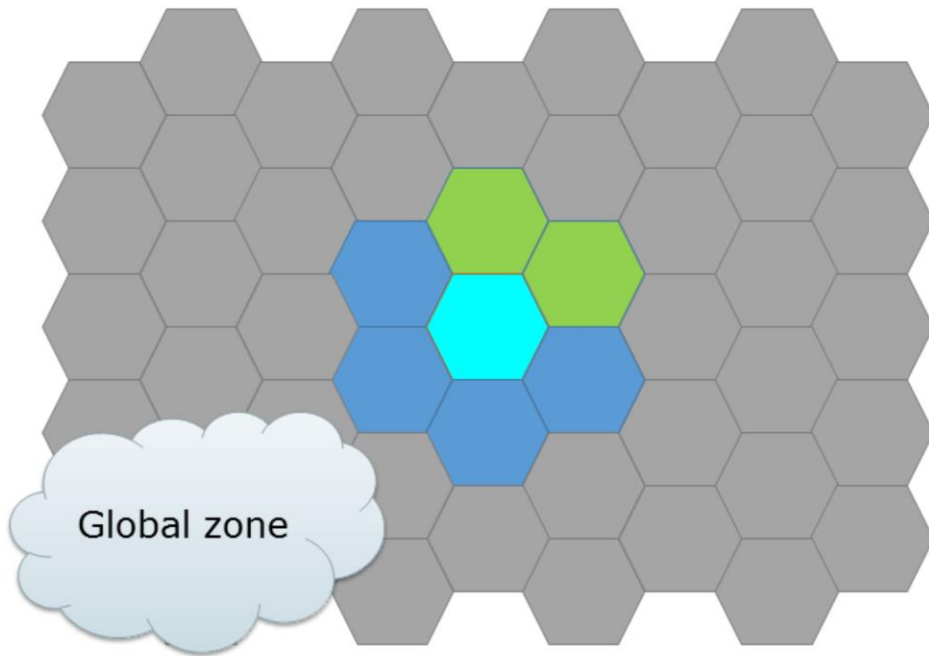
Mission X



So we invented shadow zones. They're so called because they "shadow" existing world zones. Each one has a reference to an ordinary world hex.



When that base hex is loaded AND the mission is active, the corresponding shadow zone loads and overlays on top of it. This mission has seven shadow zones. At this moment, two of those zones correspond to world hexes that are currently loaded.



So those shadow zones, and only those, will be loaded into memory and run.

It's simple and effective but requires careful memory management.



Here's an example of a base zone in the world.

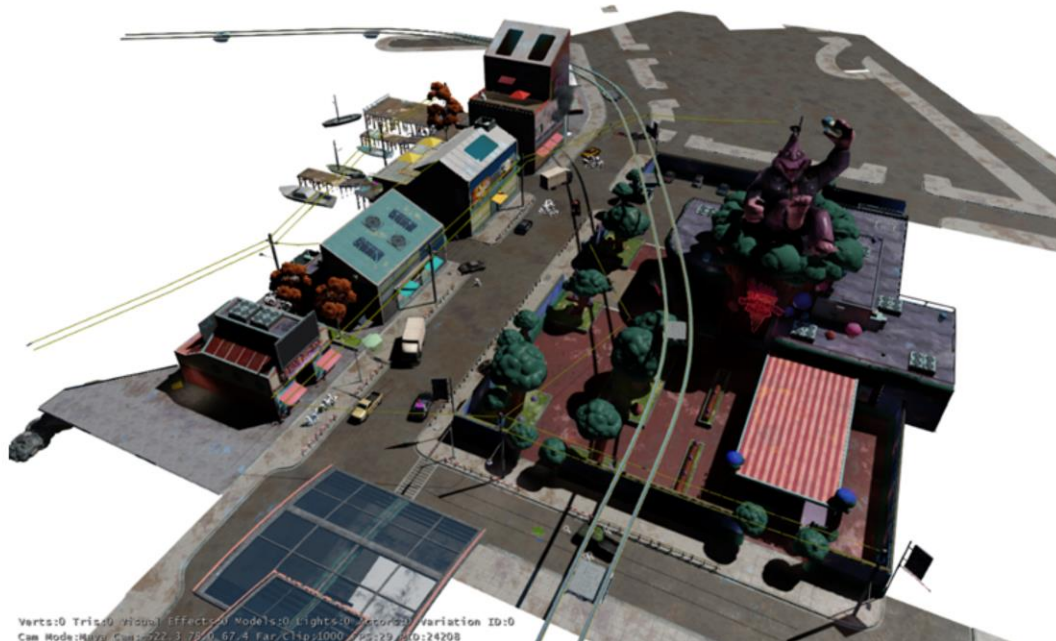


Here's a shadow zone specific to a mission. You can see it's got some enemies and spawners in it. These are the enemies for this mission, but only for this part of the world. The mission has a zone like this for every part of the world it crosses.



Here's what that hex looks like when the mission is active: the base zone runs normally, and then the shadow zone gets loaded in on top of it.

Hex_118 base



Shadow zones can contain basically anything – effect, decals, scripts...

With FNL_BOSS overlay



...or even buildings.



However...

But we also got ourselves into some trouble with these mission overlays.

We used overlays for...



missions



quests

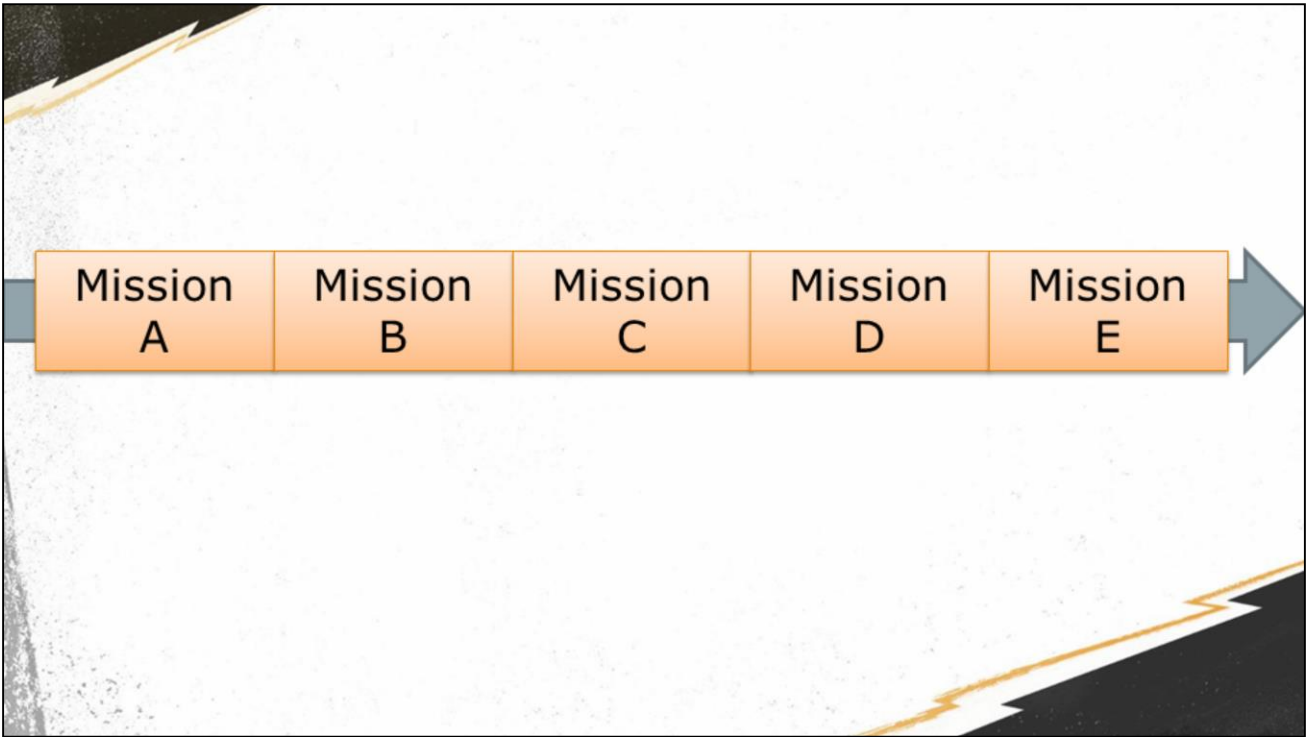


challenges

Missions – only one can be active at a time. These are the core plot

Quests – which are things like “collect all the ceramic kitties.” You can have any number of them simultaneously

Challenges – which are things like traversal ring-races.



Missions are linear; they are the core plot of the game. Only one can be active at a time, and at any time one is always active.

Quests, like “collect all the cats”



× 12

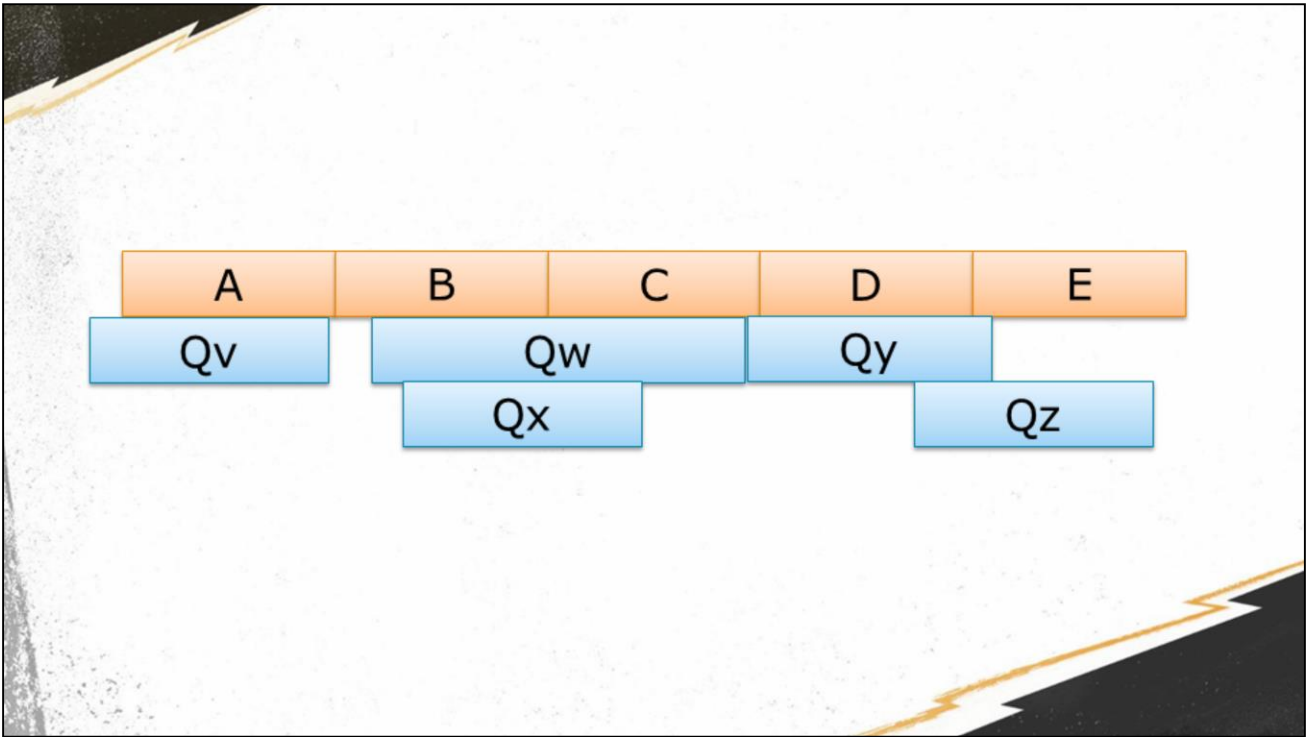


quests

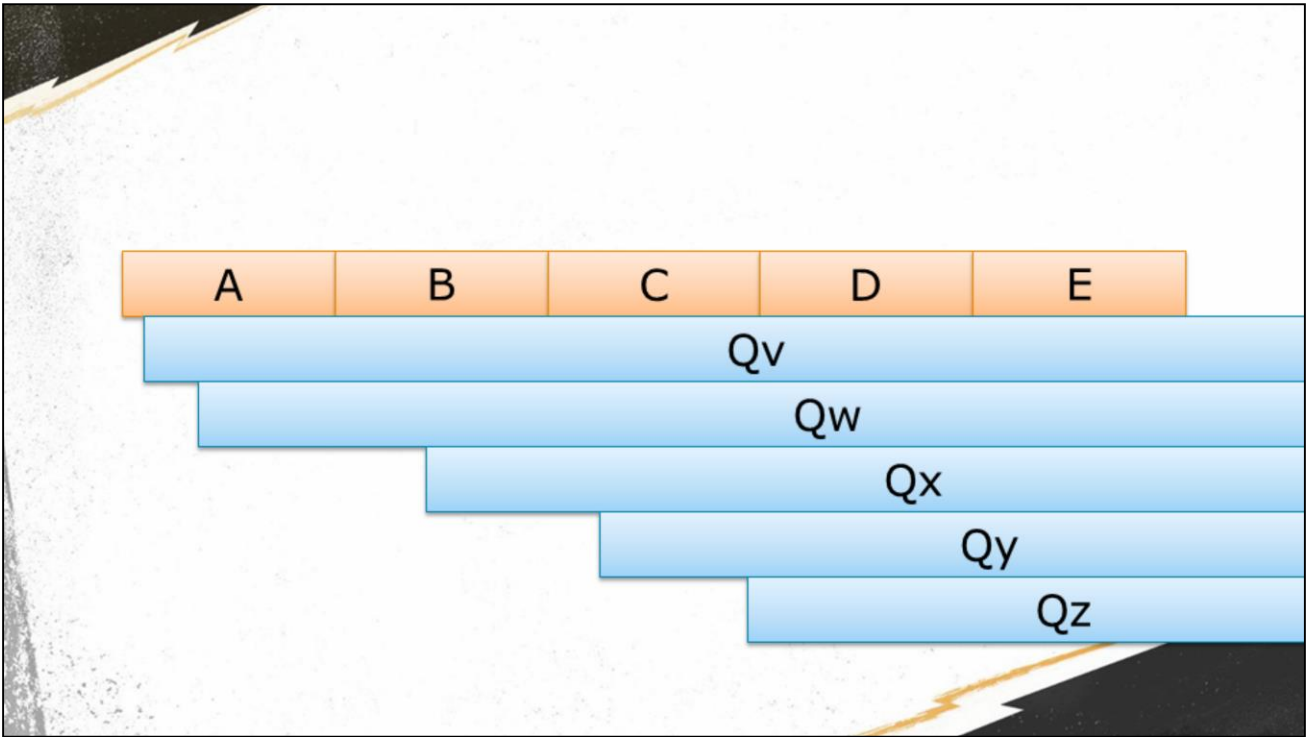
“Quests” are things like “find the twelve cat statues scattered around the city”. So, while the “find cat statues” quest is active, we’ve got to have its zone loaded into memory. That zone has cat-statue actors placed all over the city, and they stay in memory so long as the quest is active.



But you can have any number of quests active at a time. You could hypothetically have all of them active by the end of the game, if you start all the quests but never finish any of them until you've dealt with all the missions.

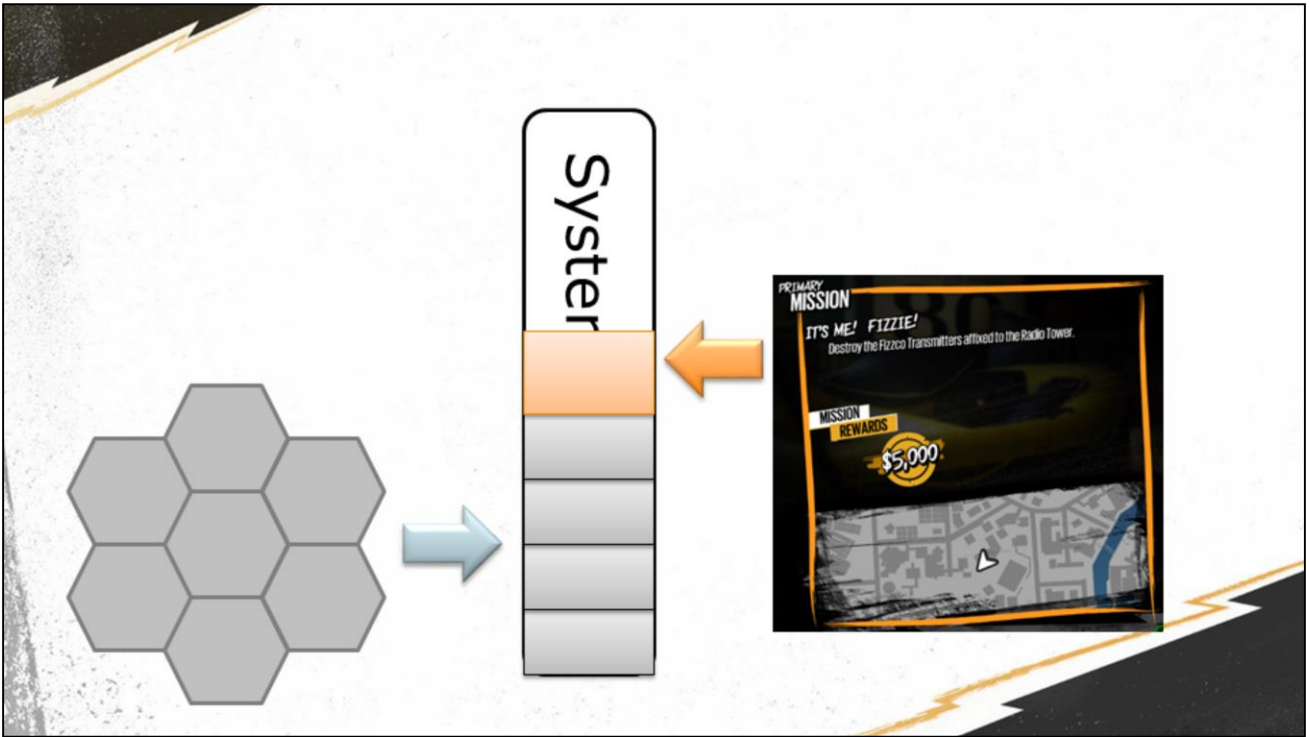


Some players might finish the quests as they come up.

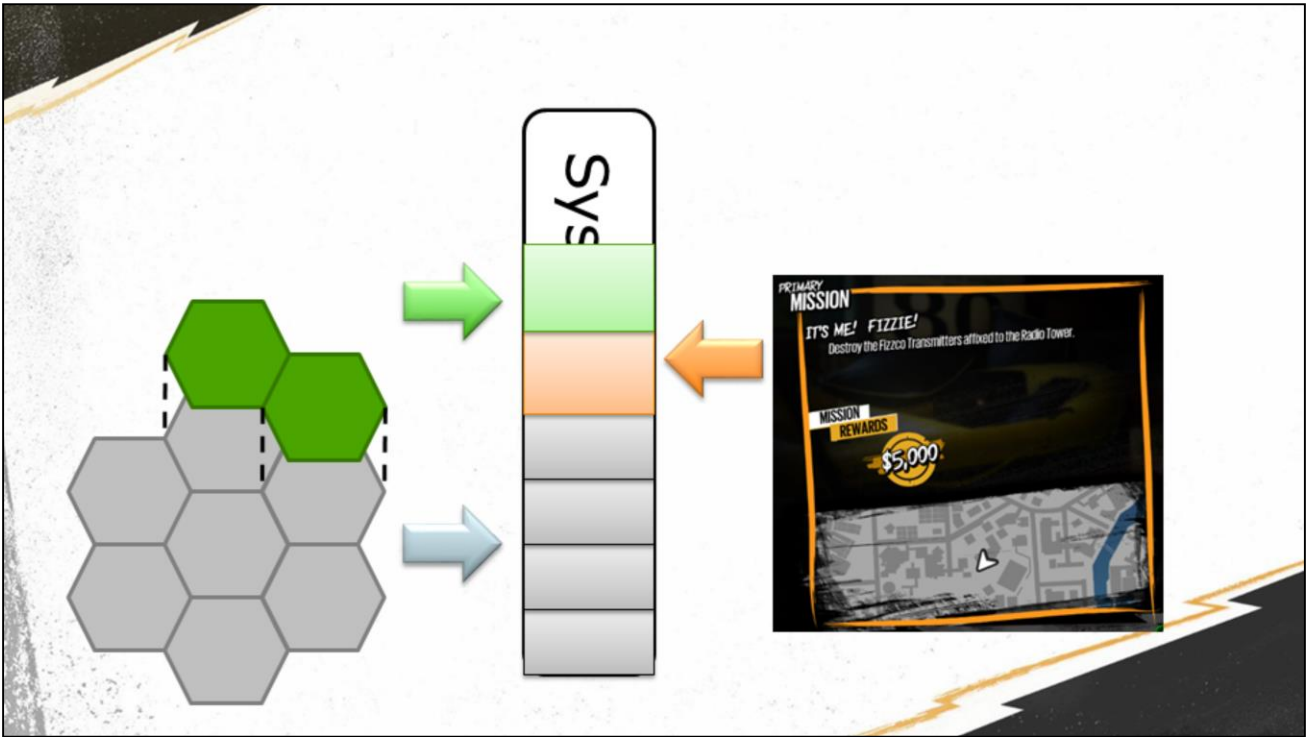


Others may leave all the side quests open until the end of the game.

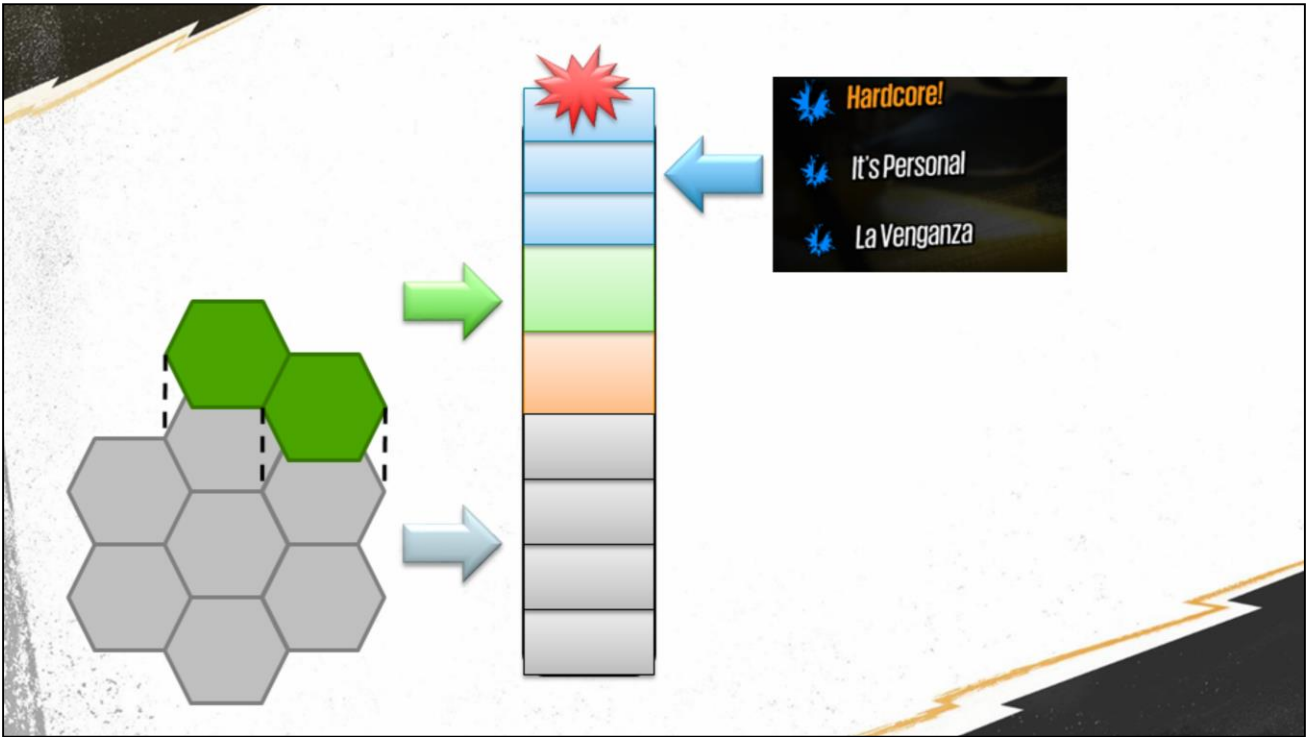
Different players will have different memory footprints depending on which quests they have active and whether they ever complete them or hang onto them.



This made memory budgeting unpredictable. We can know ahead of time how much memory a world hex will take up; that's fairly deterministic. And we know how much any given mission will take up, even though different mission have different footprints. But we can give them a budget too.



But those missions may have shadow zones, and the shadow zones may be of different sizes to each other. Some missions have more content in the global, and some have more in the shadow zone. Some shadow zones are bigger than others. Now you have several sources of variability in memory.



Because there's no limit on the number of possible open quests, hypothetically you could get to the end of the game while having *all* of the quests still open and active. Which, ultimately, meant that we had to tune the entire game with enough slack memory to accommodate all the quests being active in every possible scenario.

We found ourselves having to always budget for the worst case – the biggest zones loaded with the heaviest mission loaded and with all of the quests active.

And even so we had some spooky action at a distance. We might have a combination of mission, world zones, and shadow zones that are all running right at the limit of the memory budget. And everything is fine, but then a designer adds one more cat statue to one of the quests, and then bang we're over memory and the game crashes – but only if you're a player who hung onto side quests without finishing them and had a very specific combination of quests open by the time you got to this mission. It made QA very difficult.



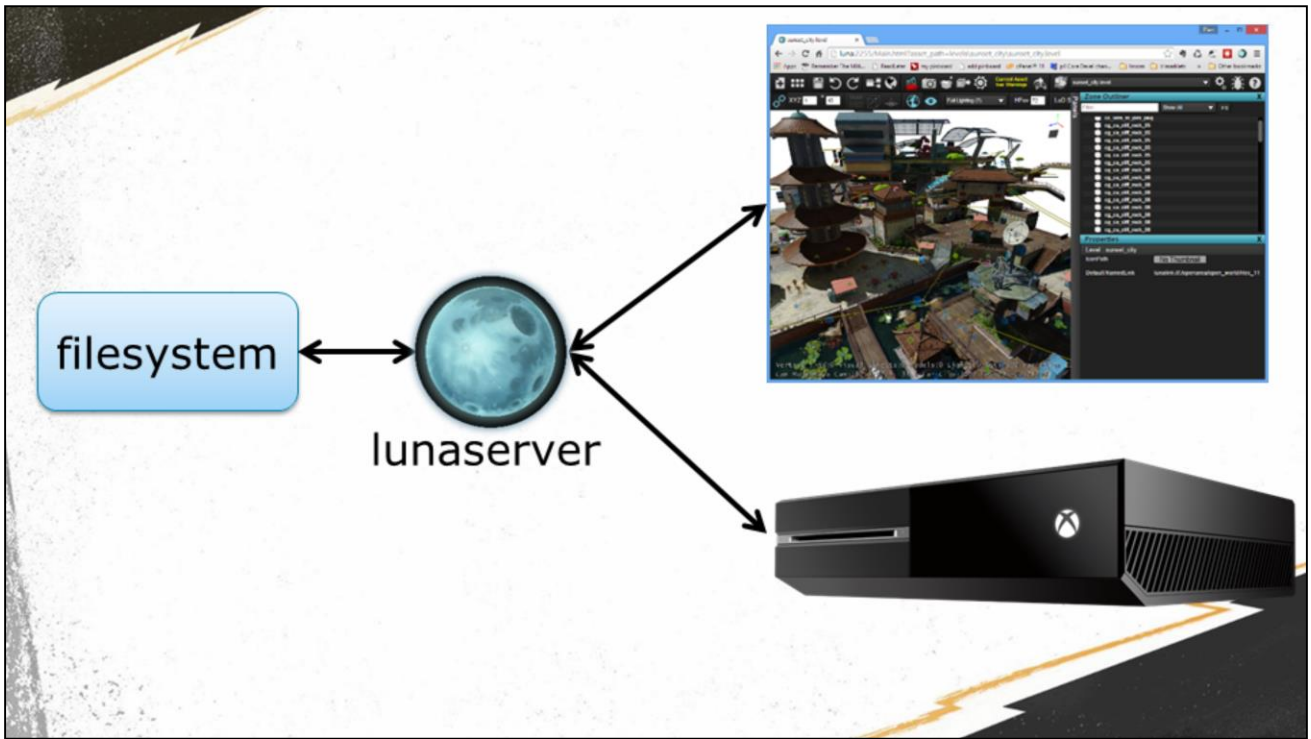
Filesystem

Quick review – so far we’ve got the hexes, zones, and asset hierarchy. Now let’s talk about the filesystem and the code that loads it.

Filesystems

- Loose files (aka “Lunaserver”)
 - Can hot-reload any asset
 - Used by developers
- Packed archives
 - Used by QA and customers

The Sunset engine has two (relevant) filesystems: one that can hot-reload any asset and is used for rapid development; and then the packed archives that we put into packaged QA builds and ultimately the customer’s disk.



The development filesystem is closely tied to our editor toolsuite, the Lunaserver.

The Lunaserver process runs on a developer's PC ("developer" here meaning everyone working on the game) and is responsible for tracking the state of all asset files, built targets, and all data interchange between tools.

Tools communicate with the Lunaserver via HTTP.

Our level editor is essentially Google Chrome. Many of our tools are either Chrome plugins, or else are written in Javascript and HTML.

The Lunaserver is running as an HTTP server on port 2255.

So our tools are basically a suite of web-apps that you access by connecting to localhost:2255.

Lunaserver tracks all of the source asset files you have on your PC (which is usually the whole game).

It monitors the filesystem journal. Whenever you touch (or sync) an asset source file (say, a model), it immediately builds the corresponding target.

Digression:- we cache all the built targets on a centralized Cached Content Server (CCS).

So if someone else has already built the version of the asset that you just synched, your Lunaserver will fetch from the cache rather than rebuild on your PC.

This speeds things up *a lot*. It was the only way we kept build times manageable enough for artists to work on a game with millions of assets.

If someone makes a code change to an asset format that requires, say, every model in the world to be rebuilt, then the Lunaserver on their PC automatically rebuilds all those assets and uploads the results to the cache.

If you wait for this process to finish before checking in your tools change, then anyone who subsequently gets the new tool version doesn't have to rebuild all the models; they download your built target from the cache.

When you run the game in "development mode", it also communicates with the Lunaserver via HTTP. Basically, it uses your Lunaserver as a network file system.

The console devkit can connect to your PC's lunaserver.

This means you can make a change to an asset and see it in the game without having to rebuild any archive files.

In fact, it means you can make a change to an asset and see it in the game immediately without having to even restart or reload! The Lunaserver pushes a notification to the game telling it to reload the asset. The game loose-loads assets from the development file system ad hoc, whenever something references them.

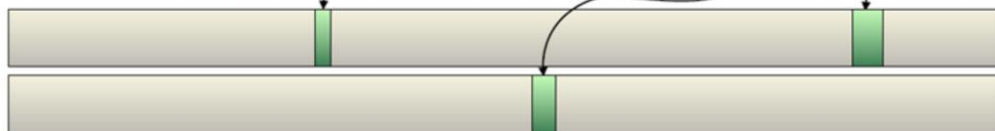
File format built on earlier tech, see: "Loading Based on Imperfect Data" Andreas Fredriksson, GDC 2013

Table of Contents File (Memory Mapped) ~800 kb

Sorted by Hash

Filename Hash	Compressed Size	Decompressed Size	Location
0001a234	1768	2896	0af82b60
0001a712	276193	358900	05fab88a
0001b682	57891	57891	1a58934c
...

Data Files (2 GB)



Our packaged, retail filesystem is based on the one we had in Fuse. You can get the details from Andreas' previous talk at GDC2013. But we made some revisions to make Sunset work.

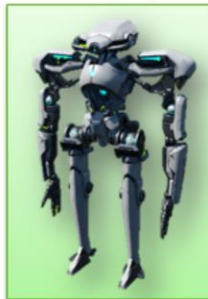


We have a set of large files (segments) that amalgamate all assets. We cap them at 3 GB arbitrarily – important to stay under 32 bits for compactness. Everything is compressed on the drive using LZ4.

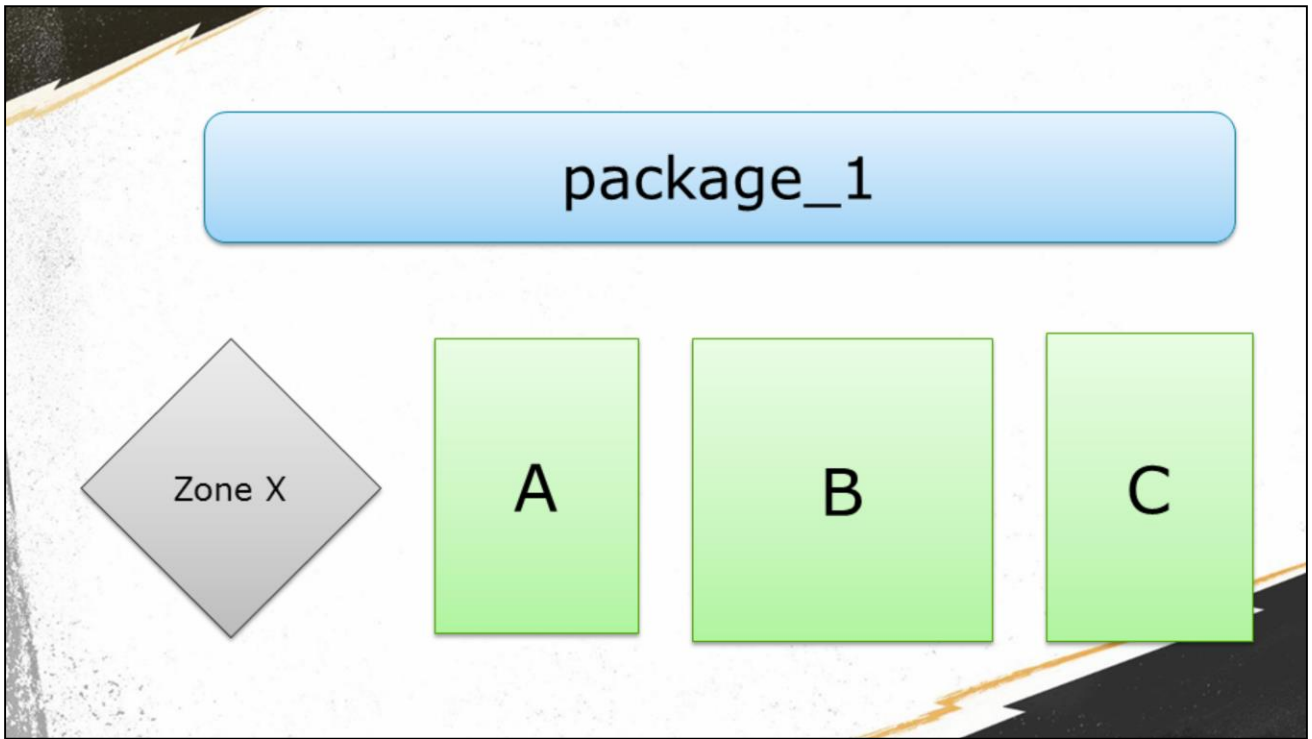
The packages are partitioned arbitrarily, so really we can just think about them as if they were concatenated into a single big package file (the difference is just that the indexes are two numbers, package id# and then offset inside the package).

package_1

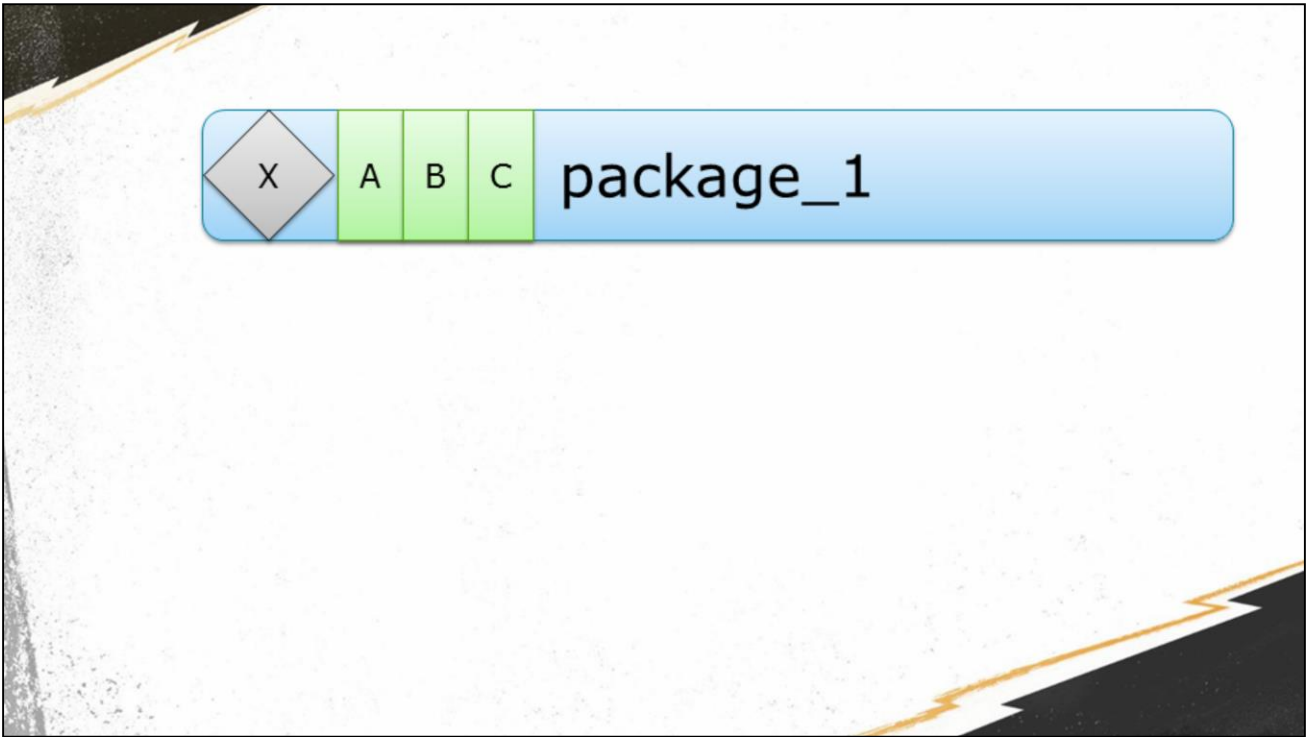
Zone X



There's two overarching categories of assets: "key assets", which in SO's case are zone files, and then everything else – decals, models, etc.

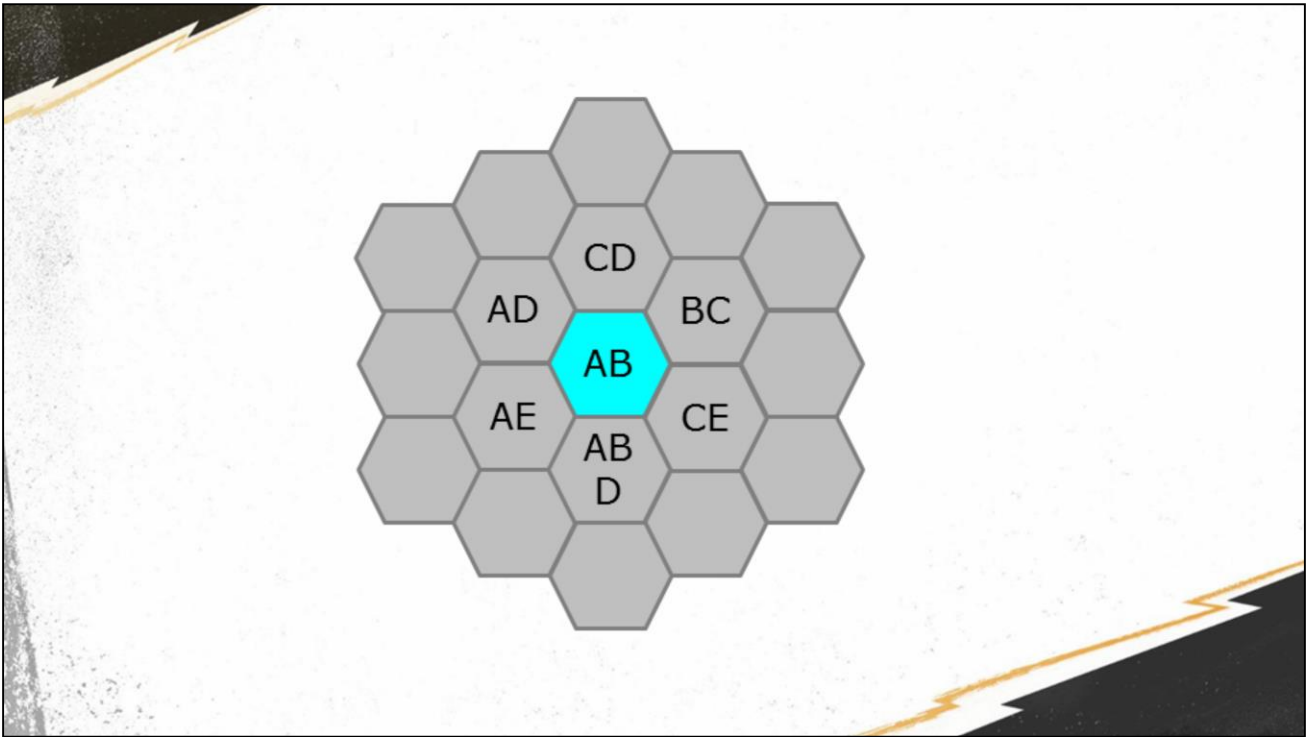


Let's just refer to them by letter to make things more legible.



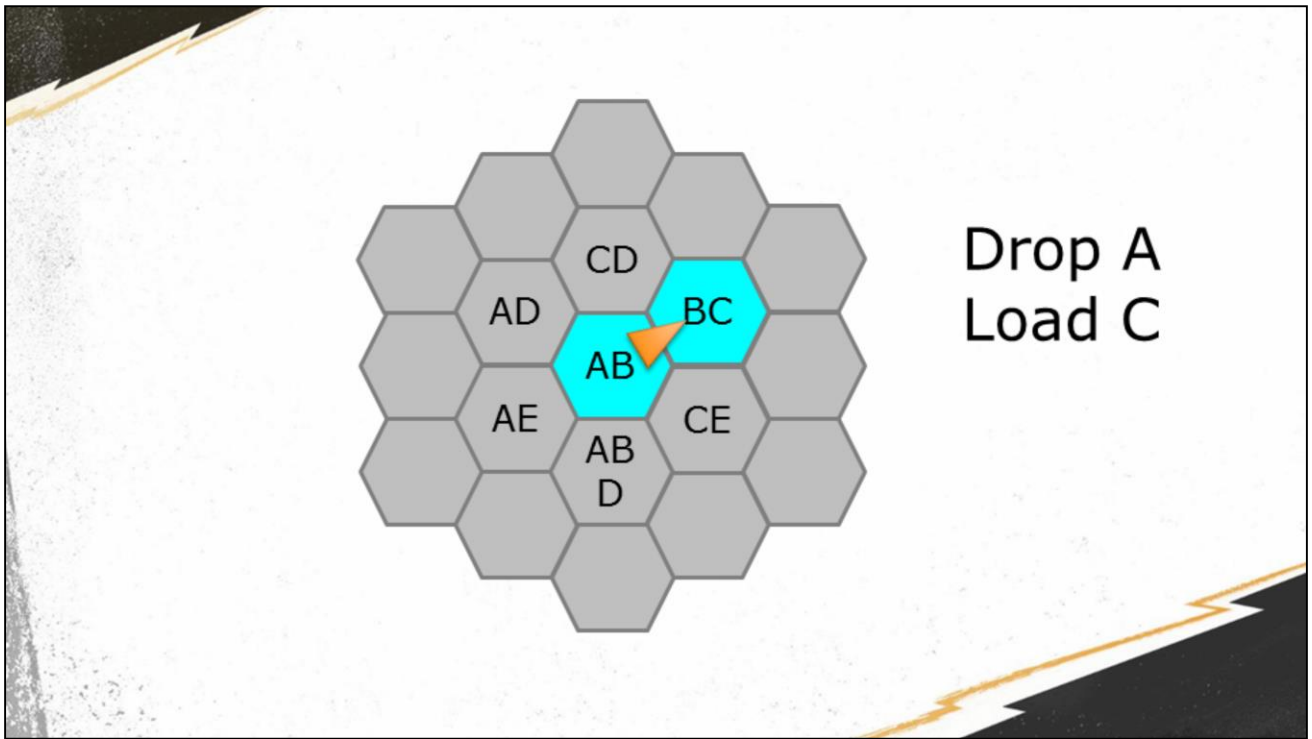
Okay, we could say “zones go into packages. For each zone, every asset it uses goes there too.”

So package files look kind of like this.

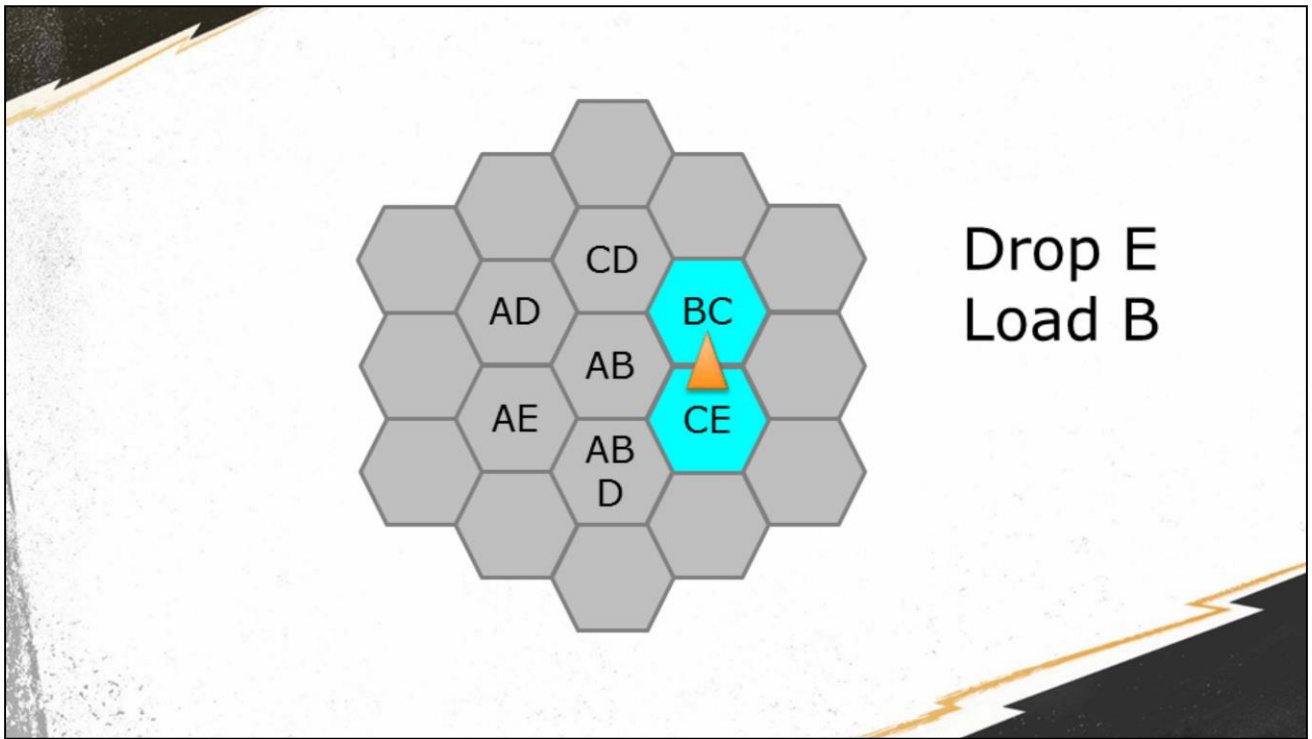


But hey, things get complicated, because you don't always need to load every asset in a hex when you step into that hex. Most of those assets are going to be shared with neighboring hexes. Many assets are found in multiple hexes – for example, the big blue umbrella is in a lot of locations throughout the game. If four neighboring hexes all use the same umbrella, you only need to load it once.

That means that typically you're only going to have to load a few assets on entering a hex – the ones it didn't share with where you came from. Imagine part of the world here where all the hexes have some subset of assets ABCDE.



When you move from the center hex to the top right hex, you need to load asset C. But this gets complicated because it depends on where you came from and which direction you're travelling.



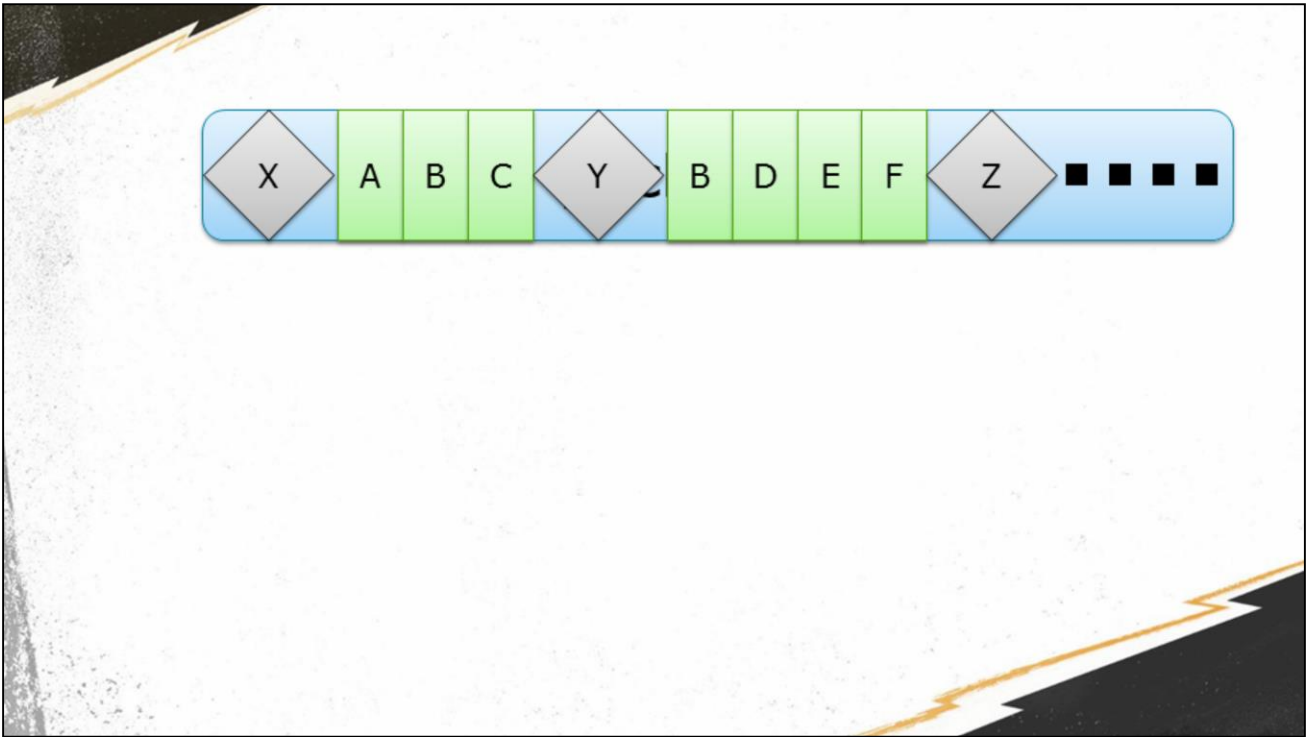
If you were coming from this other direction, you'd need to load a different asset – you'd be loading B, not C.

You can see that there's an interesting combinatorial problem here – each hex is a set of assets, and every one of those sets is a point in the graph, and every transition between hexes is an edge on that graph and a symmetric difference of hexes. So, from each hex, we can work out what the set of assets we need to get to each neighboring hex is.



But that rapidly becomes really hard when you have *hundreds* of hexes!

Figuring out which assets a hex needs to contain in the context of its neighbors and all the directions in which the player may approach it is *such* an interesting combinatorial problem...

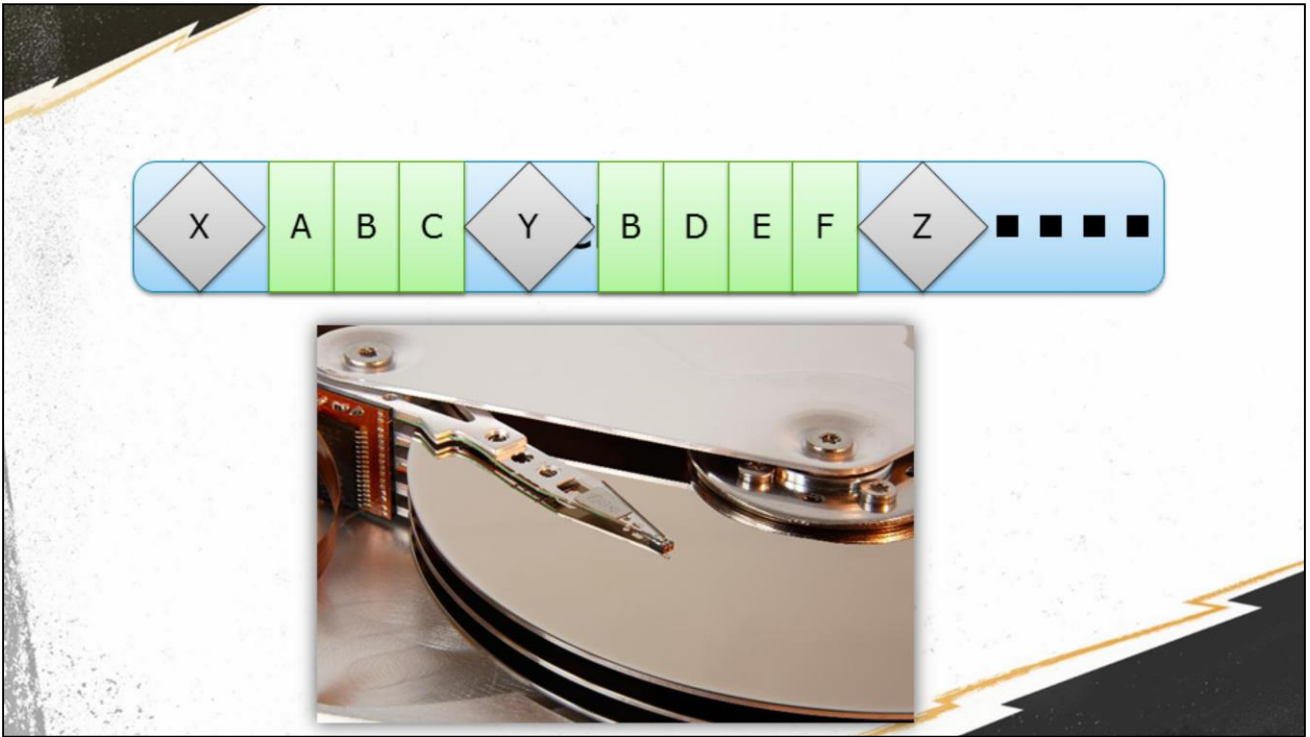


...that we didn't even try to solve it. Every hex file contains every asset that hex references. This is much, much simpler to deal with; and the Blu-ray format has plenty of space for the redundancy. And it made loading *faster*.

Key to efficient I/O:

- Consistent read size
- No seeks
 - Often better to read data and discard it than to try to seek past it
- LZ4 compression

See, the key to good I/O performance is basically NO SEEKS EVER. You want each read to consist of a very few contiguous blocks of data, which you load in their entirety.

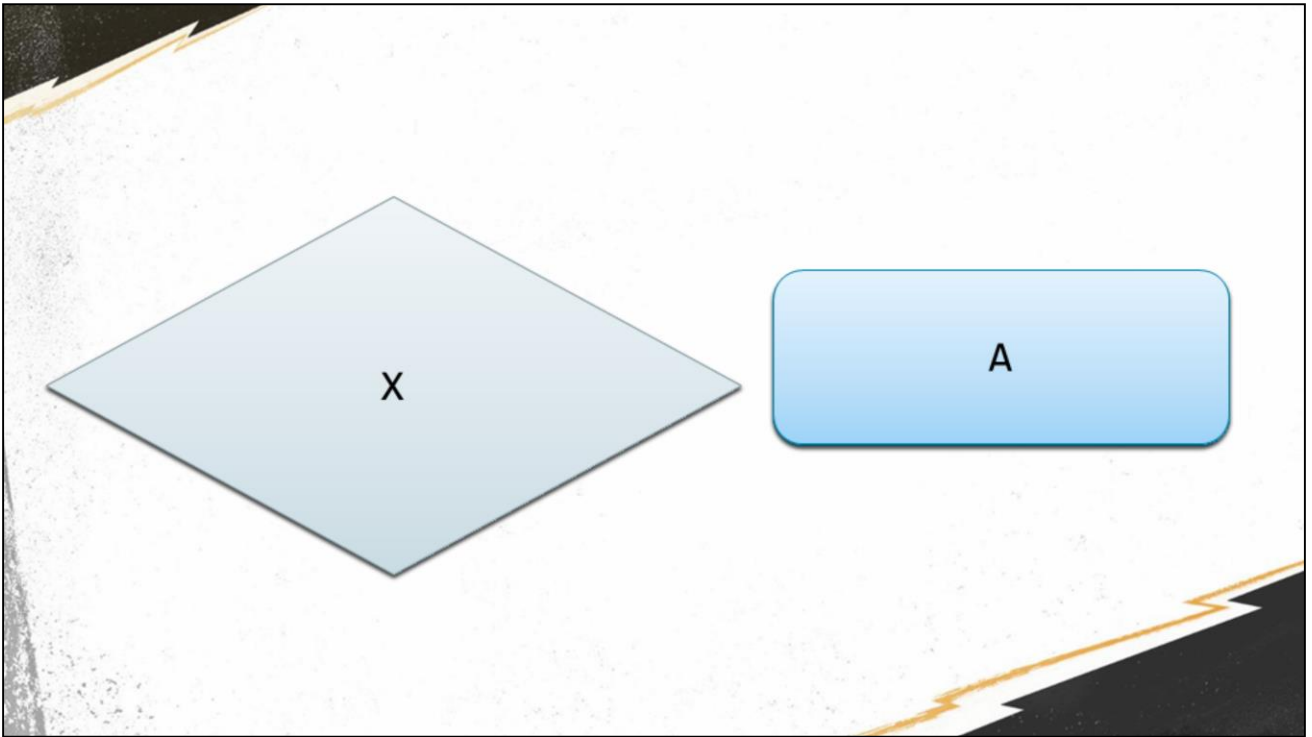


You ought to think of your read pointer as if it has mass and momentum. You want to keep it moving at a constant velocity. Making it skip ahead, or backwards, requires changing that velocity, which is costly.

And the reason for this is that it *is* a mechanical object with mass and momentum. Seeks mean a motor has to push a metal thing around. So it's sometimes better to load a little too much and then throw it out, than to try to seek over it.

We tried to keep all assets that had to be loaded together contiguous on disk. That is to say, we wanted all of a zone's assets to be adjacent to it in the package.

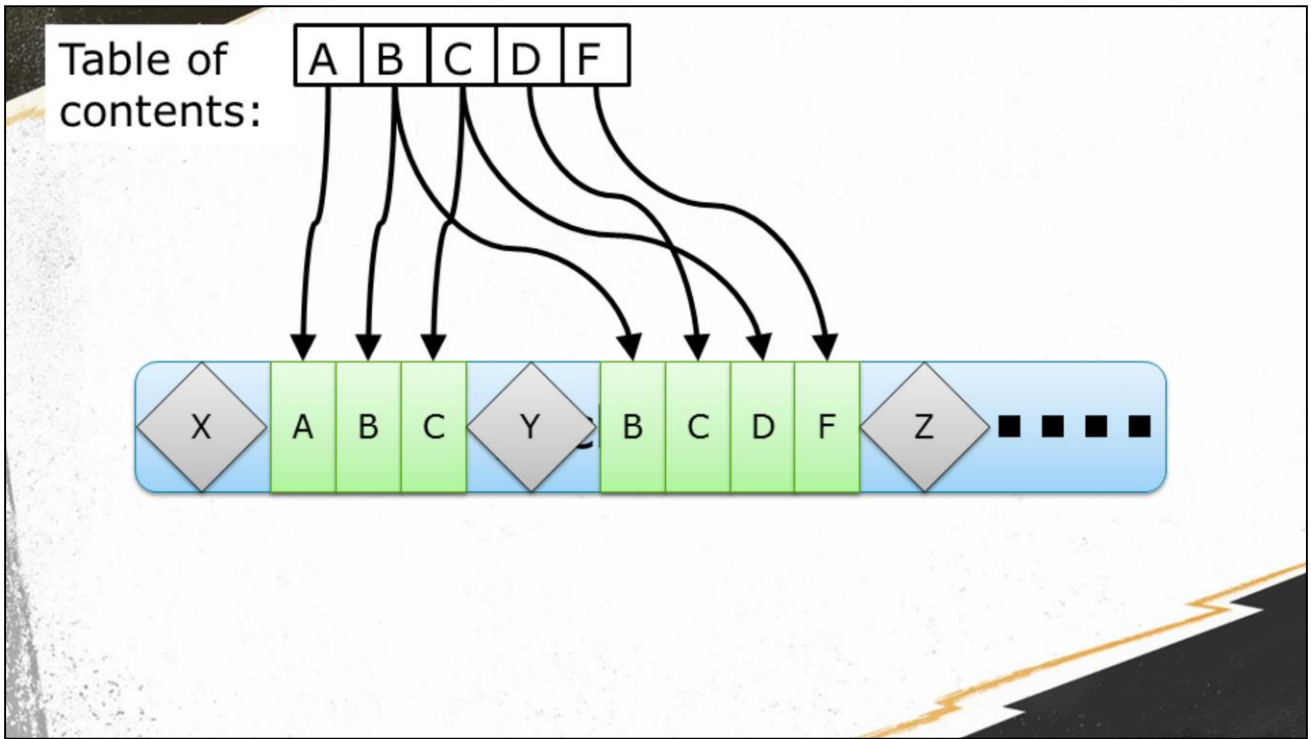
Photo credit: Wikimedia Commons user Eric Gaba - https://commons.wikimedia.org/wiki/File:Seagate_ST33232A_hard_disk_head_and_platters_detail.jpg



So, how to sort the assets on disk so that the ones which load together are next to each other?

The easiest way to index an asset is to start by hashing its filename, turn it into a 64 bit number.

And then... there is no then. That's it.



A hash is as good an index as any! It makes building the table of contents really easy, it's just a hash table.

And, having the asset duplicated a few times on disk can lead to *faster* load times, in the particular case of assets that get loaded outside of the usual zone mechanism. If those assets are in many places on disk, then the batch loader thread has more options for building the most efficient load order.

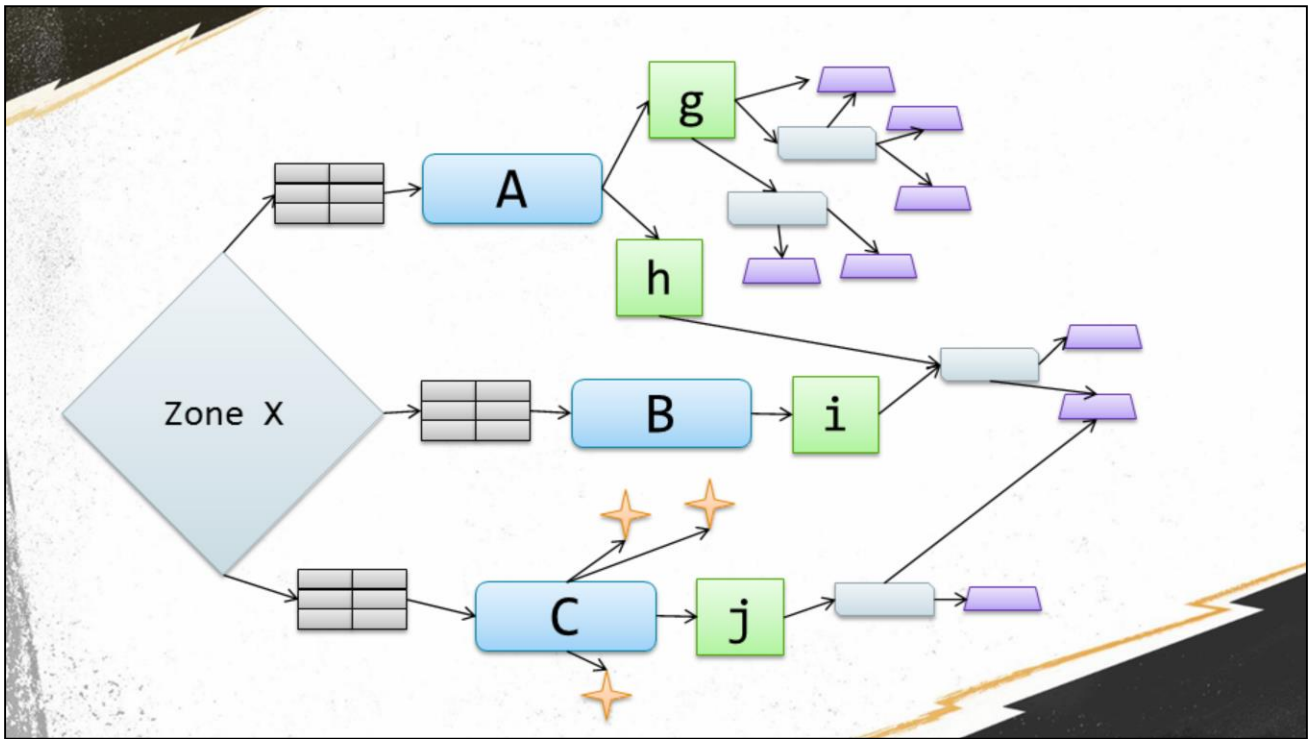
--

Duplication works with the "location selection" at runtime by just putting additional copies nearby. (This is good, because we very frequently ***don't*** want a bunch of the duplicated assets, because they're in memory already – but the # of permutations means that we only know what is needed at runtime.)

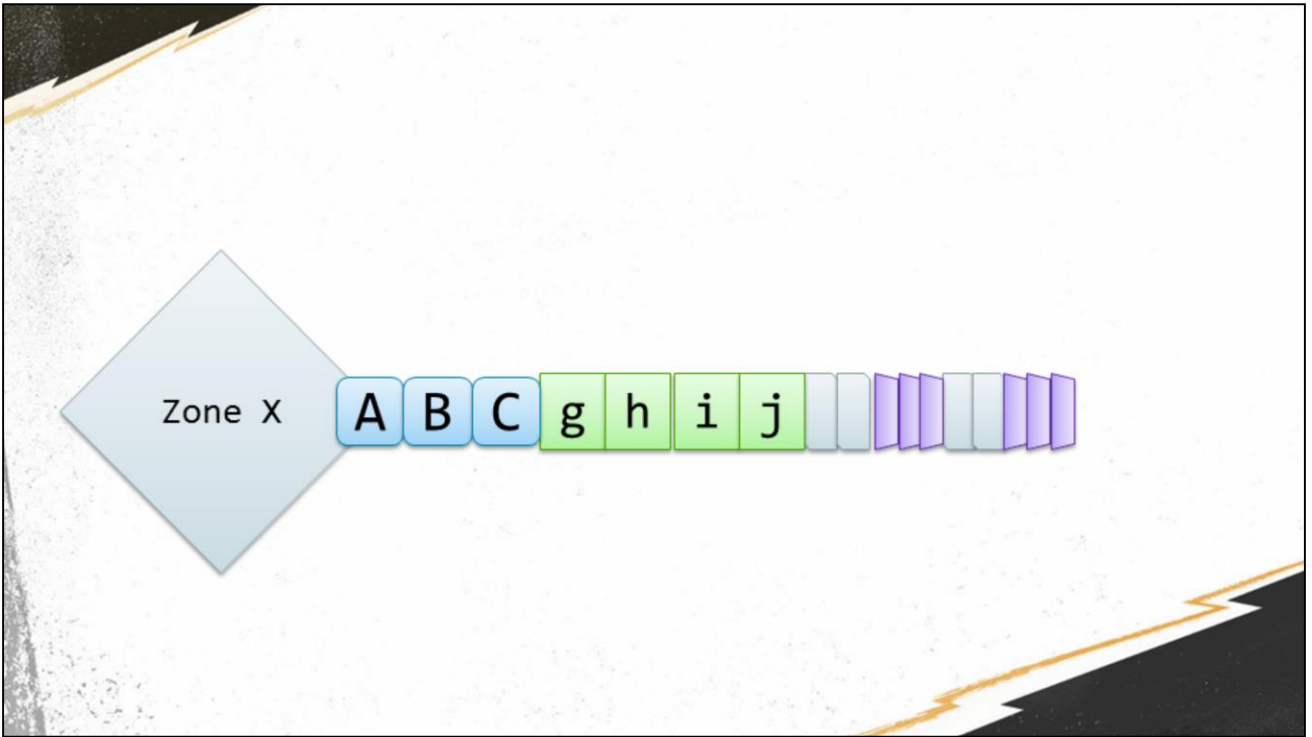
That means that when any key asset – a zone – is loaded, the loader knows to look for all other assets as close as possible to them.

This works reasonably even if numerous key assets are loaded together in a batch.

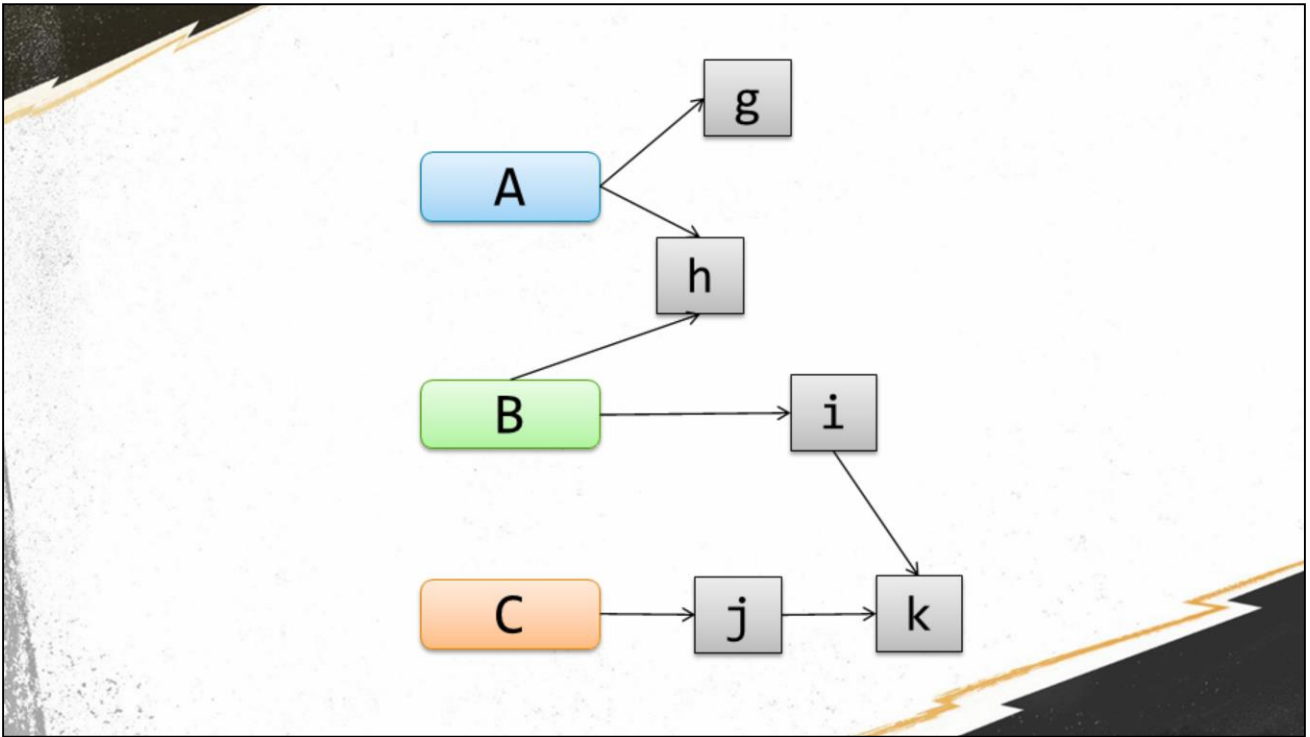
The duplication is also helpful if you find yourself having to load an asset after the fact - sometimes mission zones are loaded ad hoc, and they need to bring in a bunch of stuff. We have a background loader thread that batches up all the load requests that come into it. Having some assets be duplicated on disk lets the background job sort the requests and find the asset copy closest to the current read position. Those additional options can mean less seeks at runtime.



We can easily work out all the assets a zone needs by doing a breadth-first on its DAG of assets.

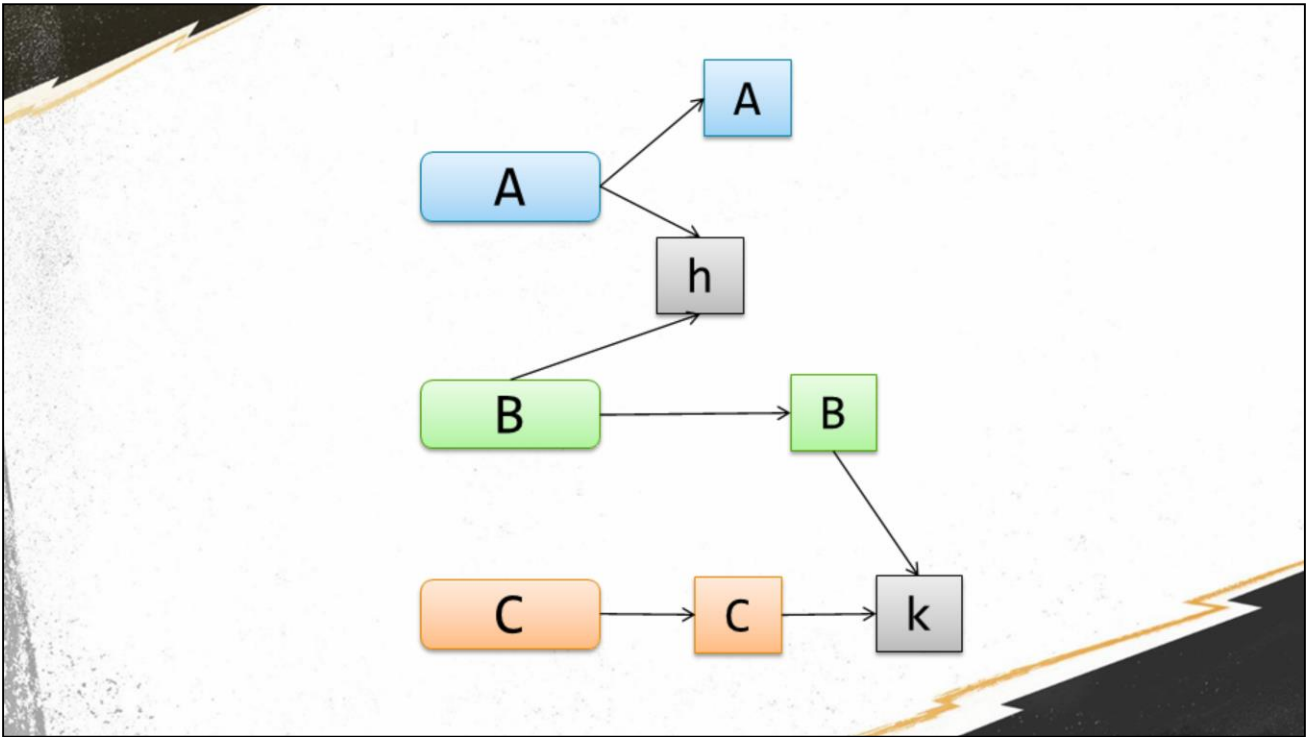


So those just go next to the key asset on disk.



There is one additional cute optimization that helps us manage assets that are only ever referenced by one other asset. If an asset has only one parent, we copy that parent's key down to be the child asset's key too – that ensures that they will be adjacent to each other on disk and basically considered the same asset.

This doesn't make much difference for things that are inside streamed zones anyway, but it does help with things that are mentioned from "global" contexts, like vanity items and mission content, which aren't bound to a geographical zone that the archiver can call a "key asset."

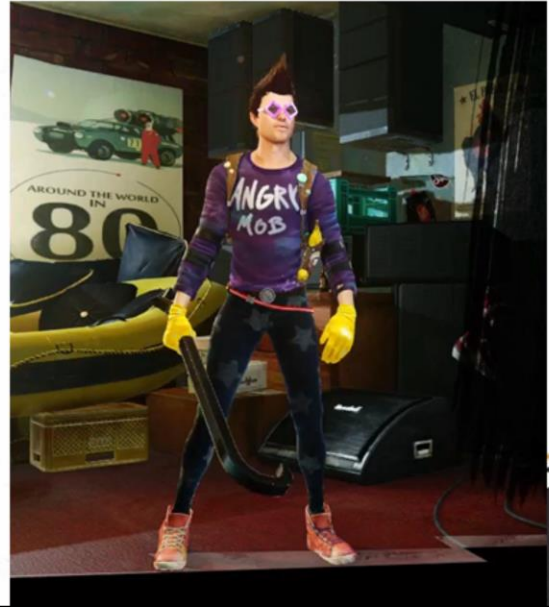


There is one additional cute optimization that helps you manage assets that are only ever referenced by one other asset. If an asset has only one parent, we copy that parent's key down to be the child asset's key too – that ensures that they will be adjacent to each other on disk and basically considered the same asset.

This doesn't make much difference for things that are inside streamed zones anyway, but it does help with things that are mentioned from "global" contexts, like vanity items and mission content, which aren't bound to a geographical zone that the archiver can call a "key asset."

Loose-loaded assets

- Highest LOD texture mips
- Long sound files (music and dialog)
- Vanity items



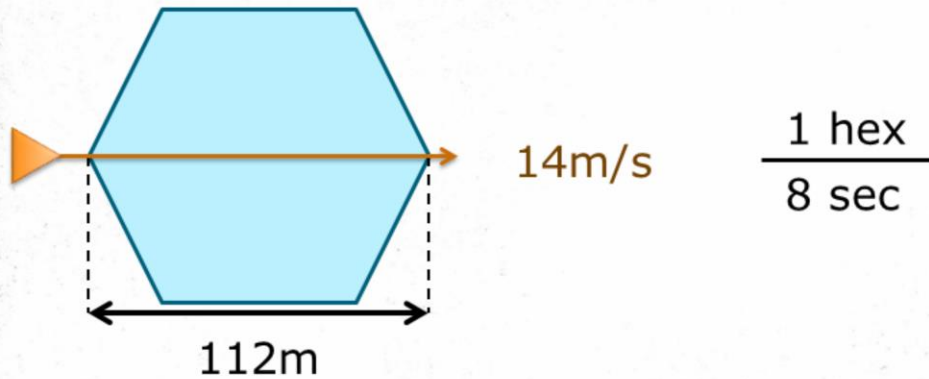
We also stream highest-res texture MIPs and long-playing sound files. This doesn't go through the general zone-loading system, it's just on demand loading of assets as they are referenced.

Vanity items (the customizable clothing and so forth) are loose-loaded, because there is no way to anticipate at build time what combinations the player will need.

Even some of our NPCs have randomized outfits.

All of this dispatches through a single "asset manager" thread that handles the disk I/O and prioritization.

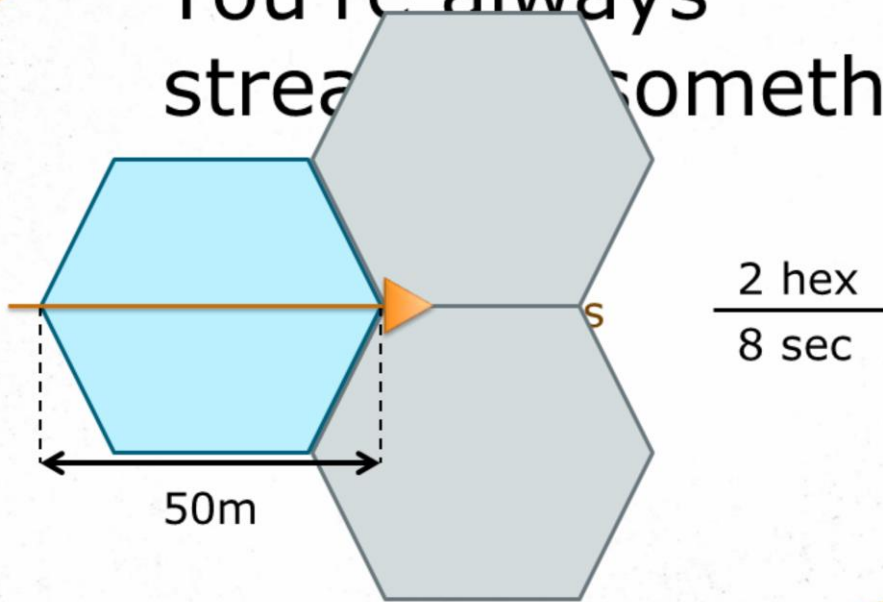
You're always streaming something



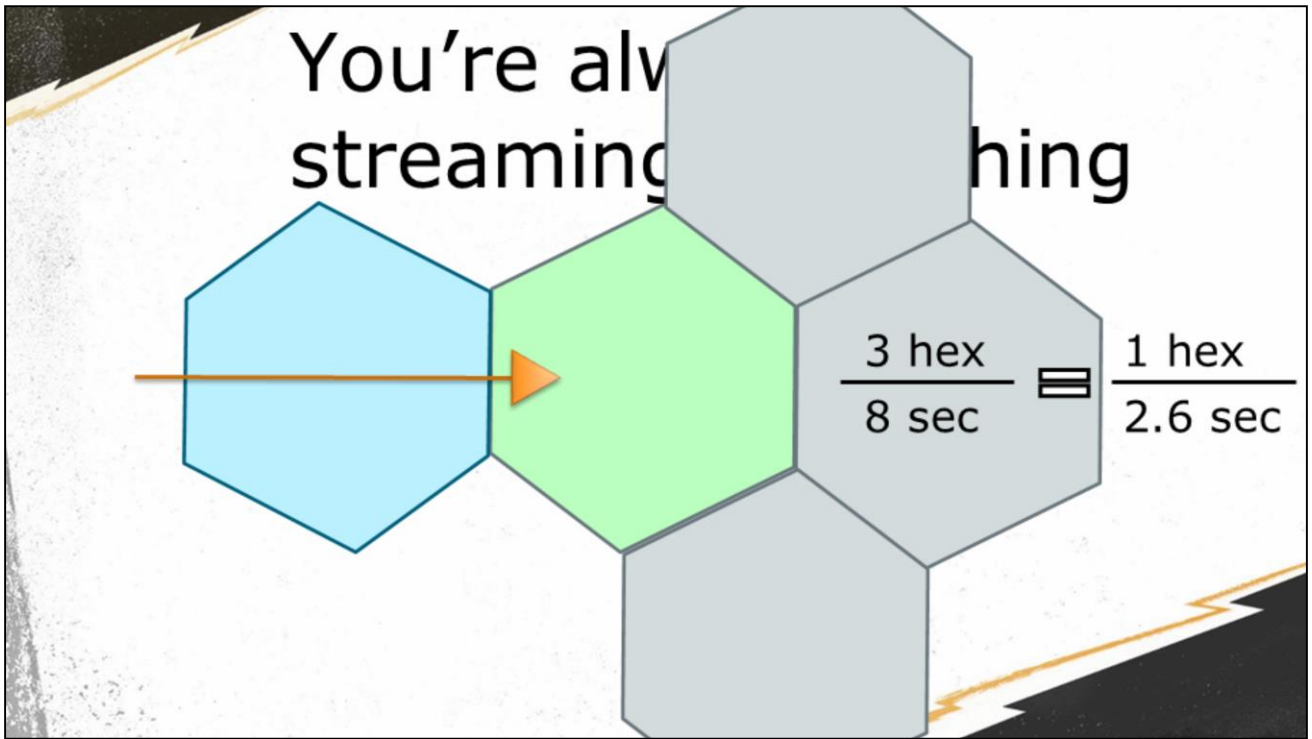
A key difference between the way you design an engine for a linear, airlocked campaign, and one that can handle huge open worlds, is that loading a region is something that happens intermittently in a linear game, and *continually* in an open world.

Most of our hexes are about 110 meters in diameter, and we budgeted things for a top player speed of 14 meters per second (although sometimes it can be a little higher than that). So, if the player is running forward, that means that you need to load a hex every eight seconds, right?

You're always
streaming something



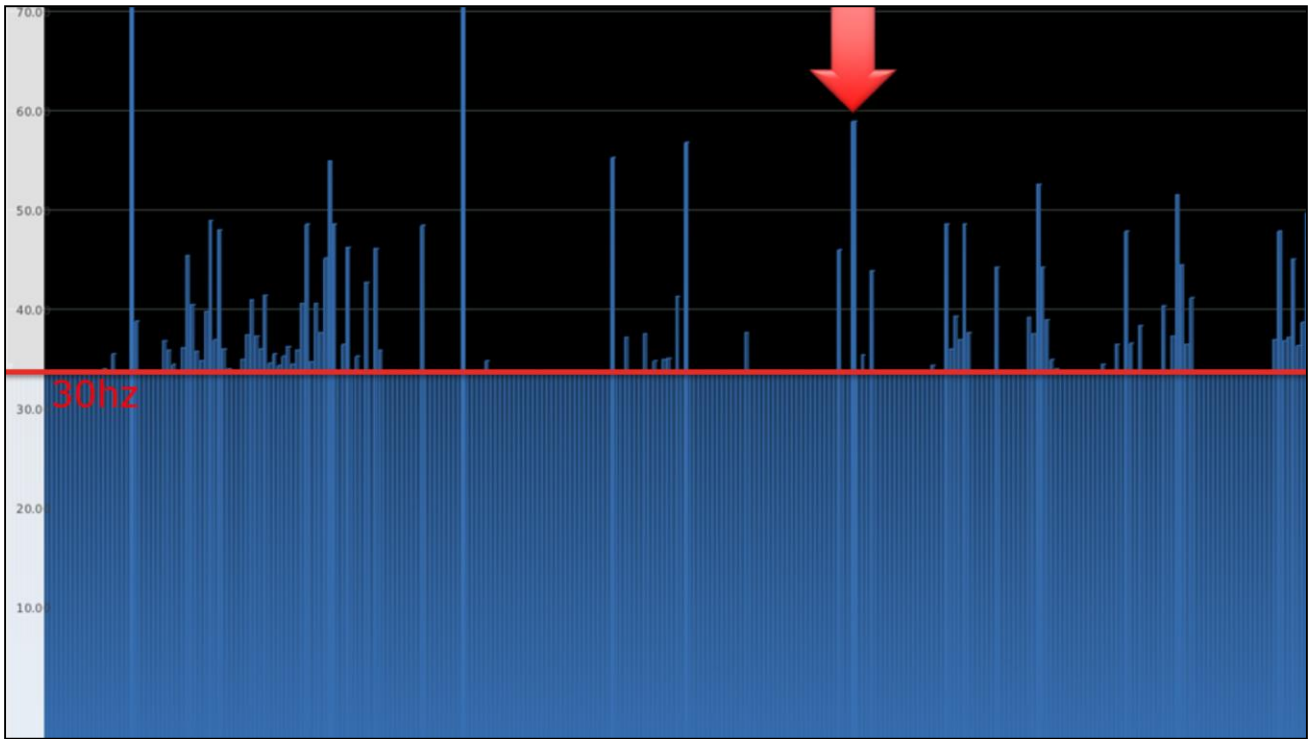
Well, no, you're loading two hexes every eight seconds.



Really it's three. So you've got to load and instantiate an entire hex in two and a half seconds... continually.

This gave us a nice figure to target for our hex size. We said, "well, we'll need to budget things conservatively, so let's say that all of a hex needs to stream in 1.6 seconds. We can multiply that by the hard drive's throughput and know how big each of our hexes is allowed to be."

But it turns out that disk I/O was not the limiting factor on streaming.

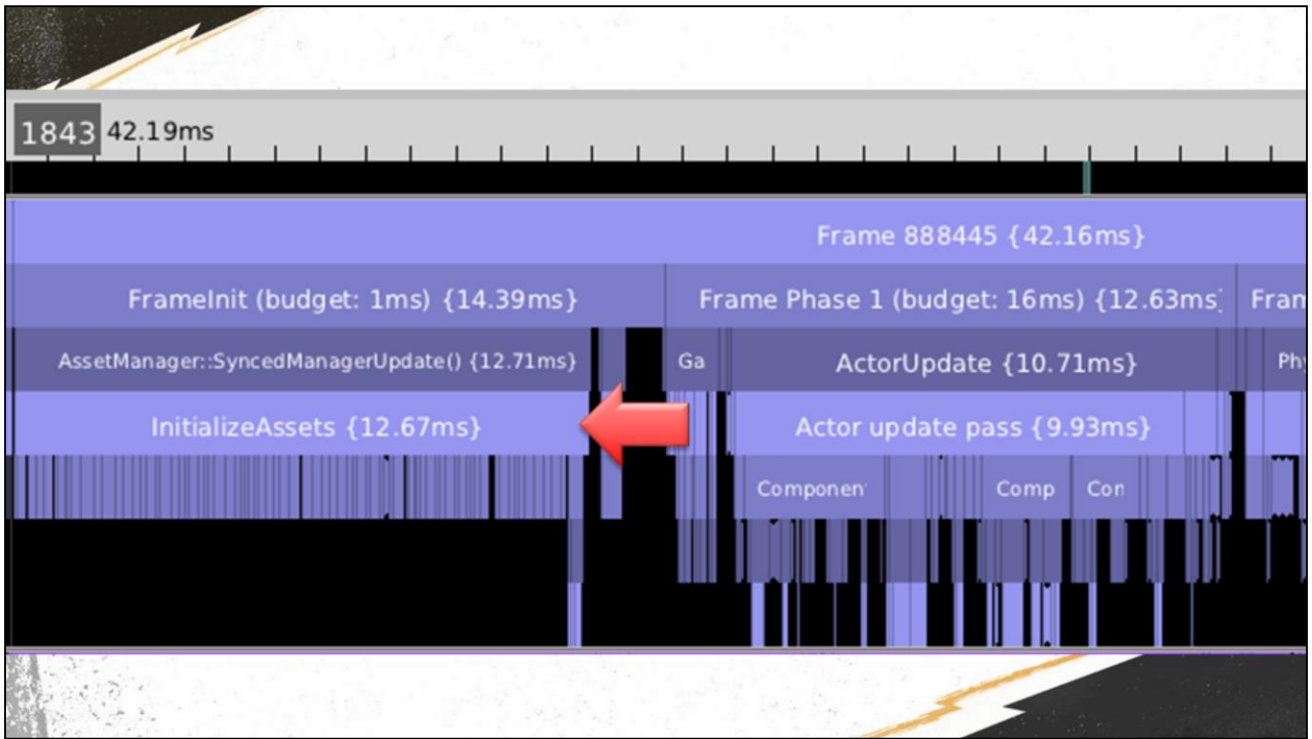


The limiting factor on how fast we could stream the world was runtime initialization, not file IO.

Our engine was designed for Resistance and Fuse – where each zone has a few hundred entities that need to initialize once loaded.

This quickly became a huge performance spike on every hex transition, which could pull in 30 zones with *thousands* of entities between them.

What you're looking at here is a performance capture from midway through development. The horizontal axis is time; each vertical bar is a single frame. The height of the bar is the frame's length in milliseconds. Every frame that pokes above the red line took longer than 33.3ms, meaning that it broke the budget to hit 30hz. Let's see what pushed it over.



That spike is the CPU side of setting up assets – not the disk I/O, but all the business of fixing up pointers and handles and initializing fields.

We had a lot of stuff in Fuse that *could* have been deterministic and done by the builder, but we did at runtime because it was easier and because “well, this only happens once on setup for this asset.” That’s okay when we load a region, like, once every few minutes, and we’ve got the player locked down while it happens. But now we’re doing all that work for all the assets in a zone, and we’re always doing it all the time – there is *always* something being loaded. So suddenly we had something that worked okay previously but fell down at scale.

$$\frac{3 \text{ hex}}{8 \text{ sec}} = 2666\text{ms/hex} = 80 \text{ frames/hex}$$



2000 entities

25 entities /
frame?

What about initializing actors? Lets say the typical zone has 2000 actors in it. You'd think, okay, well that means we have 2000 entities in a hex and 80 frames for a hex so that's 25 entities per frame. But it wasn't that simple for us. Our engine was designed with the assumption that all of the actors in a hex would initialize simultaneously.

The average time to stream a hex is about 2 seconds, so we were *on average* initializing 1000 entities per second or 33 per frame – but our engine was designed to initialize everything in a zone at once, when the zone loaded. Some things (like script) broke if actors were initialized before other things they depended on in the same zone.

Ultimately we found a way to time-slice actor initialization, so that we amortized that once-per-hex 180ms hitch into a 3ms-per-frame cost. But this was a painful retrofit, and you Think of your actors that do deterministic setup work at runtime...

Things that that you haven't gotten around to making an offline build step for, because constructing a new asset builder is complicated and you don't always have all the data that's conveniently available to an in-game entity,

and it seemed okay to have a runtime step that searches the world for all the places a pigeon can spawn because the pigeon only has to do it once, when it's instantiated.

Well, now you've got always got hundreds of pigeons being spawned and destroyed, so that "only does it once on load" code is biting you many times per second.

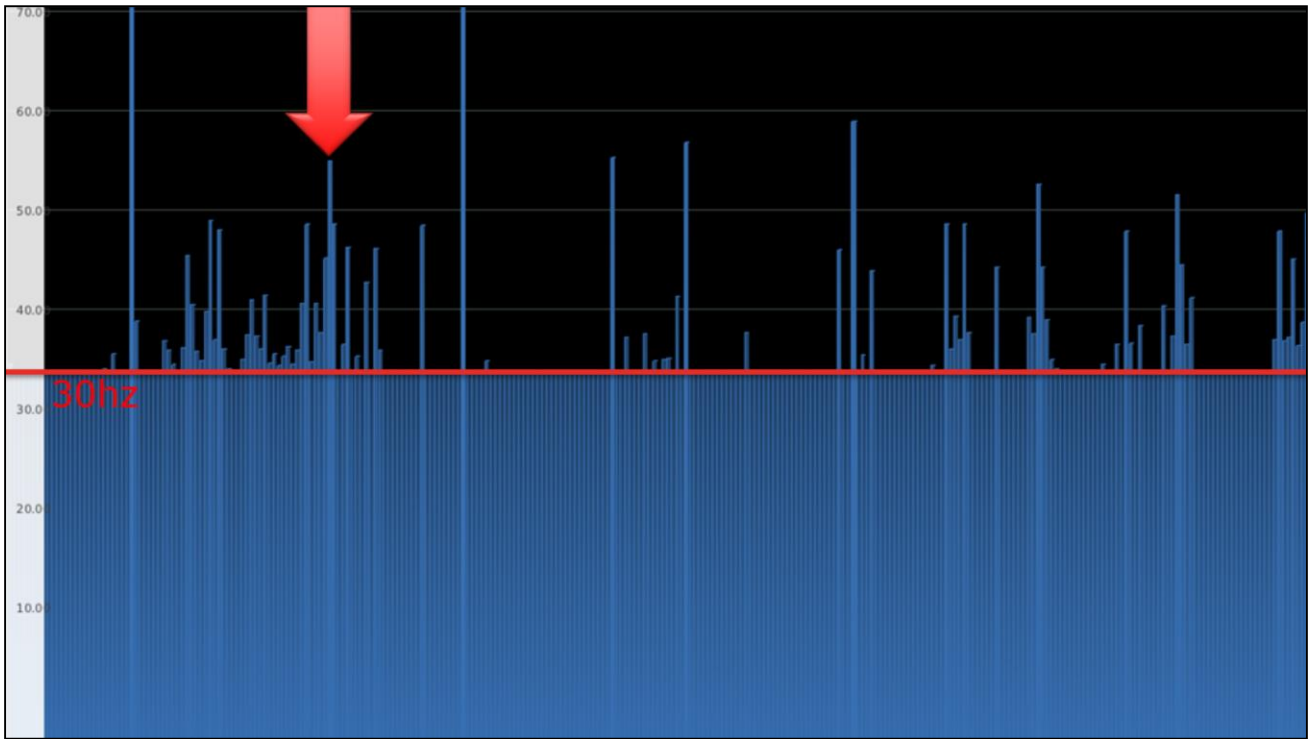
Timeslicing initialization

- Maybe init only 100 entities per frame?
- But what about...
 - Entities that refer to each other?
 - Scripts that talk about entities?
 - Things sitting on top of other things?

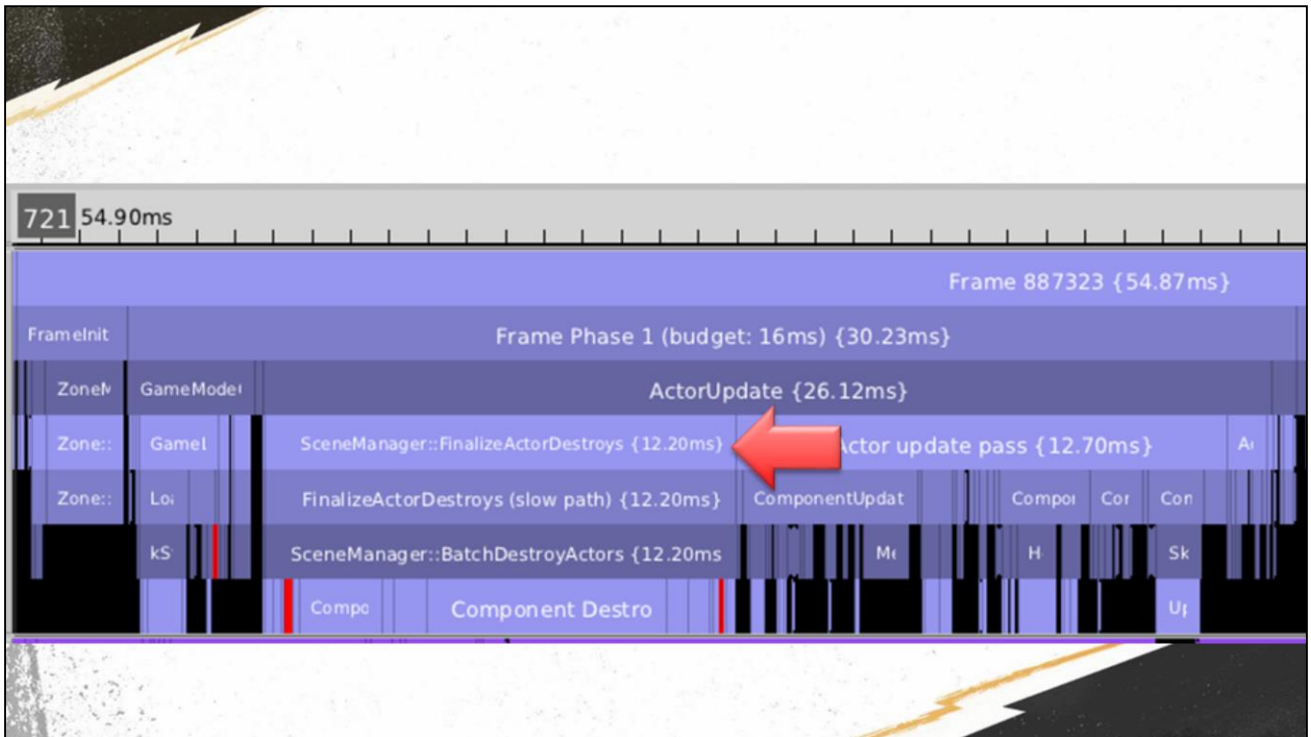
So you might have problems with entities that referred to each other; or scripts that had to set up actors in their first frame, and so things would break if the script came in before the actor did, or the actor came in before the script was ready for it.

Fortunately, not *every* object broke in this way. Order of initialization matters a lot for scripts and actors, but less so for static models and for visual effects. So we were able to time slice things with some effort, by lumping things into categories that couldn't possibly depend on one another and working one category at a time.

But this was a painful retrofit, and you need to be really careful with things that refer to each other.



But one thing still vexed us....



We never did manage to timeslice *destruction*. All the objects in a zone get purged in the same frame.



Space station bolt cleaning tool

(source: NASA)

Still, it worked well enough. The game ran smoothly and the designers felt free in what they were able to put into the world. We could have torn down and rebuilt our entire actor initialization design – forced designers to rebuild all of the actors in a totally different way, and established some kind of asynchronous messaging between actors. But before we tried that, we tried some incremental steps to make things work, and ultimately they proved to be good enough.



Our hexes are in two dimensions

You've noticed that all of our hexes are two dimensional. We stream the world as you move side to side, but not as you move up and down. Why? Well, because you can move sideways at 14m/s, but you move downwards at an acceleration of 10m/s/s. You can fall a lot faster than we can possibly stream in the world, so we realized that if the player fell from a precipice, we had to have already loaded in all the ground that the player could possibly fall into. It's just something we designed for.

Also, we just wanted to reduce complexity. We might revisit this in the future and just kill the player if he freefalls too far.

Image credit: Wikimedia Commons - Mariakeernik - <https://commons.wikimedia.org/wiki/File:Mesilask%C3%A4rg.jpg>



A continually-streaming game means that parts of the world are always coming into and out of existence. How do you handle things that connect across the seams; or that have actors in one area referring to actors in another area?

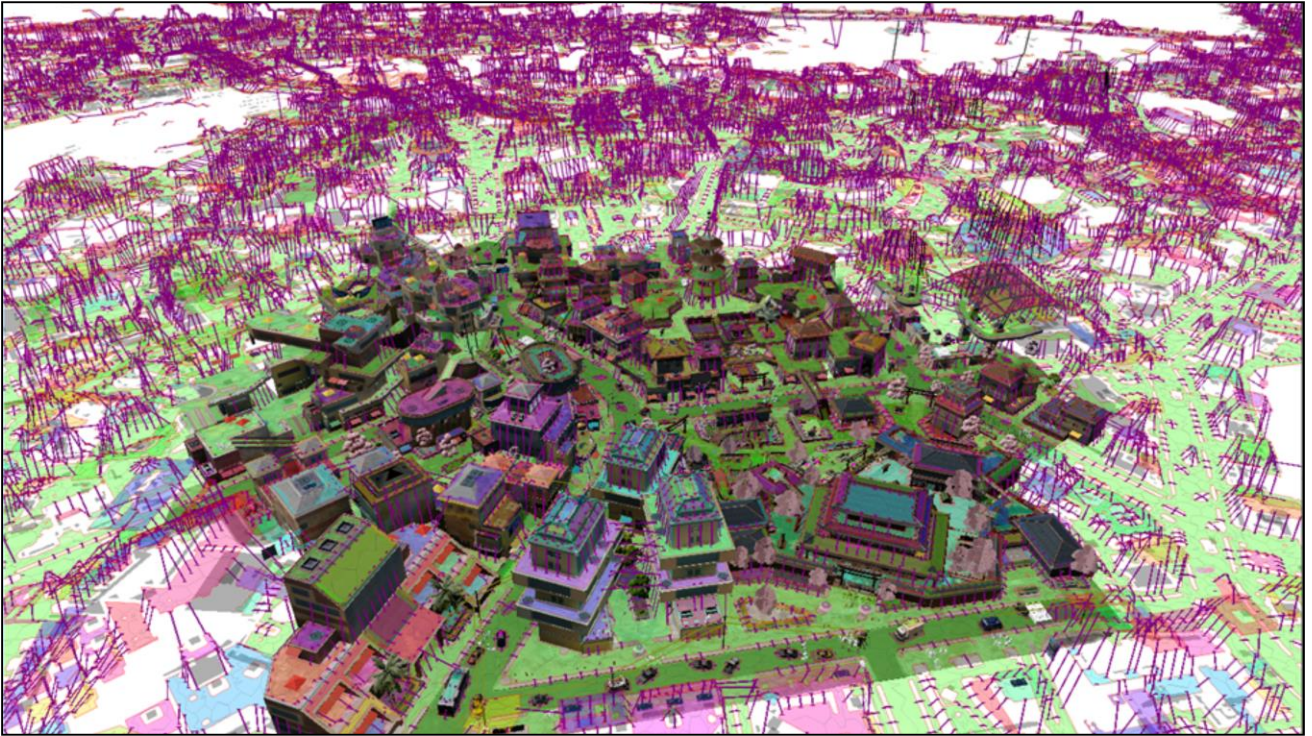
Navmesh is the canonical sticky example. Our navmesh builder from Fuse was designed to work on entire regions all at once, and to keep the navmesh loaded all the time. We struggled for a while with getting it to cut up the world into pieces, but there's also a lot of runtime complexity there – how do you weld together adjacent convexities when they load in? How can an AI reason about navigation across part of the mesh that hasn't been loaded yet?

Because our nav is on a rectangular grid and we load on a hex grid, it's complex to figure out what nav groups actually need to be loaded.

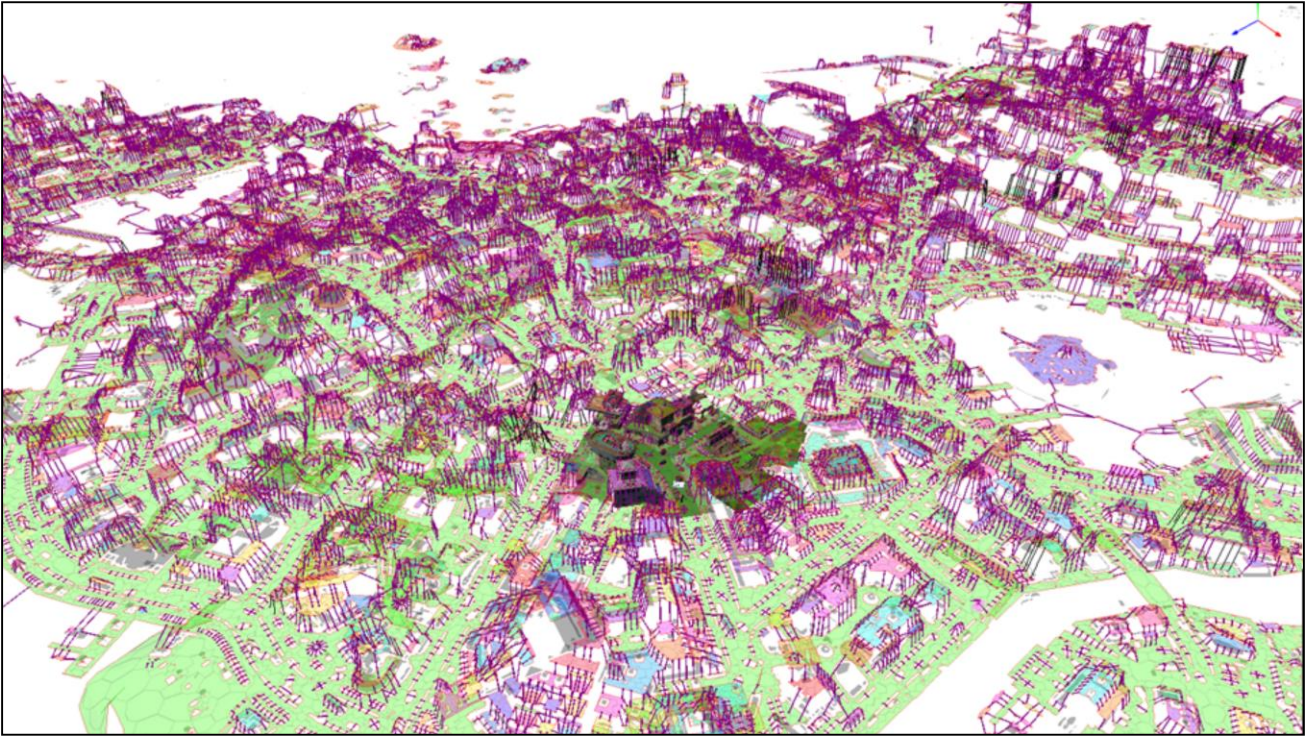
Another difficulty we struggled with was dependency tracking, and how to accurately build subsections when something changes. Nav is the endpoint of the dep chain for all of the

models, and the markup on the models, and composite models. So if you change one static model that's used 100 times then you have to rebuild 100 navmesh islands..

Ultimately, the solution we hit on was

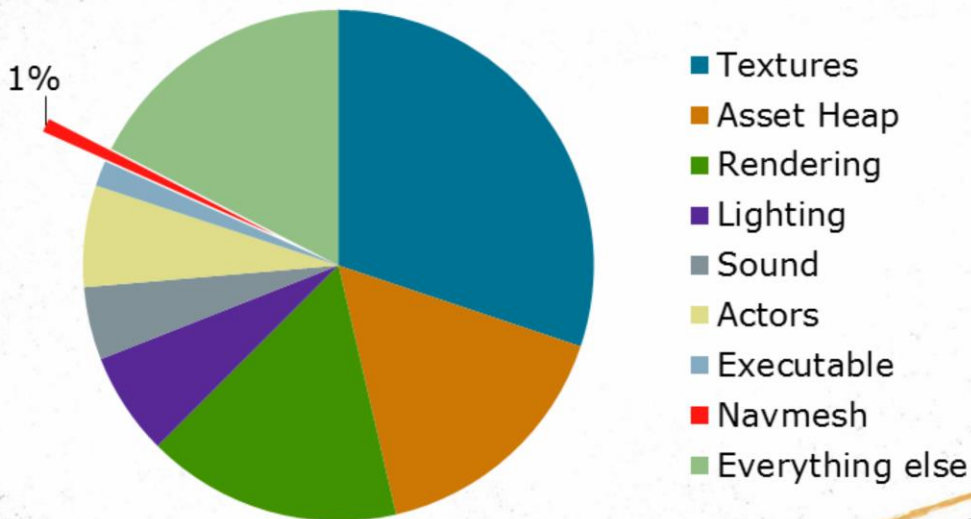


To just keep all of the navmesh loaded in the world all the time. This completely sidestepped all the issues of building it in pieces and welding those pieces together at runtime, and it means that we could stick with our existing tools to a great extent.



On the other hand, the memory cost of having all the navmesh in the world loaded all the time is

Sunset Overdrive Memory Budget



Negligible. It eats up less memory than just the *animations* on the hero character.

When hexes unload, there is still *visible* low-res ground geometry in their place – the stand-ins. But we don't have solid *collision* geometry loaded.

We try to mark the navmesh corresponding to the just-unloaded regions of world as being temporarily impassible, so that AIs don't wander into areas with no collision. But it isn't perfect, and often when you stream out part of the level, some of the characters stand on it will simply plummet through the floor to the killplane many meters below. Did anyone notice?



Things that are hard

There were a bunch of things that we knew we needed to build, and that we planned for. And then there were some surprises. There's always surprises.

Load the
ground
first!



For example. When we're running missions, sometimes the actors for that mission are in a global zone. And that global system doesn't necessarily know what parts of the regular world have streamed in. So, if you've got a mission where there's things standing on the ground, make sure you've got the ground in place before you make the things.



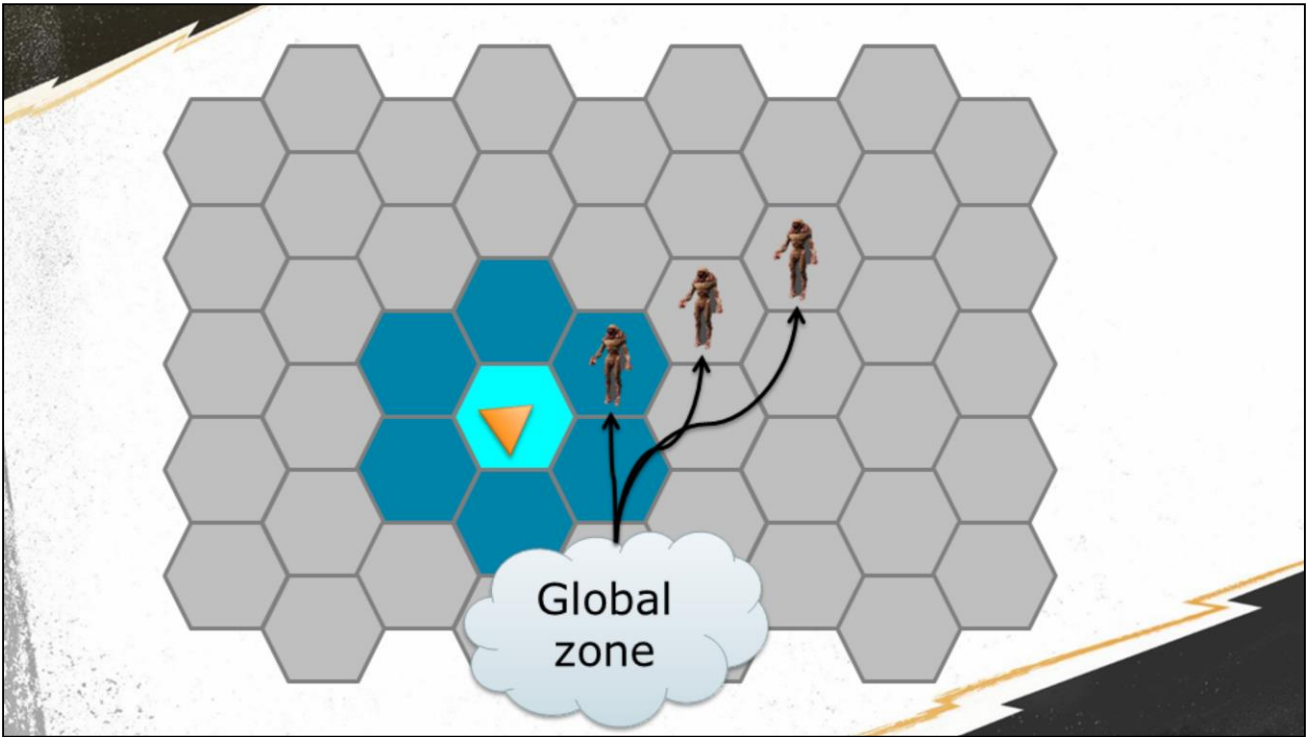
For example, we have this mission where you're chasing a train along some elevated tracks. Both you and the train go on this big chase across the city, at pretty much the maximum possible movement speed.

The design for this mission included enemies in unique setups along the train route, with some custom geometry and behavior.

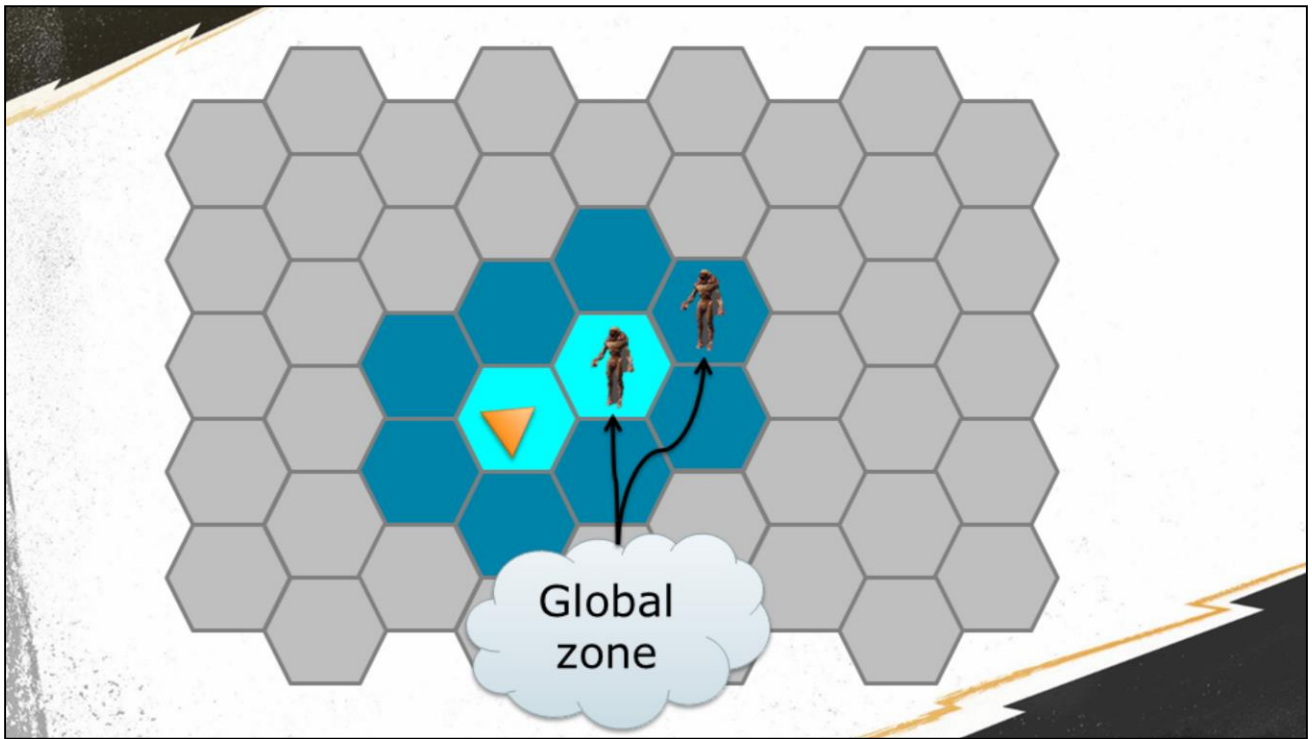
You can see them attacking the player here. They need to be in their places and initialized in time for them to take potshots at the player as he zooms past.

But your movement here is really right at the upper limit of our supported speed. So sometimes you'd be rushing into a part of the world that had only just loaded, or wasn't totally finished loading yet. In that case, the enemies wouldn't have spawned in to attack you, or they might still be too far back or setting up or not have run their AI yet.

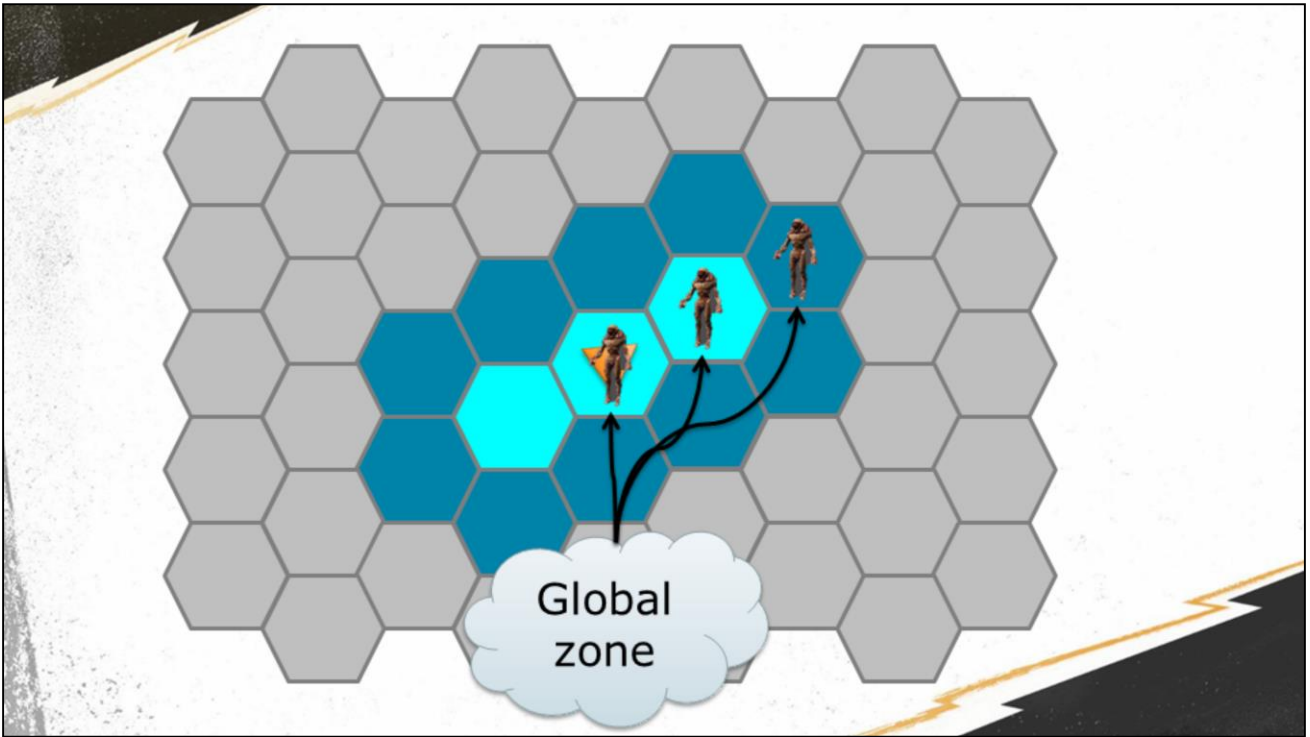
View this video at: <http://youtu.be/tF8su9RdrZQ>



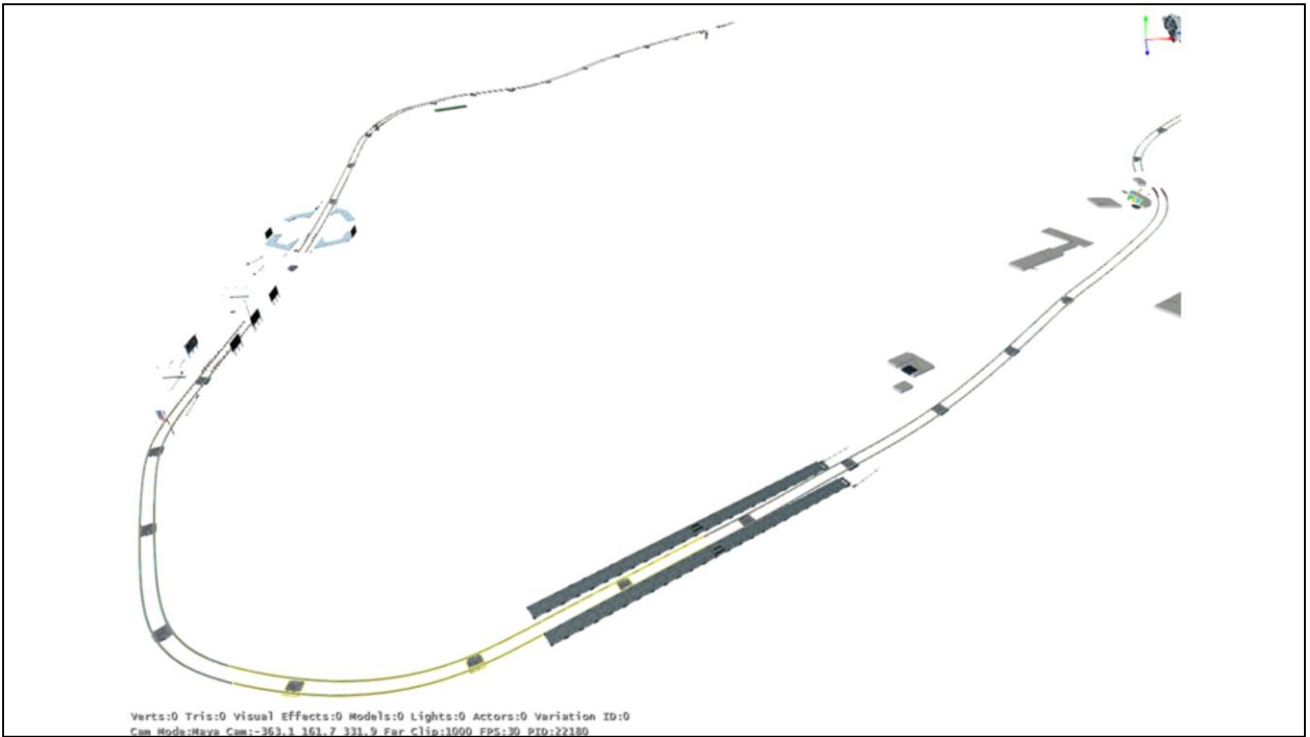
Ok, so first we dealt with this by putting all of the mission in a global zone, and having all of the script in that zone, and then spawn the enemies from script.



As the player and the train moved along the tracks, they'd hit trigger volumes. The trigger volumes would spawn enemies into place so they'd be there to attack you.

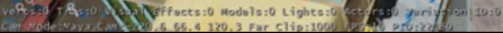


Sometimes – not often but sometimes – you find yourself grinding into a section of the world that hasn't completely loaded yet. And in that case, the enemy might spawn before the ground zone streams in. With the result that, like Wile E. Coyote, the enemy would appear, simulate long enough to realize that there was no ground underneath it, and then plummet out of the world.

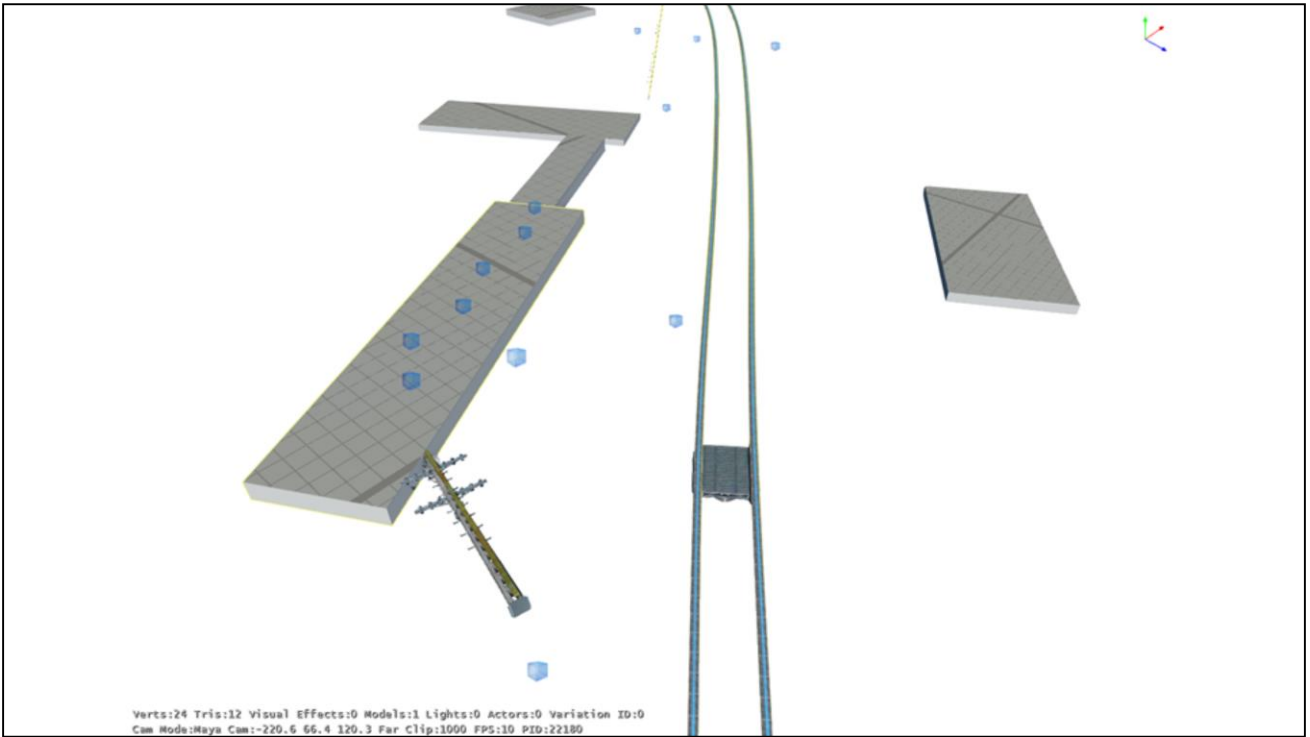


So in this case, the global zone contained the entire train track – while this mission was underway, we disabled the track that was in the world, and replaced it with the one that was duplicated into the mission. That way we could be sure that there was no way you could run off the end of the track while the mission was active.

Also, the global zone had solid collision platforms under the enemy spawners.



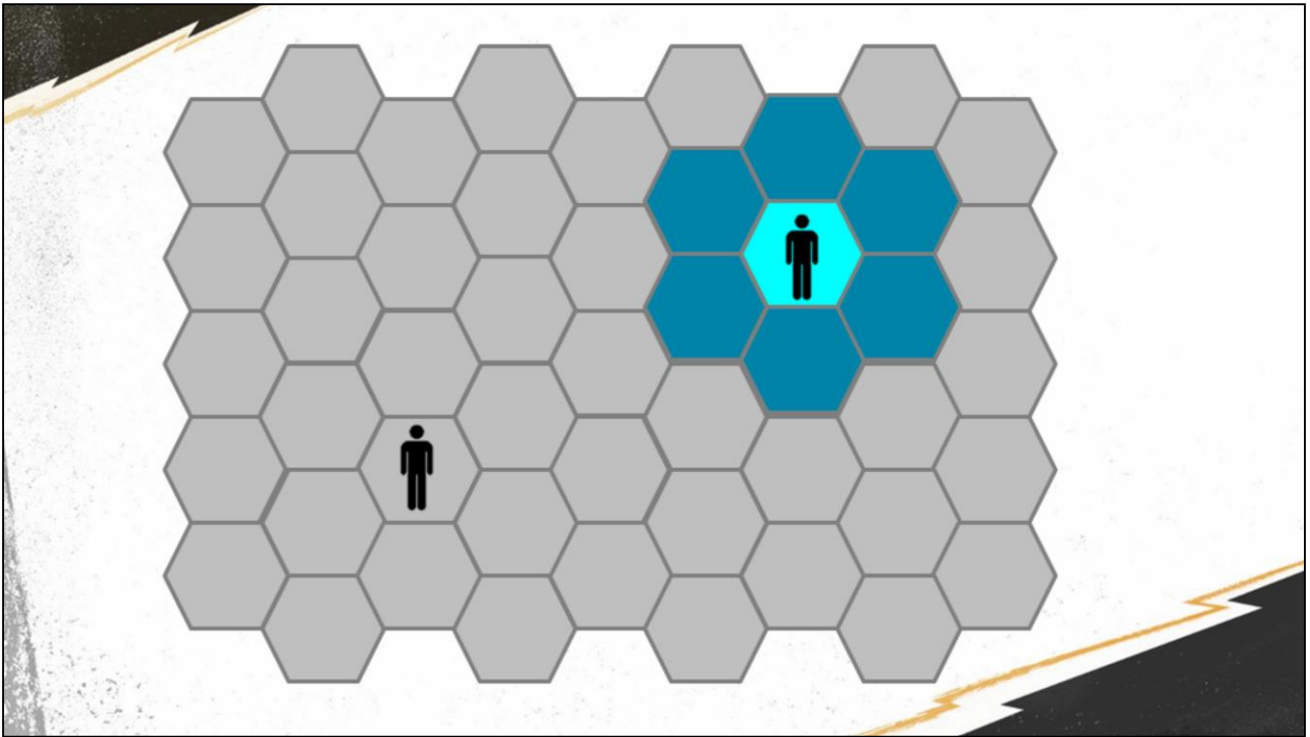
Vertices:0 Triangles:0 Version: Effects:0 Models:0 Lights:0 Actors:0 Environment ID:0
Can Mode: Maya Camera: 20.6 55.4 120.3 Far Clip: 1000 FPS: 16 Proc: 22780



We put little safety platforms just underneath the visible surface of the building.



Teleporting the player, or even just the camera, is really tricky in a streaming game.



If we teleport the player (or the camera) to a location outside the currently loaded zones, he'll find himself surrounded by low-res stand-ins and no ground to stand on. The engine has to scramble to immediately load *all seven* of the surrounding zones at once rather than streaming just two or three as happens during normal movement. And, since the player arrives before the ground does... once again, we have someone plummeting out of the world.



In previous games, often we'd script missions such that when they ended, they would automatically teleport you to the quest-giver for the final cutscene, or to the start of the next mission. That's because we didn't want the player to have to make a long, boring walk back to base. But teleporting gets you into the problems we mentioned earlier, so we tried to design our missions such that they looped around and by the end of them you ended up where the final cutscene was supposed to start, or to where the next mission begins.



Or you can use a bit of stagecraft – like bringing down a proscenium curtain while the stagehands do a scene change.

Image: Radio City Music Hall. Credit Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Radio_City_Music_Hall_Stage_Curtain_1.jpg



Here's a mission that involves chasing a dragon all over the city. We don't know when the player will succeed in killing the dragon, so we don't know where they'll be when the mission ends; but the final cutscene has to take place at a specific location. So we warp the player and the dragon upwards a distance into a pocket dimension, put some animated backgrounds behind them, do this elaborate scene to cover the load, and then carry on from the destination area.

View this video at: <http://youtu.be/kaVSypx2t0A>



Elsewhere in the game, we've got this big setpiece object, this huge radio tower in the middle of the city. You have a big battle there.



But the cutscene that sets up your battle at this place – the one that tells you what to do – takes place in an entirely different part of the world. We need to show you the tower so that you know where to fight, but if we simply teleport the camera there for a close-up, all you will see is the low resolution stand-in. So instead, we just solve the problem in-character, without having to move the camera or force-load things.

View this video at: <http://youtu.be/P0fgZaUzIt0>



The battle that takes place here has you running all around the tower. It involves a lot of vertical and sideways movement; you circumnavigate the entire structure, and really explore the space. You'd think this would mean that we'd have the whole tower loaded all the time, right?

View this video at: http://youtu.be/KtK-KC_NKp8

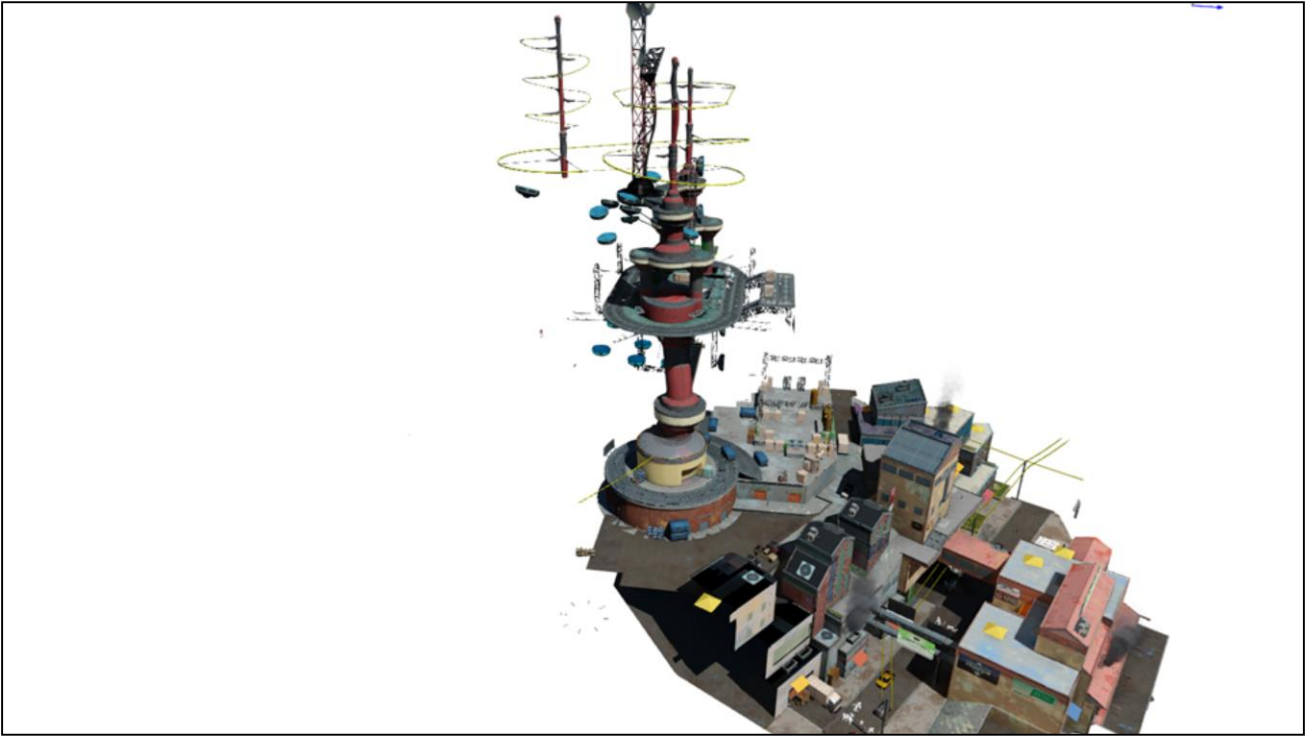


In fact, this tower got built right at the intersection of three hexes. It was one of the very first missions we built, while we were still sussing out the strengths and limitations of the system.

The tower itself isn't bigger than a hex, but it sits right on the seam.



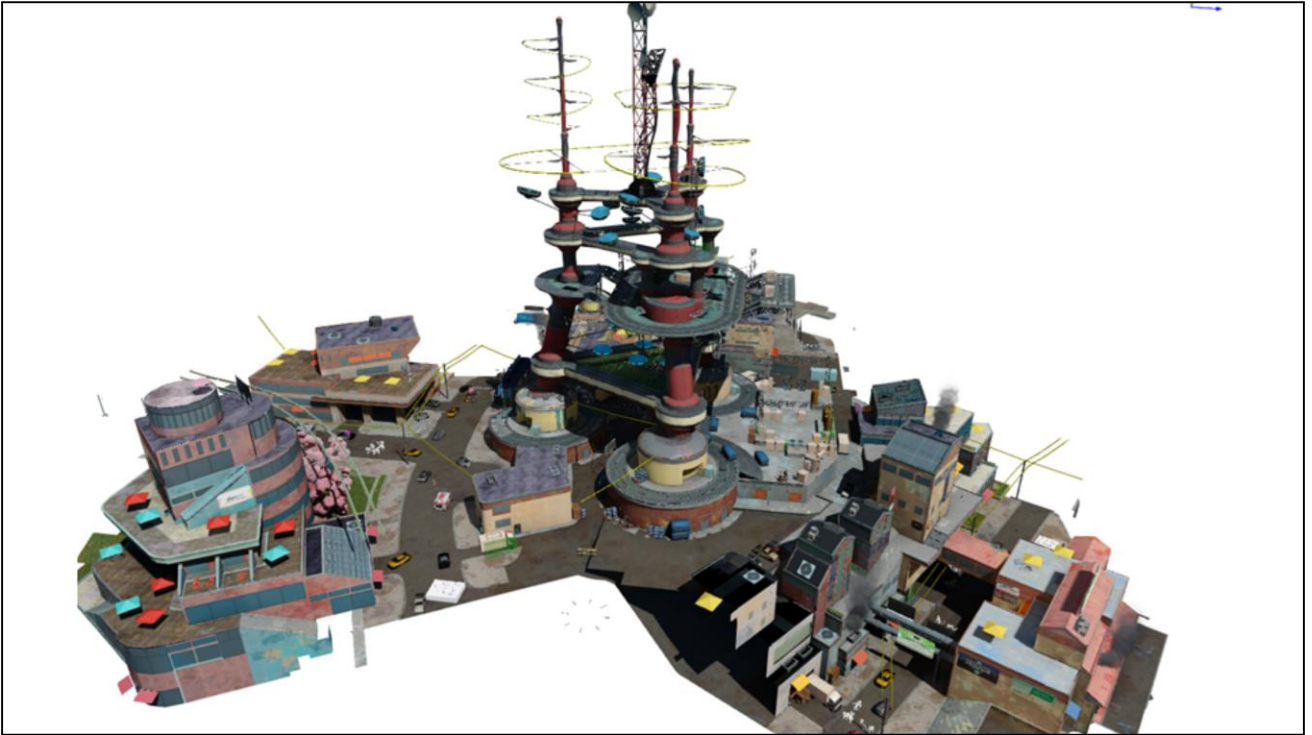
It was this piece..



...and this piece...



... and this piece...



...that made up the structure.

So you're continually loading/unloading regions, and paying the CPU time for that, while just moving back and forth in a single combat setup.

Was it a huge problem? Not really; it just put pressure on IO bandwidth and limited the amount of high detail MIPs and other stuff that we could have in the area. Like I said, this is one of the very first missions we made, and we could have gone back and re-done it...

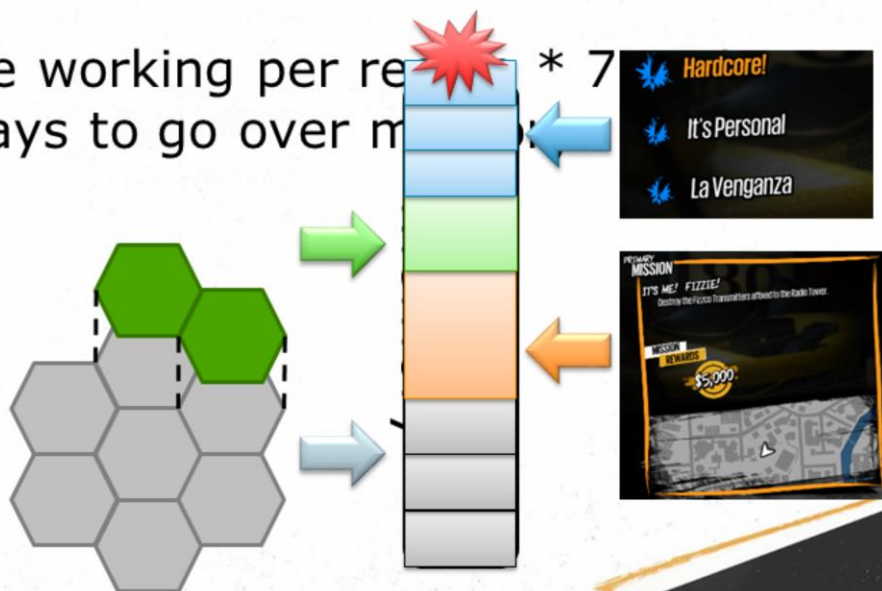


**@TODO:
HUMOROUS PICTURE
ILLUSTRATING
"GOOD ENOUGH"**

...but ultimately, it was fine. When you're in pitched combat like that, you tend not to notice that the building across the street has really low rez textures.

Memory diagnostics are critical

- 4 people working per resource * 7
= 28 ways to go over memory



There are some diagnostic tools whose critical importance we learned only after having been bitten by bugs we couldn't figure out how to fix.

Memory costs were much more unpredictable in sunset than they were in Fuse. You can tell up front how much a world zone will take, except sometimes that varies in the context of its neighbors and which assets are shared and which are not. And the overall footprint while a mission is underway depends on the size of the mission, and its shadow zones, and whatever quests may be loaded (which can be between 0 and all of them). That's a lot of different people, and a lot of opportunities for someone to make an innocent change over there that breaks a thing over here.

Having nice, clear tools that show exactly who is consuming memory, and which things are shared, and in general the weight of things, would have really really helped us early on.

--

The point is that many artists and designers are all inhabiting the same space for different reasons now. The people building the world make spaces that support many different overlaid missions, and the people building missions create encounters that can span many hexes in a long chase scene.

Sometimes the characters from the base combat setups and the dynamic encounters and the mission-specific characters are at odds. They can contend for memory and CPU resources in ways you didn't expect.

We have systems where the mission designer can say "cap the number of proximity spawned ambient world characters at 20, so that I can be sure of room to spawn 60 mission-specific characters."

But even so there were still bugs, and it wasn't always clear what needed to be capped, or what had sent us over the cap.



Other useful diagnostics:

- Visualizing region boundaries in game
- Forcing specific regions to load (or unload)

It's also really helpful to be able to see the streaming boundaries in the world while you're playing, rather than to try to guess at which hexes are loaded and which are low res by looking at the current player coordinates and a paper map you have on your desk.

Also, being able to point at a specific hex in the world and say "load just that one hex, and not the things around it," makes it a lot easier to track down bugs in a specific region that would be hard with all the noise of the living world around them.

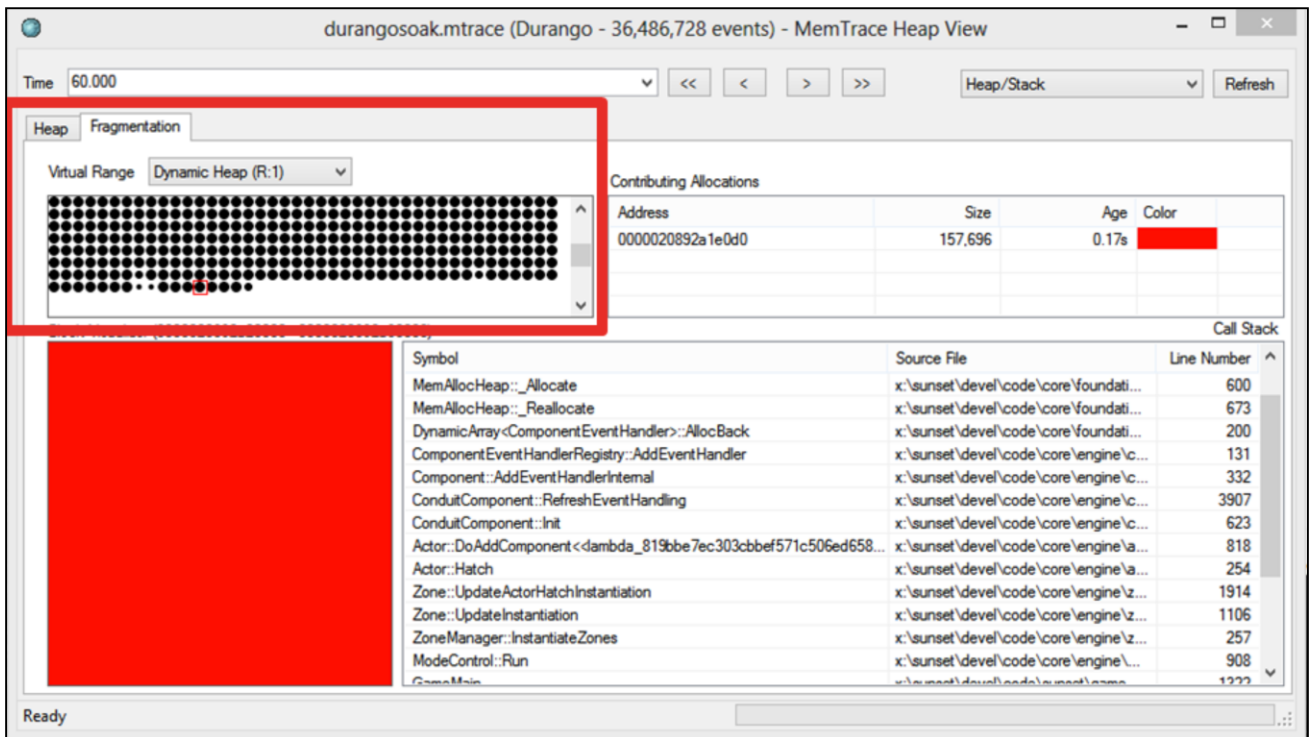


Fragmentation really matters

As game developers we're used to writing fast memory allocators. A bad (which is to say, default) implementation of `malloc()`, or overly liberal ad-hoc allocation in general, can seriously hurt perf in a death-by-a-thousand-cuts kind of way. But in a continually streaming game, *fragmentation* matters as much as speed.

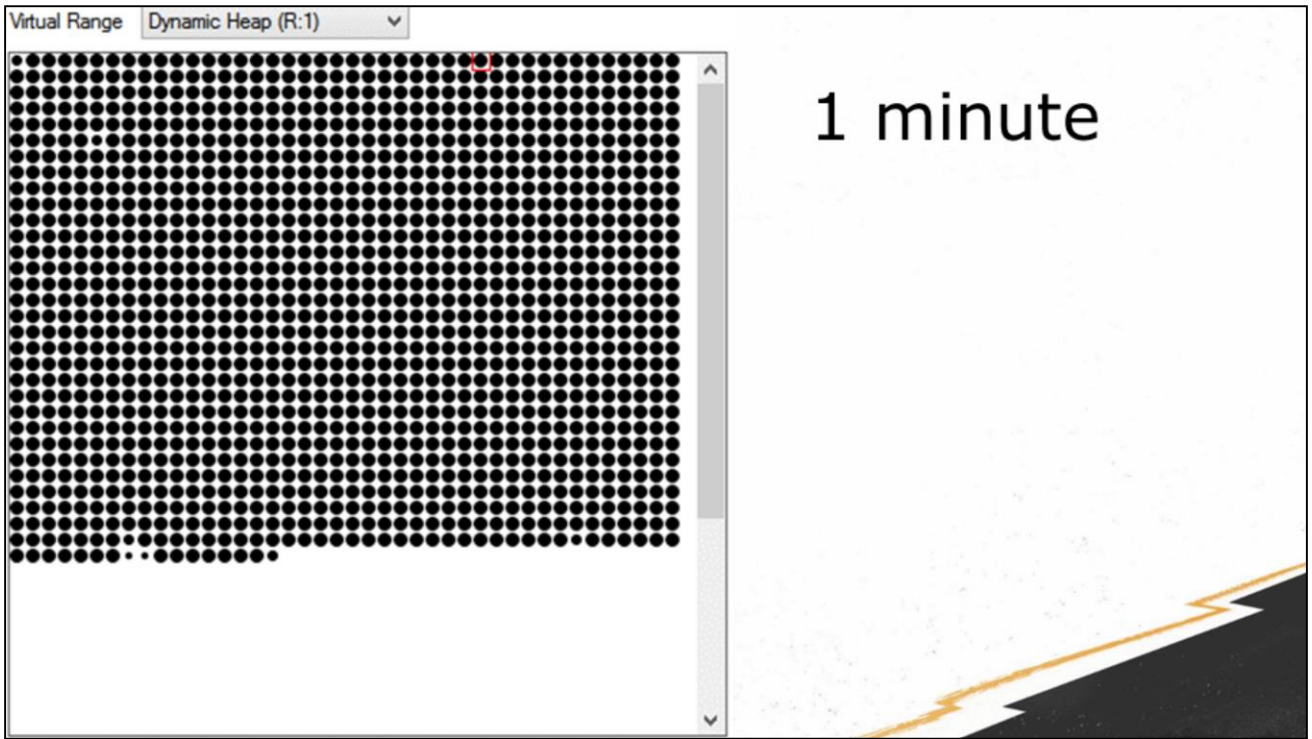
In the old games, the dynamic heap was part of the level, meaning that when we left the level, we freed the entire `malloc()` arena for all the things that had been in it, and started with a nice clean contiguous page of memory for next time. On the PS and PS2, sometimes we even just soft-reset the entire console between levels.

But that cleanup option isn't available in an open world game; you can never just declare jubilee on an arena and start over. Also, in an open world game, you are continually instantiating and destroying things, which hammers the allocator all the harder. The insidious thing is it's death by thousand cuts – no one allocation shows up on the profiler, but they're like a tax that inflates everything else

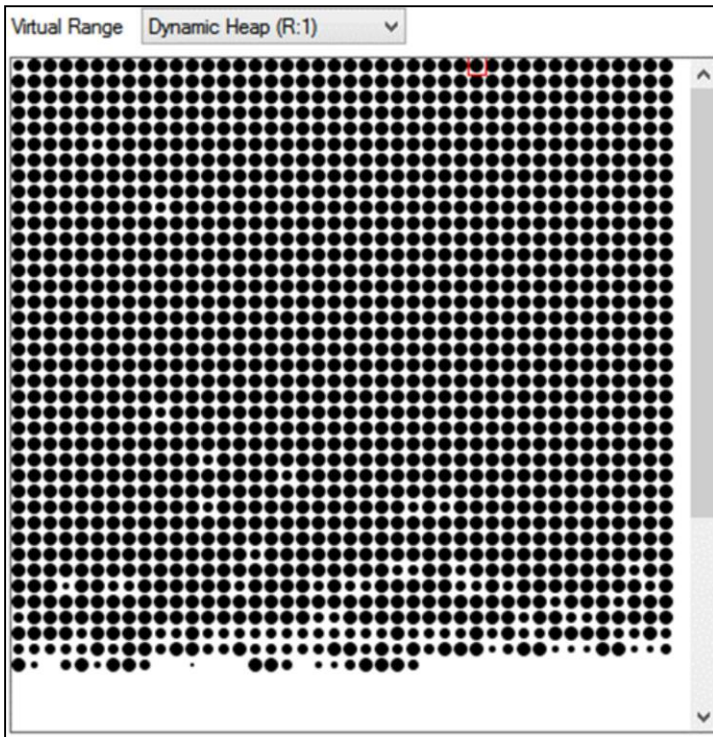


Here's a picture of our internal heap-allocation-tracking tool. It's really handy. It records every single allocation made by the game, so that for every byte that's been allocated we can see the callstack responsible.

The pane at the upper left is used to track fragmentation – each dot represents an entire memory TLB page. If the dot fills its whole square, then the page is 100% utilized. Smaller dots indicate that some space in the page is unallocated; therefore a lot of small dots implies a lot of wasted space.

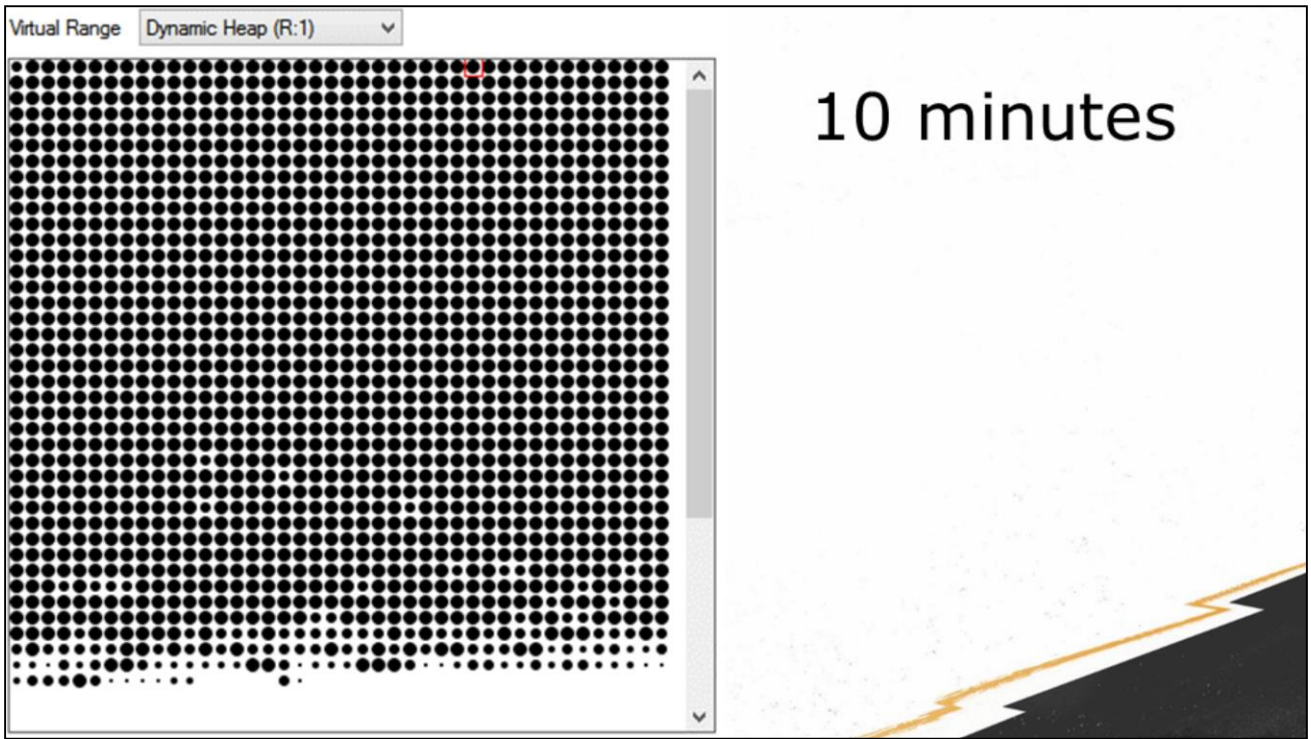


So here's a trace from a build of our game that had a really tiny 37-byte memory leak. One minute in, we're fine.

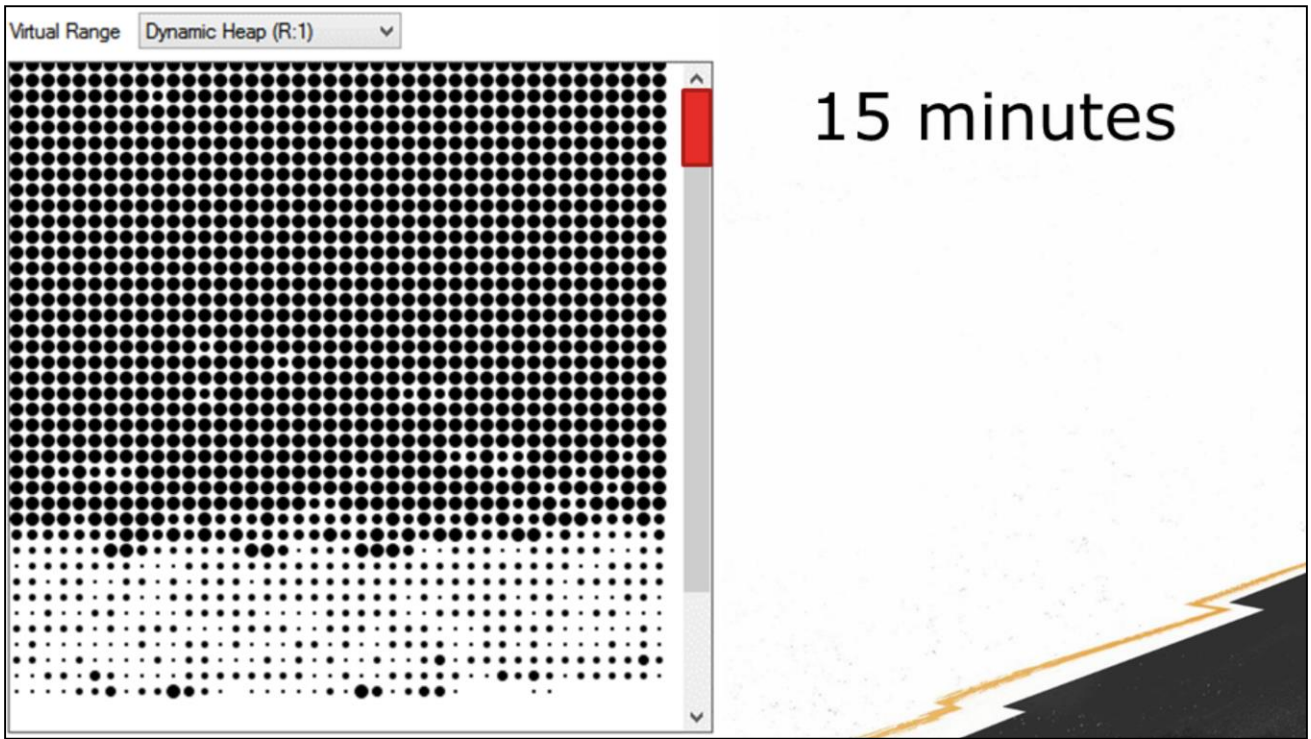


4 minutes,
35 seconds

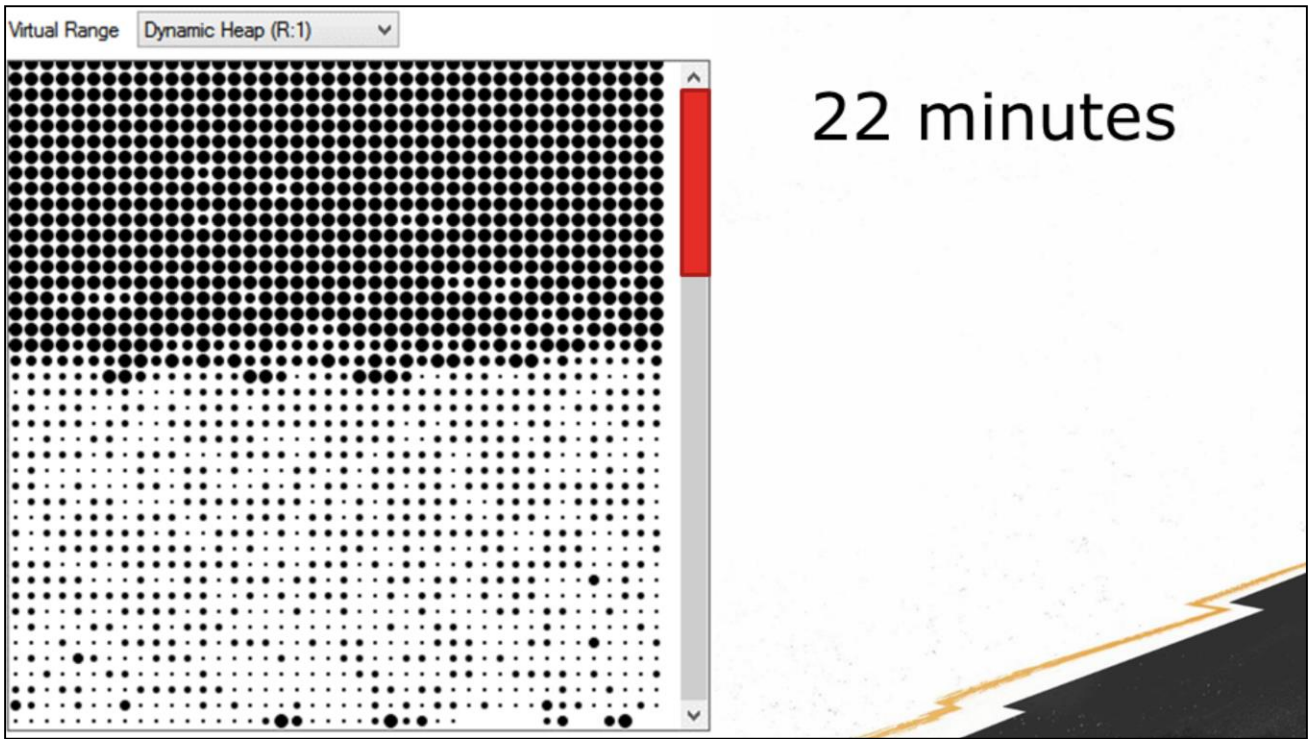
4 minutes, still fine.



Okay, fragmenting a little, but still mostly ok.



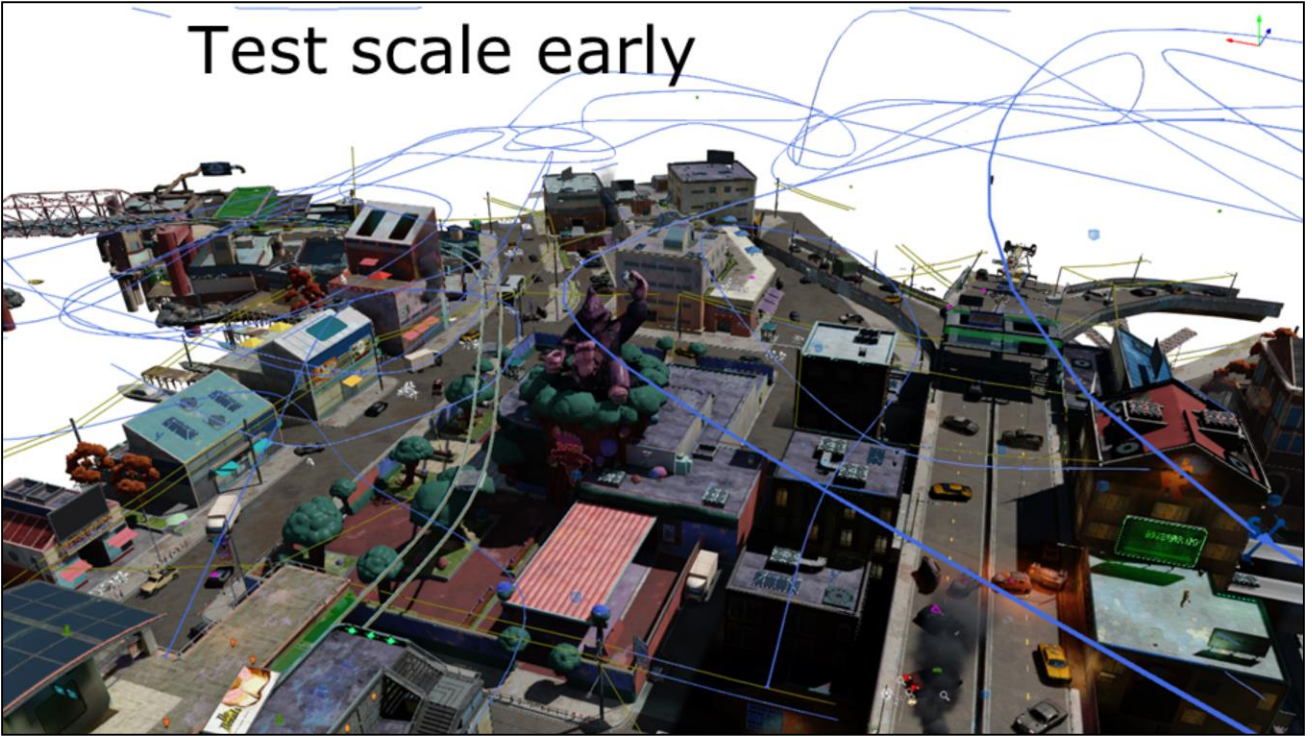
After only fifteen minutes of soaking, we're starting to get into trouble. Not only is there a lot of fragmentation, but you can see that the allocated address space no longer fits in one window.



22 minutes in and we have a problem. The game started to crash during allocations! But it had plenty of total free memory left. If we asked the allocator how much memory we had, it would have said 100mb free. And we could have allocated 100 one-megabyte objects. But a single ten-megabyte object would crash, because we didn't have any contiguous regions of 10mb.

Memory allocation is one thing we really, really had to sweat over in SO. Things that were tiny problems in linear games became very magnified in an open world. Also, finding this issue would have been really hard without this tool – another example of how critical good memory diagnostics are.

Test scale early



Test your scale early

Have a populated region of world early for stress testing.

Ideally, find a way to semiautomatically generate large datasets for testing.

Or at least, strive hard early on in your project, to make a region as object-dense as you expect the whole world to be.

You can try to plan ahead for the things that you know will be problems – 100 enemies take ten times as much memory as 10. But there are a lot of things you *don't* know will be problems until you've tried to simulate 10,000 of them at once. Pigeons, for example.

- AIs, entity logic, script...
- Memory can be extrapolated, CPU is less linear
- 120 zombies > 10 × 12 zombies



LOD all the things – not just for memory and render, but also for CPU load.

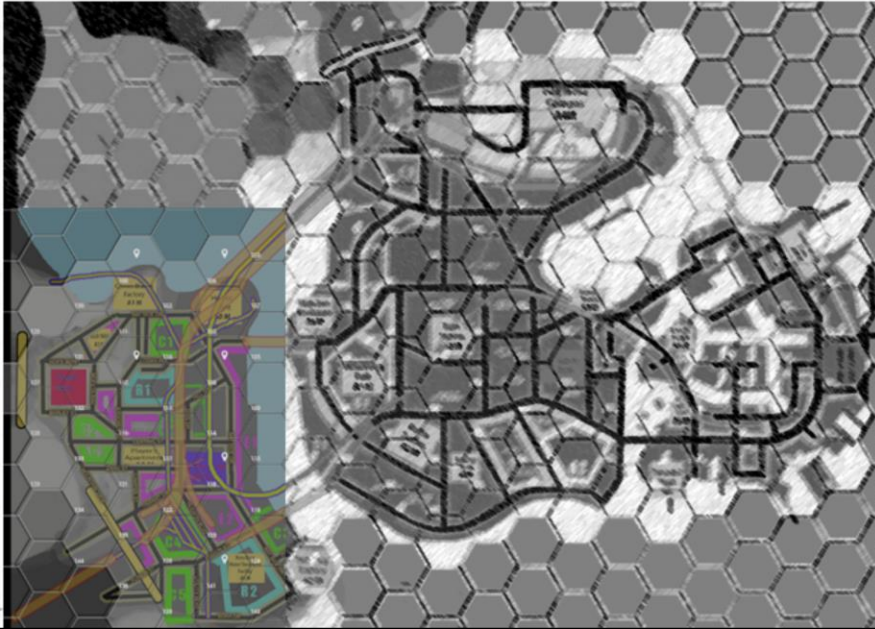
We have levels of detail on NPC AI – the further an NPC is from the player, the dumber it gets, and the choppy its animation becomes. We didn't need it in a game with 16 simultaneous enemies but we sure need it in a game with 100 zombies on screen along with 12 allies.

Try to design systems so that you can anticipate their peak scales.

Memory costs can be extrapolated. CPU perf is less linear.

The only real way to determine how scalable a thing is to try to understand how the feature will be used.

Plan ahead for streaming install



This is a thing that most games have now, but it's still a lot easier if you plan for it early in development. We retrofit it late in the project, and it was kind of a pain because we needed all of the low-LOD hexes installed as part of the first "batch" of content, even when you were just playing the tutorial section.



In conclusion:

You can live with "good enough"

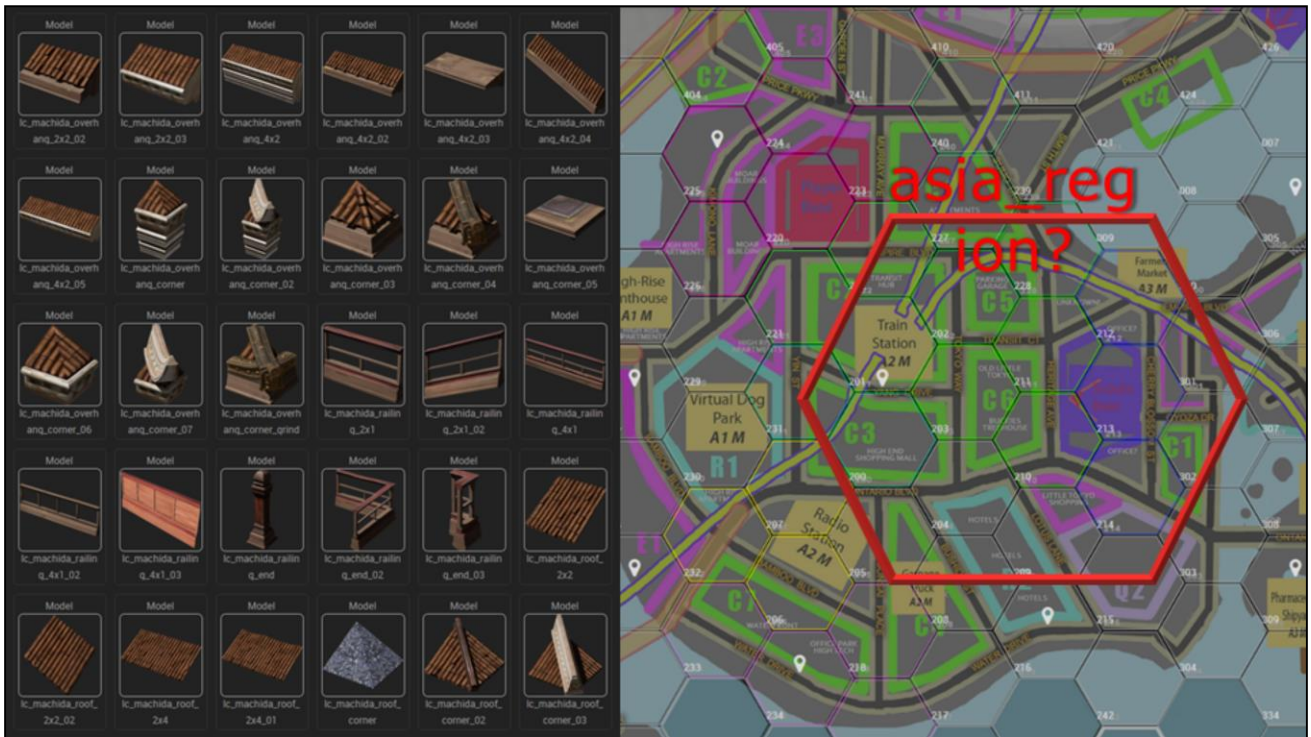
- We could have had...
 - Middle-LOD zones
 - Smaller hexes
 - Elaborate asset combinatorics

As you've seen, we made a lot of compromises where we incurred runtime inefficiencies in order to conserve developer time.

And it came out fine. The game works.

For example, we could have had middle-LOD zones.

We could have reduced the average hex size, and the time it takes to stream anything, by carefully managing combinatorics to reduce redundant loads.



For example, every hex in the Chinatown area of the game redundantly includes a lot of the same models and textures (Asian lanterns, torii, etc)


If we had created a single, larger “Chinatown-wide” uber-zone that encompassed the whole region, we could have put all those common assets into it and left them out of the individual hexes.

It would have taken longer than an ordinary hex to load, but you would only do it on entering and exiting the uber-region, and we could have hidden that by eg making you enter via an underground tunnel or some other occluder.

But this would have required a lot of manual management of what assets go where, and code in the hex-file-builder that knows to omit redundancy in these cases. It would have been a lot more work for art and production, just to achieve something that *feels* like a smarter engineering solution. But would that cleverness have really helped the customer? Or did giving the artists the flexibility to put any asset in any hex

make the game better?

The simplistic redundant data approach means that our zone loads are on average slower than they would otherwise be, but it also means that we were able to build 160 hexes!



Don't let the perfect be the enemy of the good.

Don't let the perfect be the enemy of the good. You can't possibly make the perfect, totally efficient streaming engine within a single title's development. And you don't need to.

What's more important is that you come up with something that's tractable for designers to build with, has clear constraints so that everyone knows how to live within them, and is reliable enough that you can develop gameplay features without having to continually clean up core engine.



Resist rebuilding key systems

Don't rebuild key systems. You'll have your hands full making existing systems scale.

Every sequel or new franchise feels like an opportunity to throw out that gameplay system you felt had become unwieldy and design something new. Resist that urge if this is going to be your first open world game. Simply refitting your existing tech to handle tens of thousands of things will keep you more than busy.



Pizza should not contain avocado.

There've been lots of great talks by really smart people at GDC about how to build streaming games. And sometimes it will seem like you need *all* that tech, working perfectly, for your game to work at all. But what matters is what you need to make a fun game. Build just enough tech to prototype how you want your game to play; test the prototype, and then fix what needs to be fixed.

Thanks to...

Mark Cerny

Adam Noonchester

Andreas Fredriksson

Bryan Intihar

Cameron Christian

Doug Sheahan

Ian McMeans

Jason Priest

Jonathan Adamczewski

Jonathan Garrett

Jonathan Hunt

Mark Stuart

Mike Acton

Neil Walker

Peter Kao

Ron Piekert



Questions?