



SIGGRAPH 2026
Los Angeles 19-23 JUL

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

SLIM: Scaling User-Generated 3D Worlds on Roblox

Sergey Makeev
Senior Technical Director at Roblox



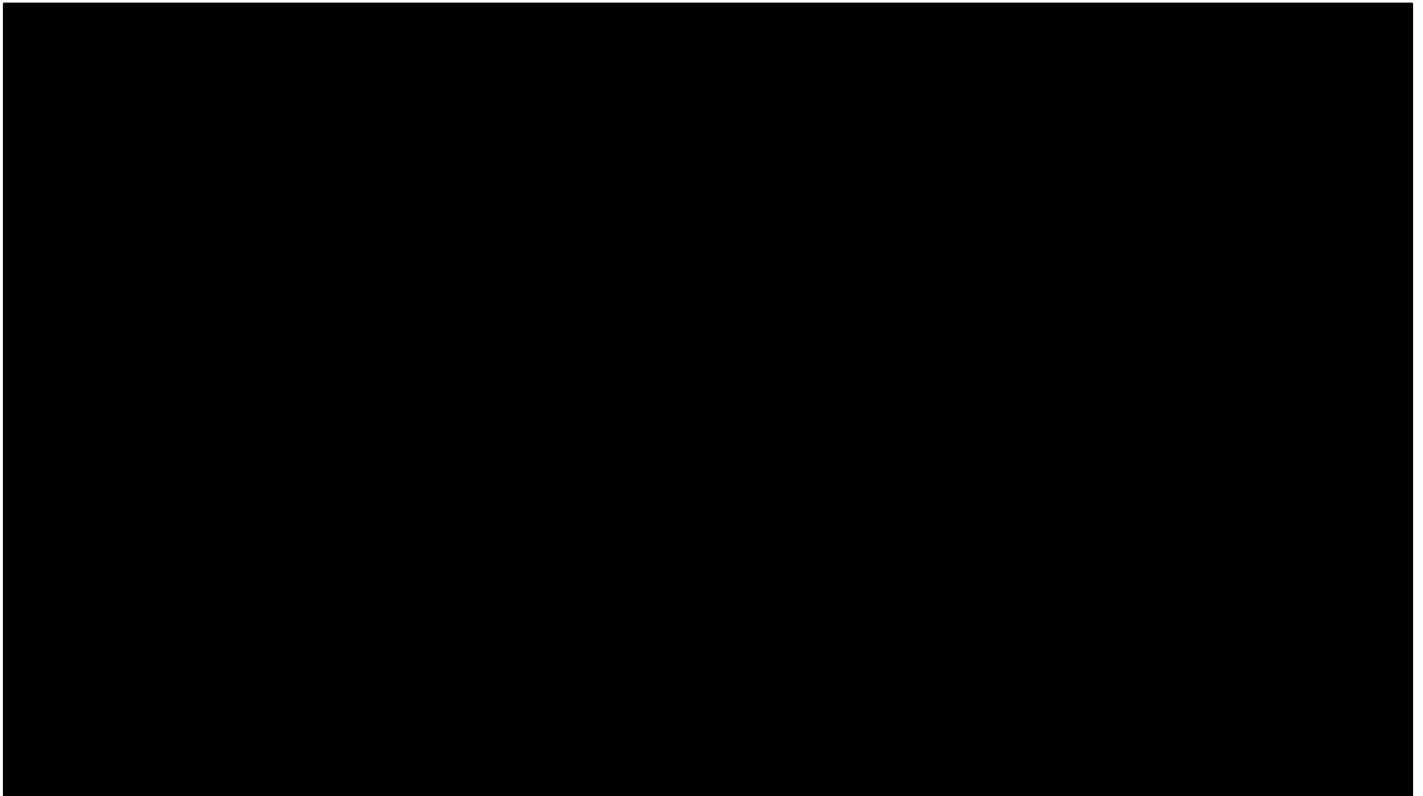
© 2025 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.



Hello, everyone. My name is Sergey Makeev, and I'm a Senior Technical Director at Roblox.

Today I'll talk about scaling user-generated 3D worlds: why rendering on Roblox is a hard problem and how we approached it.

I'll give you some context about the platform, show you a demo, and then dig into how it all works.



First, let's watch a video showing different Roblox games.

[play video]

When people hear "Roblox," they usually picture simple, blocky games - like the ones you see here.

And that's fair - those games are a big part of our history. But Roblox creators build all kinds of games, across many genres and visual styles.

And some of these worlds are really large and detailed.

One thing to keep in mind while watching: every game here runs on the same engine, and the same version of its code and content must run well on every device we support.

That range is a big part of why rendering at Roblox is hard.

I'll let you enjoy the video.

Roblox is a platform

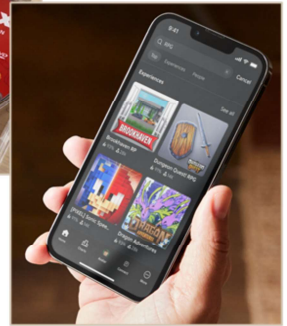
SIGGRAPH



- Not a game / virtual world
- Not a game engine / middleware
- **Platform** = Engine + Cloud + Content + Community
 - *Creative platform*: everything is user-generated (UGC)
- Delivers to PC, Mobile, Console, VR, and Smart TVs

From a rendering perspective:

- No imposed art direction
- No manual optimizations
- No fixed content budget
- No single game or hardware target to optimize



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

So, what is Roblox, exactly?

It's not one game, not a single virtual world, and not an engine that others can license. It is the platform: the engine, the cloud services, the content system, and the community of creators and players on top.

Everything you saw in that video is user-generated. Creators make worlds, avatars, clothing, and tools - and the platform delivers it all to consoles, PCs, phones, tablets, VR headsets, and even smart TVs.

For a rendering engineer, this removes every assumption you'd normally lean on. There is no single art direction, no fixed content budget, and no known hardware target. The renderer has to handle whatever creators build next.

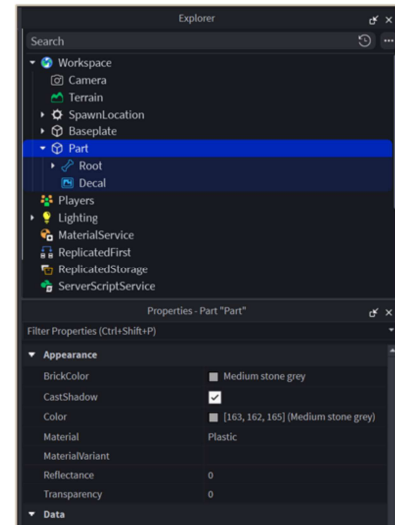
To see why that's hard, let's look at the world from a rendering perspective.



- Data model = live object tree
 - Instances, properties, scripts, and references to assets
 - Server authoritative
 - Roblox Studio uses the same engine as runtime
 - Always editable, scriptable, and collaborative

Rendering consequences:

- “Static” is a runtime assumption - not a permanent property
 - Stable long enough → optimized rendering path
 - Change detected → dynamic rendering path
 - Scripts, replication, physics, etc.
- Update only what changed - the world never stops changing



At the center of every Roblox world is the data model. It's a live tree of objects, properties, scripts, and references to assets. Think of it as the HTML DOM, but for a 3D world.

The game server owns the simulation and updates this tree. Scripts and physics can change anything at any time. Each client receives only the part of the tree it needs.

Our editor, Roblox Studio, uses the same data model and the same engine code, with collaborative multiplayer editing on top.

Here's why this matters: on Roblox, "static" is an assumption, never a guarantee. That rules out most standard tricks that depend on offline optimizations or a one-time bake.

A typical Roblox world just keeps changing, and it never reaches a stable state.

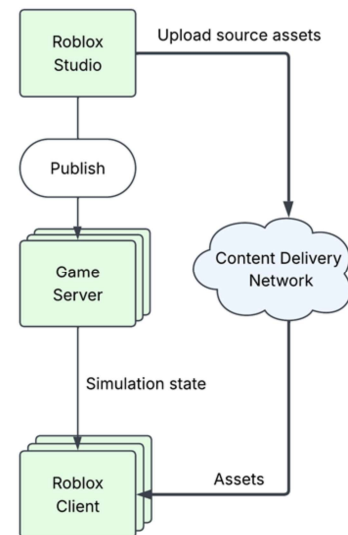
Live on every device in a single click



- Publishing
 - Edit → Publish once → one version, immediately available, everywhere
- Join like opening a web page
 - Simulation state (Data Model) replicated from the game server
 - Assets (images, meshes etc...) fetched independently from CDNs
 - Only the Roblox client is installed (~13 MB)

Rendering consequences:

- State and Content stream continuously - no steady state
- No guaranteed build-time optimization point
- Device-specific content representations are derived after content publishing



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

Now let's look at the same problem from the content delivery side.

A creator publishes a game to every platform with a single click. They click the "Publish" button, and the new version is immediately available on all devices. Players don't install a build or wait for a patch - joining a Roblox game is closer to opening a web page.

When a player joins, the server starts replicating the live data model, while meshes, textures, and audio arrive separately from the CDN. And everything keeps changing while this happens - the simulation runs, and assets stream in and out as the player moves around.

So again, there's no steady state and no single point where we can say, "The world is final now."

Millions of assets must work everywhere



1.2M

meshes uploaded **daily**

1.6M

images uploaded **daily**

26%

of top 1,000 games updated **daily**

8 platforms

Windows, macOS, iOS, Android,
Xbox, PlayStation, Meta Quest, Smart
TVs

~800x compute range

GPU architectures, memory,
bandwidth, thermals

20+ years of content

700+ Petabytes of stored data

© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

And the volume of content is huge.

Every day, creators upload about 1.2 million meshes and 1.6 million images. On any given day, about 26 percent of our top 1,000 games ship an update that goes live to millions of players.

All of that must work across eight platforms, even though our high-end devices have about 800 times more computing power than our low-end devices.

We have also accumulated 20 years of content - over 700 petabytes. And all of this still has to run on devices that didn't exist when it was created.

So, there's no "content complete" step either. New content keeps arriving, old content has to keep working, and the hardware target never stops moving.

A UGC-shaped demo, built by a non-artist



- Kitbashed from catalog assets
- Uneven density and overlapping geometry
- Per-asset LODs exist → the composition is the problem
- Rendered naively:
 - 172,531 instances
 - 59.4M triangles per frame
 - 3,526 draw calls per frame
- Too expensive even on desktop - yet it must run on phones



© 2026 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

For this talk, I built a demo that represents a typical Roblox world.

Fair warning: I'm a graphics engineer, not an artist. But I built it the way many creators do - by kitbashing thousands of assets from the catalog.

And it has exactly the properties you'd expect: uneven triangle and texture density, tons of small objects, and lots of overlapping geometry. Even when each asset is fine on its own, hundreds of thousands of them don't add up to an efficient scene.

This scene has approximately 172,000 data model instances. Rendered naively, that's 59.4 million triangles and about 3,500 draw calls per frame.

That's too expensive to render even on a desktop - and yet it must run on a \$100 phone.



But here it is.

[play video]

It's running in real time at 4K on a desktop, using our optimized representation, which we call SLIM.

Same demo, on a phone

SIGGRAPH



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

And this is the same scene on a phone.

[play video]

It's the same data model, the same assets, and the same engine.

There is no separate mobile version - the only difference is the runtime representation.

Three pillars behind that result

SIGGRAPH



Naive: **59.4M** → Desktop: **4.78M** → Tablet: **2.4M** → Phone: **1.1M** triangles/frame

How did we do it?

Asset Transcoders

- Keep stable, versioned source assets
- Generate texture formats, compressed meshes, and LODs
- Re-derive on demand as devices and algorithms evolve

SLIM world transcoder

- Extract render-relevant state from the live data model
- Merge geometry and materials; simplify and remove hidden geometry
- Generate multiple fidelity levels

Runtime orchestration

- Adapt to memory, performance, and network budgets
- Maximize visible quality within those budgets
- Stream updates and fall back as the world changes

Here are the numbers. Naively, rendering the scene requires about 59 million triangles per frame. With SLIM, that drops to 4.7 million triangles on desktop, 2.4 million on tablet, and just 1.1 million on phone.

So how is that possible?

We built the system on three pillars.

First, asset transcoders: we keep creator content as stable, versioned source data and derive whatever runtime forms we need.

Second, the SLIM world transcoder - the same concept, but applied to an entire world instead of just a single asset.

And third, runtime orchestration: each client picks the best detail that fits its memory, rendering, and network budgets.



Asset Transcoders

Let's start with asset transcoders. The idea is simple: preserve the source and derive the runtime form.

A live world references stable source assets



- **The world changes continuously**
 - Instances, properties, scripts, and physics state change at runtime
 - Creation, destruction, and modification are expected anytime

```
TestGame.rbxl x RainbowColorScript x
1 local part = Instance.new("Part")
2 part.Name = "Brick"
3 part.Anchored = true
4 part.Parent = workspace
5 part.CFrame = CFrame.new(Vector3.new(0, 10, 0))
6
7 local hue_cycle_speed = 0.5
8
9 local hue = 0
10 for i=1,300 do
11     hue = (hue + hue_cycle_speed * task.wait()) % 1
12     part.Color = Color3.fromHSV(hue, 1, 1)
13 end
14
15 part:Destroy()
16
17
```

As we saw, Roblox worlds never settle down. Instances come and go, scripts change properties, and physics moves things around. That's not an edge case - that's normal behavior.

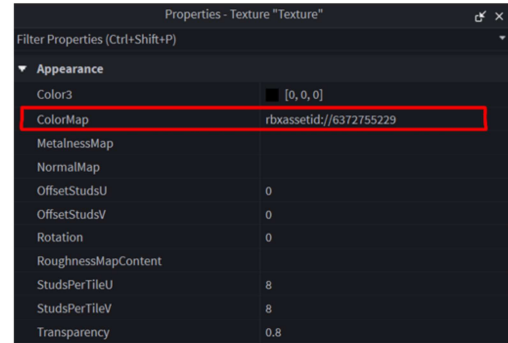
We can't freeze the world and run an offline build or bake.

But that doesn't mean everything changes all the time.

A live world references stable source assets



- **The world changes continuously**
 - Instances, properties, scripts, and physics state change at runtime
 - Creation, destruction, and modification are expected anytime
- **Source asset versions remain stable**
 - Immutable, addressable, and versioned
 - Edits create a new version; existing references remain valid



The live data model is mutable. The assets it references are not.

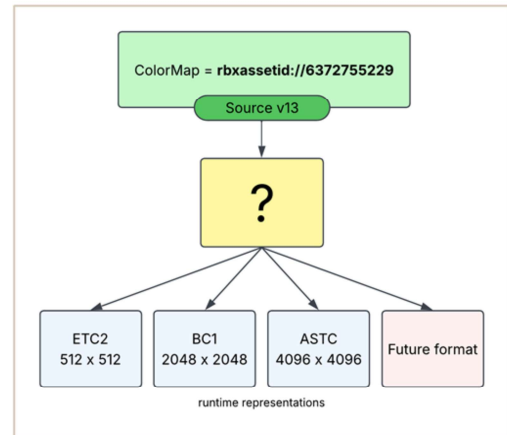
An asset reference points to one exact, immutable version: specific texture bytes and specific mesh data. When a creator edits an asset, we create a new version - we never touch the bytes under the old one.

So even while the world keeps changing, asset references give us stable inputs. That's the foundation everything else stands on.

A live world references stable source assets



- **The world changes continuously**
 - Instances, properties, scripts, and physics state change at runtime
 - Creation, destruction, and modification are expected
- **Source asset versions remain stable**
 - Immutable, addressable, and versioned
 - Edits create a new version; existing references remain valid
- **The runtime target keeps moving**
 - Devices need different formats, layouts, and fidelity levels
 - New hardware and better algorithms arrive after publishing
 - No creator re-export or republish required



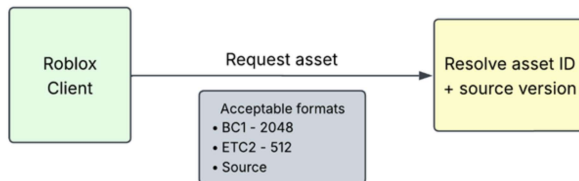
But a stable source doesn't mean a fixed runtime format.

When an asset is created, we don't know its future requirements. Compression formats vary across platforms. Encoders keep improving. We add new platforms, and our in-house formats evolve - long after the content was published.

One permanent encoding simply cannot serve every device.

And we never ask creators to republish old content. Existing content must become forward-compatible automatically, which, at first, sounds impossible.

The Content Platform is the missing build step



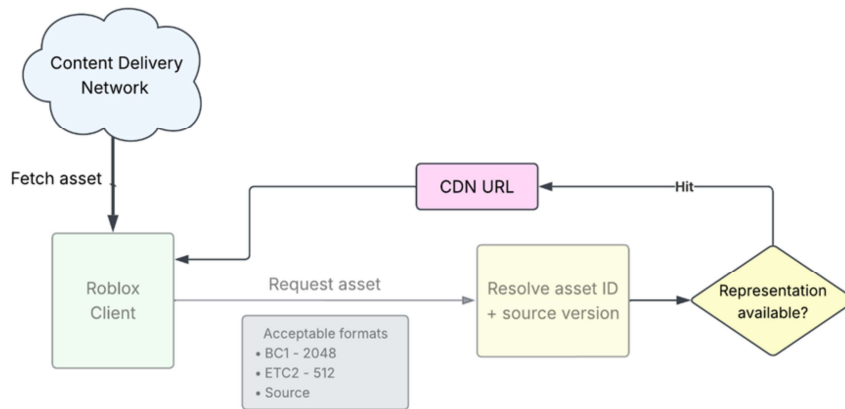
The answer is the Roblox content platform. Let me walk you through one asset fetch request.

The client takes an asset reference from the data model and sends the asset ID, the exact version, and an ordered list of formats it understands.

In this example, it requests a texture: BC1 first, then ETC2, then the original source. The order is important - the first entry is the preferred format, and the rest are fallbacks.

The content platform resolves that ID, version, and format into actual content bytes.

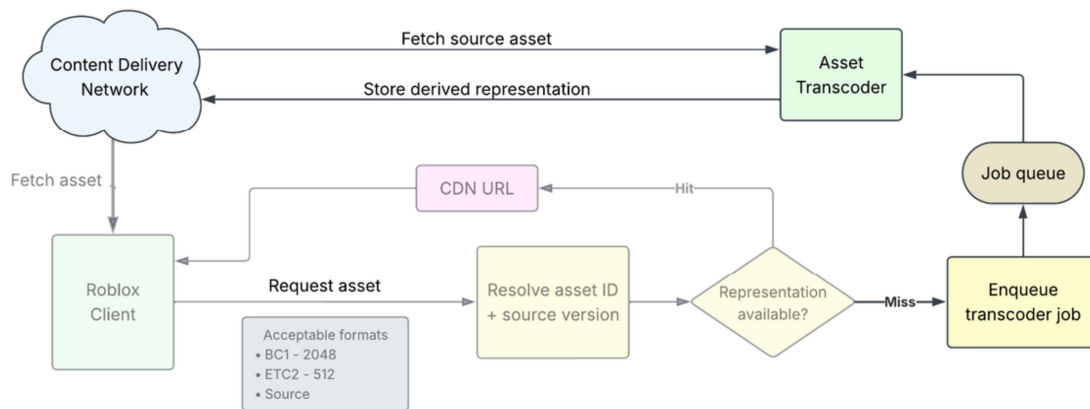
The Content Platform is the missing build step



The best case is a cache hit. The requested form already exists, so the platform returns a CDN URL, and the client downloads it directly.

Nothing gets rebuilt - every client asking for the same thing gets the same result.

The Content Platform is the missing build step

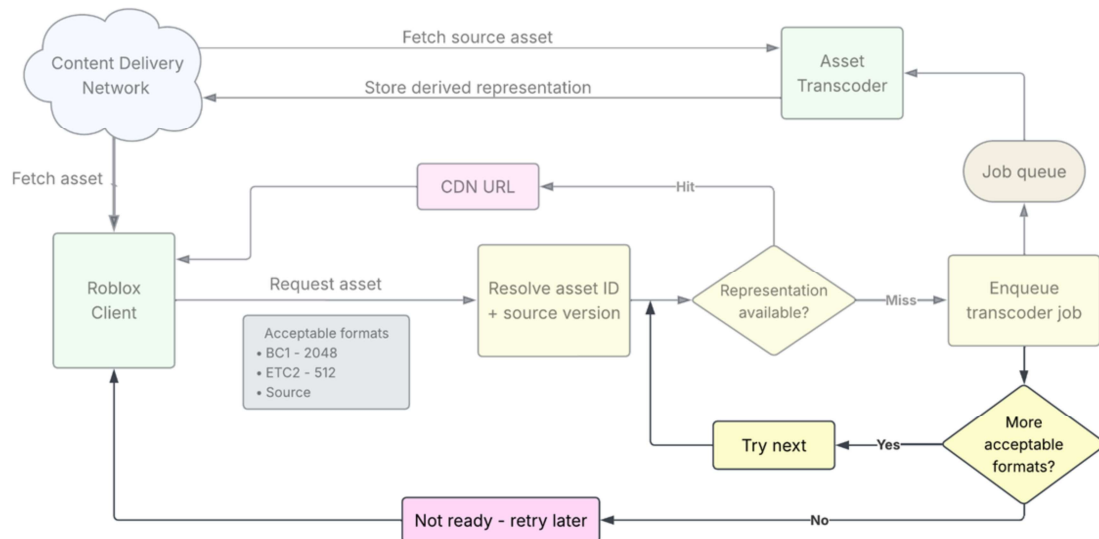


Now let's look at a cache miss. Remember, the creator uploaded only the source; the runtime formats are derived from it.

When the desired representation is not available, the platform submits a transcoder job. The job goes into a queue, a transcoder picks it up, fetches the exact source version, and processes it - in this case, by generating mipmaps and compressing the texture. The result is published as a new representation of that asset.

From then on, it's just cached content - every subsequent request reuses it. But this work is asynchronous, so the miss path needs a safe fallback.

The Content Platform is the missing build step

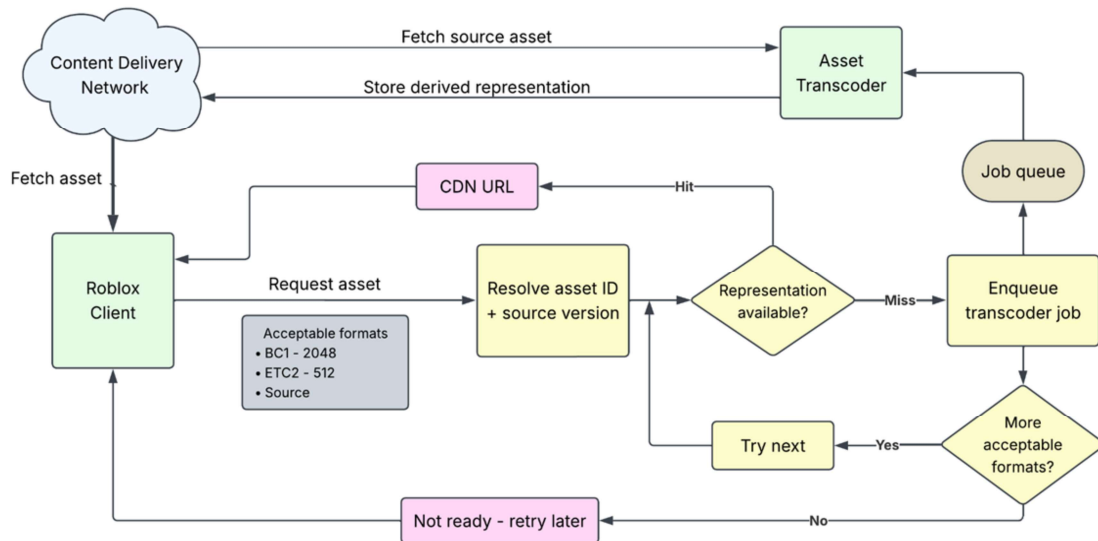


The client doesn't wait for transcoding to finish.

If the preferred format is missing, the platform tries the next one on the list. If a lower-priority format is ready, the client uses it while the preferred format is generated in the cloud. If nothing is ready, the service says, "Not ready - retry later," and that's exactly what the client does.

The key point here is that fallback is not an error path. It's part of the content contract, and it lets new formats appear without ever blocking the game.

The Content Platform is the missing build step



So, here's the whole picture. Traditional asset cooking didn't just magically disappear - it moved inside the content platform, hidden from users and running on demand.

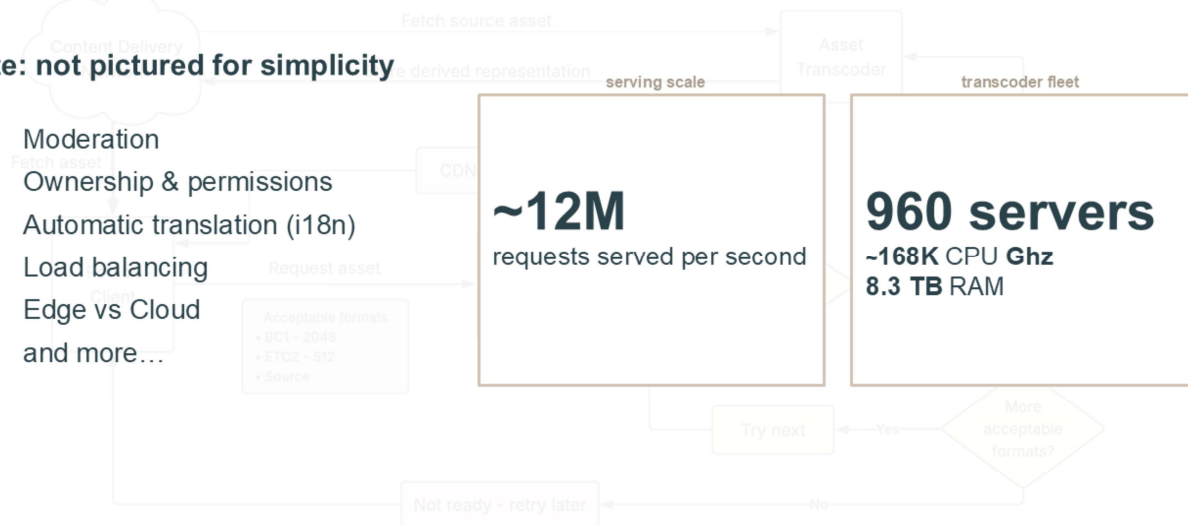
One stable source produces any number of runtime forms, and every result is cached and shared.

The Content Platform is the missing build step



Note: not pictured for simplicity

- Moderation
- Ownership & permissions
- Automatic translation (i18n)
- Load balancing
- Edge vs Cloud
- and more...



This diagram was simplified, of course. The real system also handles moderation, permissions, localization, load balancing, and much more.

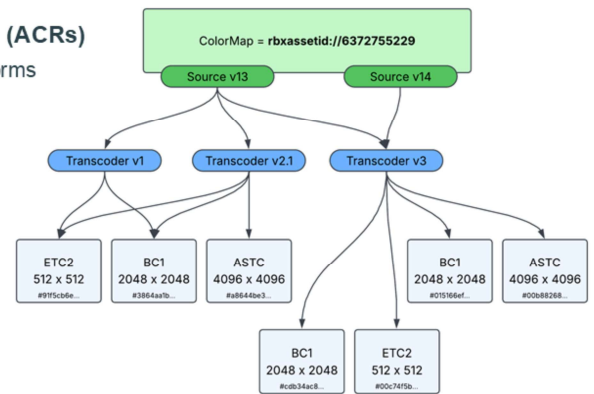
At production scale, it serves about 12 million requests per second. Our transcoder fleet has 960 servers, with roughly 168,000 gigahertz of aggregate CPU capacity and 8.3 terabytes of RAM.

Now let's define these derived forms properly.

One source version produces many runtime representations



- One stable asset ID can have multiple (immutable) versions
- Each can generate multiple **Alternative Content Representation (ACRs)**
 - Texture encodings, mesh LODs, occlusion data, other runtime forms
 - ACRs describe representations - not device classes
- Two independent version axes
 - Source changes → new source version
 - Transcoder changes → new ACR format version
- ACR data is content-addressed, cached, and reused
 - Identical outputs are automatically deduplicated
- ACRs are disposable; only the source is permanent



We call each derived form an alternative content representation, or ACR for short.

One asset ID has multiple immutable source versions, and each source version produces many ACRs: texture encodings, mesh LODs, occlusion data, and so on.

Keep in mind that there are two independent version axes. When a creator edits an asset, we get a new source version. When the transcoder code changes, we get a new ACR format version - from the same source.

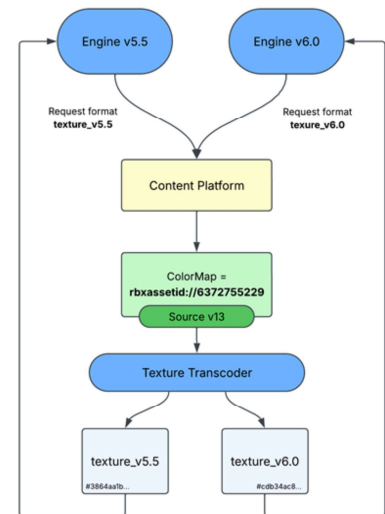
ACRs are content-addressed, cached, and deduplicated. The source is permanent, but every ACR is disposable because the system can regenerate it at any time.

One source version can serve multiple engine versions



Each engine requests a compatible ACR; old and new formats coexist during rollout

- The source asset and its version remain unchanged
- The engine controls which ACR formats it requests
 - Older engine → existing ACR format
 - Newer engine → new ACR format
- Both representations are cached and served side-by-side
 - No global conversion or coordinated client upgrade
- Old formats age out after clients stop requesting them
 - The backend garbage-collects unused ACRs



This gives us another very useful property: old and new engine versions can coexist, even when the asset format changes.

Each engine version requests only the formats it understands, and the platform serves different ACRs from the same source. There is no global conversion and no coordinated update.

And when old clients stop asking for an old format, the platform simply garbage-collects it.

Move the pipeline, keep the representation



Transcoders are deterministic: the same input produces byte-identical output, wherever the pipeline runs

Placement balances: latency, available compute, reuse and distribution, trust and authority, cost

Cloud

- Durable, reusable ACRs
- Cached and delivered through the CDN
- Shared across sessions and games

Roblox Studio/Client

- Immediate preview
- Cached locally (on disk)
- No shared cache required

Game Server

- Live, editable session assets
- Low-latency processing near the simulation
- Cached on server (disk)
- Shared within the game session; bypasses the CDN

One more nice property: the same input always produces the same output, no matter where the code runs. So the execution location becomes a deployment choice.

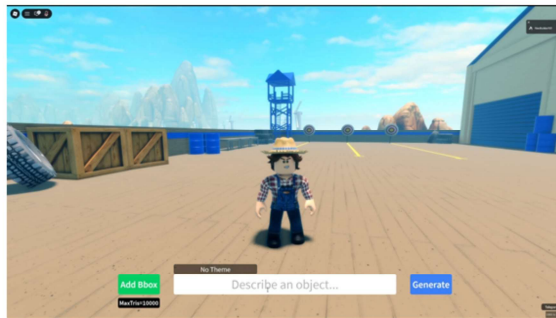
In the cloud, a transcoder produces results that are shared with millions of players.

We can run the same transcoder in Studio or on a client, and the same code will generate an instant local preview.

Or we can run it on a game server, where it can process content created inside a live game session.

Latency, compute, reuse, and trust determine the placement - not the code.

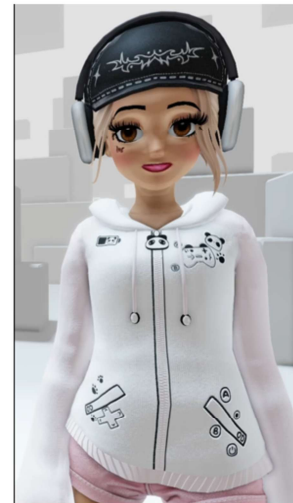
Move the pipeline, keep the representation



Cube4D

Cube4D can be used inside a running game, we transcode on the game server (local session).

Makeup uses two transcoders: on-client for instant preview when changing makeup; on-cloud for better compression and cross-experience caching.



Makeup

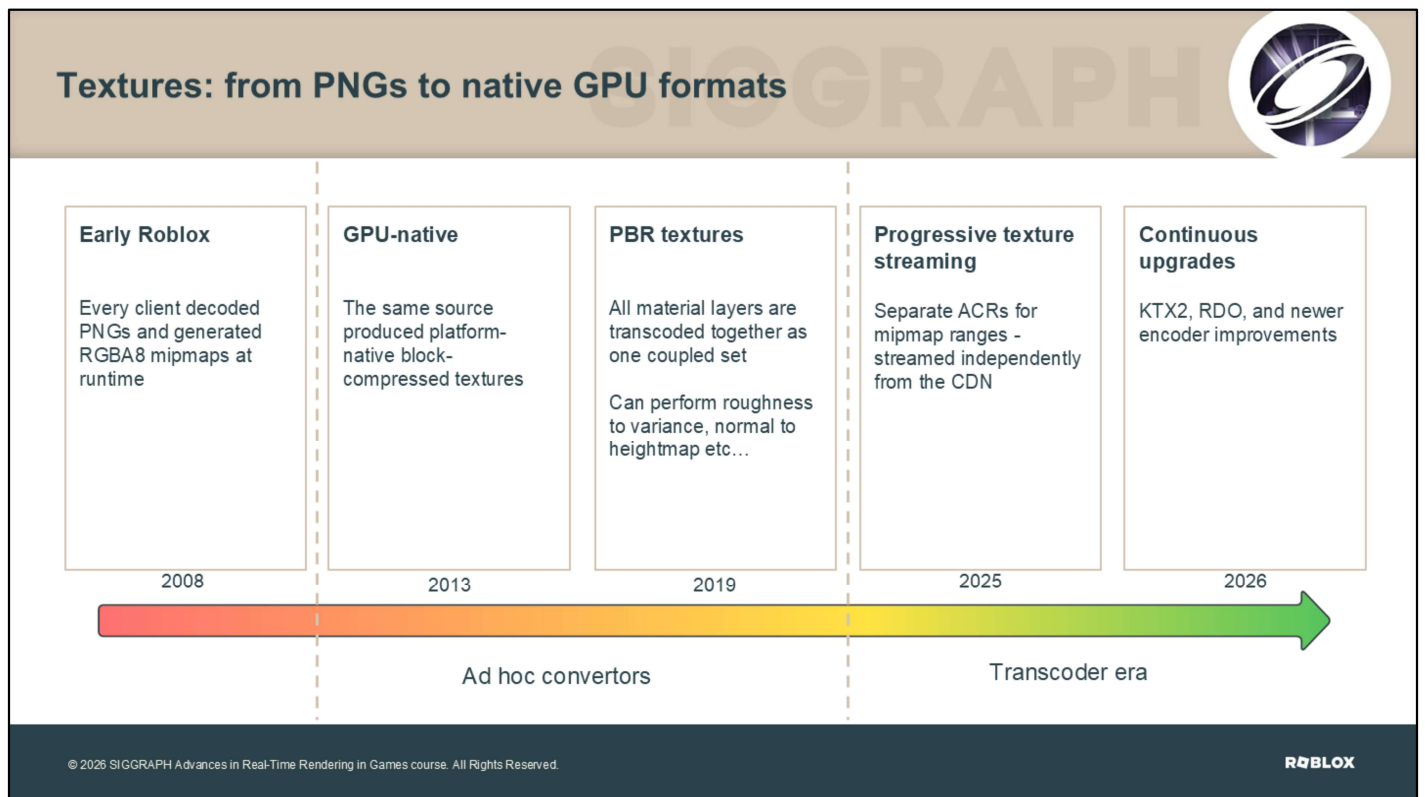
Here are two real examples.

The video on top is Cube4D - our generative AI model that creates content inside a running game. A player describes an object, and the model produces meshes, textures, and scripts right in the game session.

In that case, transcoding happens on the game server. The result can be used for that session, published to the platform, or discarded if the player doesn't like it.

The video on the right is Roblox makeup - a layered texture that the player edits directly. That fires two transcoder jobs: one local job for an instant, player-only preview, and one cloud job for safety checks, better compression, and cross-game sharing.

It's the same contract in both cases - the work just runs where it's needed.



Here's how the texture pipeline evolved under this model.

In the early days, every client downloaded a PNG, decoded it, and generated mipmaps. Then it created an uncompressed RGBA texture at runtime. It worked, but every client paid that cost.

Around 2013, we started producing GPU-native, block-compressed textures using an ad hoc cloud texture converter. It handled textures only and supported just a few hard-coded formats.

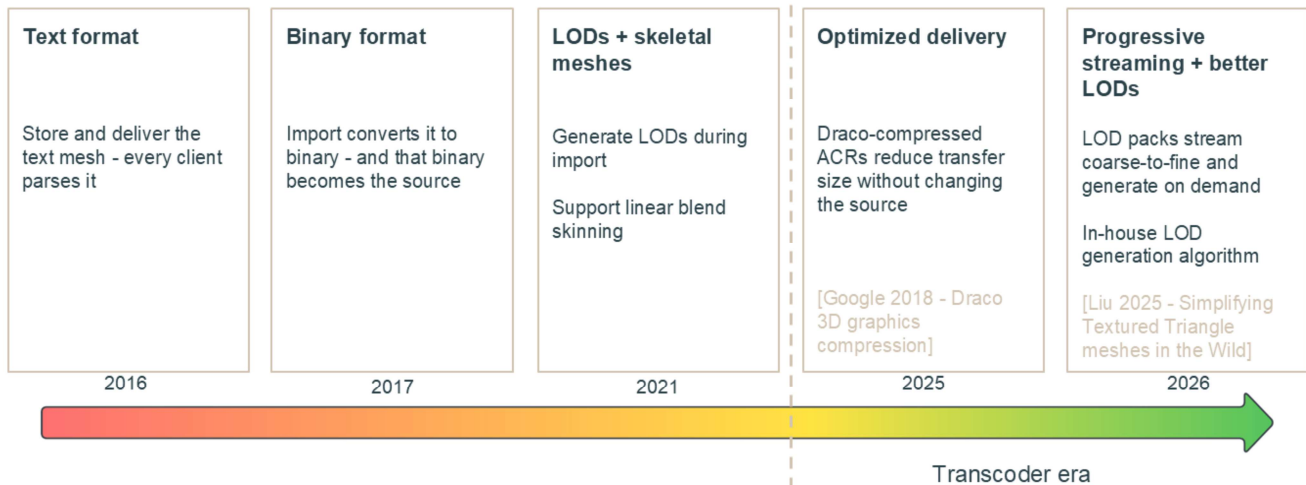
In 2019, we added PBR materials. Those texture layers must be processed together because the normal map affects roughness when we generate mips. That pushed us into the transcoder era.

With transcoders, we split textures into multiple ACRs in 2025, allowing them to stream progressively from the CDN. We also shipped rate-distortion optimization, which noticeably reduced texture download sizes.

Today, the transcoder generates KTX2 textures using modern encoders, and it supports progressive texture streaming.

And the source never changed. A texture uploaded 20 years ago gets all of this automatically.

Meshes: From ASCII to state of the art LODs



© 2026 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

Meshes followed a similar path - with a small detour in the middle.

We started with a simple text-based format in 2016; every client parsed those text files on load. In 2017, we started generating binary files during import - a client-side processing step before upload. By 2021, import also supported skinned meshes and automatically generated LODs. But all that processing happened before upload, effectively changing the definition of the source asset.

And here's the problem: every improvement created another mesh format because we couldn't change old meshes. At the peak, the engine supported ten different mesh formats, including the original text-based one.

LODs suffered from the same issue. LODs were generated only once, during import, so better algorithms and bug fixes never improved existing assets.

Then we moved to transcoders. We shipped Draco compression to reduce download sizes. We plugged our newest mesh reducer into the transcoder, and every existing mesh got better LODs without anyone reuploading anything.

And now we're adding progressive mesh streaming on top of that.

Publishing freezes the source - not the optimization



Publishing freezes the source, not the optimization

- **~27K/min**
Texture ACRs generated
- **~24K/min**
Mesh ACRs generated
- **136M+/hour**
ACRs fetched from the CDN
 - Even after >70% of ACR requests are served from local disk cache

Transcoders optimize one asset at a time.

One optimized asset is cheap to render.

A world with millions of them is not.

Can a transcoder optimize an entire live world?

So here's the takeaway: publishing freezes the source, but not the optimization.

Whenever we build a better encoder or a better LOD algorithm, old content gets all the improvements with the next request. We improve 20 years of content without asking anyone to reauthor it.

And this runs at serious scale: about 27,000 texture ACRs and 24,000 mesh ACRs are generated every minute. Over 136 million ACRs are fetched from the CDN every hour.

But an asset transcoder has one hard limit: it sees one asset at a time. A world made up of millions of individually optimized assets is still far too expensive to render.

So the question is: can we apply the same idea to an entire world?



World transcoder

[time check 21:00]

And that's the second pillar - the world transcoder.
Let's scale from one asset to a whole world.

When optimizing individual assets isn't enough



- Roblox worlds are kitbashed from many small, reusable parts
- One cube may already be optimal; a million cubes are not
- Asset transcoders see each asset in isolation
- Composition enables optimizations that individual assets cannot
 - Merge geometry and materials
 - Remove hidden and overlapping surfaces
 - Generate aggregate LODs
- The optimization unit must expand - from one asset to a region of the live world



Roblox worlds are built from many small, reusable parts. One cube may already be optimal. A million cubes are not.

An asset transcoder sees each mesh in isolation. It doesn't know that these cubes form a building. It cannot see their relative importance, their scale, or the surfaces hidden between them.

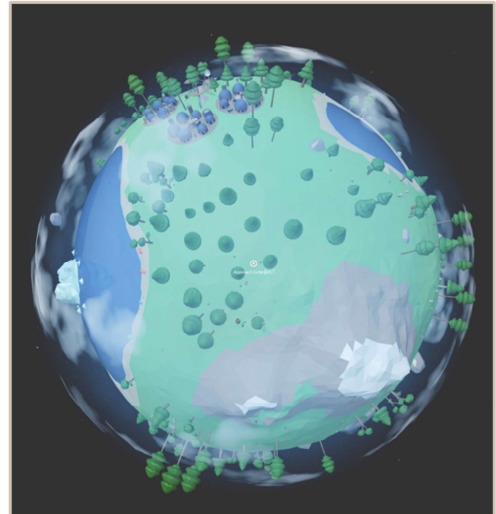
Those optimizations become possible only when you look at the whole composition. With that wider view, we can merge geometry and materials, remove hidden surfaces, and build LODs for the aggregate.

So the optimization unit must grow - from one asset to a whole world.

The same machinery can optimize a live world



- The transcoder contract is more general than assets
 - Stable, versioned inputs
 - Deterministic computation and outputs
 - Flexible execution: On-demand, local or remote
 - Content-addressed, reusable outputs
- Assets were only the first workload
- SLIM: Scalable Lightweight Interactive Models
 - Applies the same contract to selected parts of a live world
 - Derives multiple runtime fidelity levels
 - Keeps the live data model authoritative and editable



And here's the key: the transcoder contract was never really about assets. It needs a stable, versioned input and deterministic processing, and it produces output that is content-addressed, cached, and reused. Assets were just the first workload.

SLIM stands for Scalable Lightweight Interactive Models. It applies the same machinery to a selected region of a live world.

The source data model stays authoritative and editable. The world transcoder takes a rendering view of that region and derives lighter runtime forms, and clients use them exactly like any other ACR.

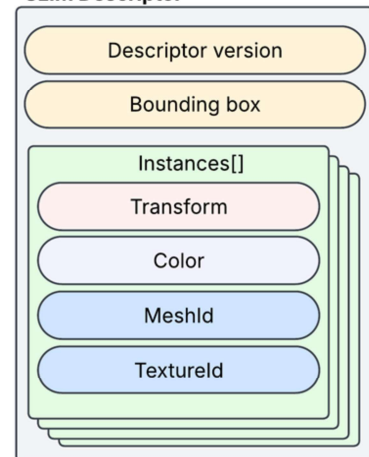
But first, we have to turn a live, mutable data model into a stable transcoder input.

A SLIM descriptor makes live world state transcodable



- The server determines the SLIM boundary
 - Today: the creator marks a data model subtree as SLIM-able
 - Future: spatial partitioning and other heuristics
- The server generates and uploads a versioned SLIM descriptor
 - Geometry and material references, transforms, colors
 - Asset payloads remain external - the descriptor stores references, not bytes
 - Updated incrementally as the live data model changes
- The SLIM descriptor is the world transcoder's input
- Clients consume the derived representation
 - The original data model subtree may never be replicated to the client

SLIM Descriptor



We call that input a SLIM descriptor. The server creates it because the server owns the full data model and sees the entire world.

Today, a creator marks a subtree of the data model as SLIM-able. Automatic boundary selection is a work in progress - we're exploring automatic segmentation and other ideas.

The descriptor stores only what the transcoder needs for rendering: geometry and material references, transforms, and colors. Asset payloads stay outside - the descriptor holds only references, never the actual bytes.

As the live world changes, the server updates the descriptor, and clients consume the derived SLIM output.

Here's a nice bonus: the client no longer needs the full data model just to render. Its data model view can shrink substantially, which saves memory, improves performance, and reduces network bandwidth.

For SLIM, the source itself is content-addressed



- Traditional source assets are addressed by asset ID + source version
- A SLIM descriptor's bytes define its identity
 - Canonical descriptor → content hash
 - The content hash becomes the source identifier
- The server computes the hash before uploading descriptor
 - Existing hash: reuse the descriptor and derived representations
 - New hash: upload once, then transcode and cache
- Identical structures anywhere resolve to the same source
 - Duplicate descriptors are detected before upload
 - Existing descriptor and derived representations are reused
- But this only works if equivalent models produce identical descriptors



SLIM source ID: #3812fd7a...

Now we have a compact, stable source, but it needs a stable identity.

Regular assets have an asset ID and a version. A SLIM descriptor is never explicitly published, so it has no ID. Instead, the content itself defines its identity: we hash the descriptor bytes, and that hash becomes the address.

If the hash already exists, we reuse the descriptor and everything derived from it. If it's new, the server uploads it once, and then the platform handles the transcoding and caches the results.

Here's the nice part: content addressing catches duplicates before upload. So the same model gets reused across different game sessions and even across different games built by completely different people.

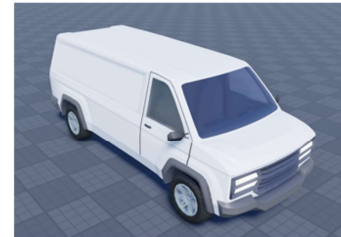
Normalization makes equivalent models byte-identical



- Equivalent models may differ in:
 - World transform
 - Child order and instance names
 - Harmless floating-point noise
 - Defaults and equivalent property encodings
- Before hashing, SLIM normalizes the descriptor:
 - Factor out parameterizable state: transforms, colors, materials
 - Quantize floats
 - Canonically order instances and properties
 - Remove all non-rendering data
- Different data model trees → identical descriptor bytes
- Identical bytes → the same SLIM descriptor hash
 - Reuse the descriptor and every derived representation



normalize



SLIM source ID: #3812fd7a...

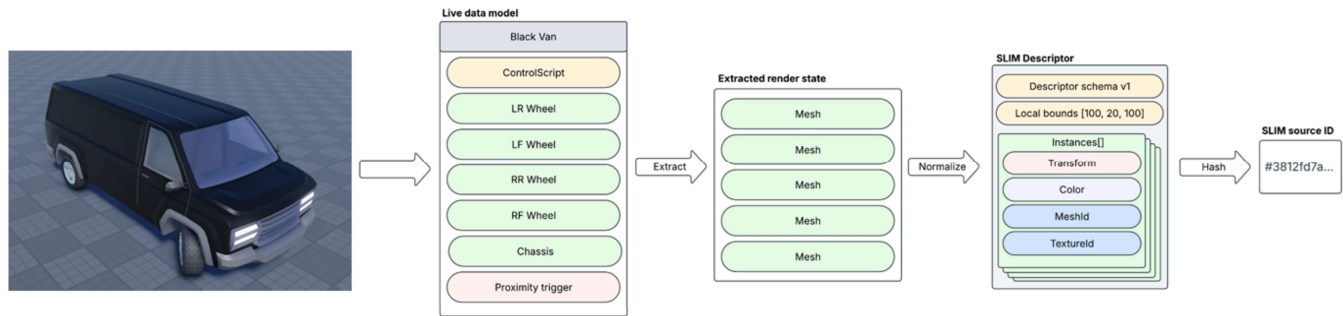
There's one important catch: models that look the same on screen must produce byte-for-byte identical descriptors. Otherwise, the whole deduplication system falls apart. That's why we normalize before hashing.

Two models can represent the same object using very different data model subtrees: children in a different order, different instance names, or floating-point noise in the transforms. If we hash the raw data, each of those becomes a different identity.

So we strip everything that doesn't matter for rendering. We factor out state that belongs in runtime parameters - transforms, colors, and materials. We quantize floats, sort instances and properties into a canonical order, and drop the rest.

After normalization, equivalent models produce the same bytes and the same hash, and that hash becomes the address.

Live Data Model → reusable SLIM source



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

Here's the full process for this van.

The server takes the live data model subtree, extracts the rendering state, normalizes the structure, factors out parameters, and removes everything that doesn't affect the image. The normalized bytes give us the SLIM address.

An equivalent van somewhere else - with a different name, transform, or history - resolves to the same address.

So we have turned a piece of a live world into a stable, reusable input. Now let's make it renderable.

From SLIM descriptor to renderable artifact



World transcoder

- Convert every part into renderable geometry in model space
- Group geometry by compatible material and shader state
 - Preserve texture IDs and material parameters
- Merge each group into one or more geometry chunks
 - 60 source meshes → 1 geometry chunk in this example
- Remove hidden interior surfaces
- Create several detail levels
 - Record byte size and geometric error for each level
- Package geometry, material mappings, texture IDs, and level metadata into SLIM ACRs

Client

- Select the best detail level for the current view
 - Visibility and screen coverage define "ideal LOD"
 - Memory and rendering budgets constrain the choice
- Fetch the selected SLIM ACR
- Fetch (or reuse) the texture ACRs per referenced texture ID
- For distant levels, assemble the 64 × 64 texture ACRs into an array
 - One shared runtime material per shader variant
 - Texture IDs select array slices
 - 13 source materials → 1 runtime material in this example
- At the lowest levels, replace textures with solid colors

So what does the world transcoder actually do?

It converts every part into renderable geometry in model space. It groups geometry by compatible material and shader state, merges it into larger chunks, removes hidden interior surfaces, and builds several levels of detail for the aggregate.

Each detail level records its byte size and geometric error.

The output is packaged as multiple SLIM ACRs - one per fidelity level. The transcoder doesn't target any specific device. At runtime, each device selects a fidelity level based on the camera frustum, screen size, and its own performance and memory budgets.

Up close, we render each material separately with its original textures. At a distance, we combine the materials into one runtime material. We put smaller versions of the textures into a texture array. This lets us render the whole van in one draw call. Even farther away, we replace the textures with solid colors.

For this van, 60 source meshes become one geometry chunk, and 13 source materials become one runtime material.

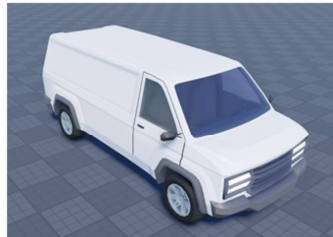
Let's walk through the pipeline step by step.

Example: from descriptor to SLIM ACR

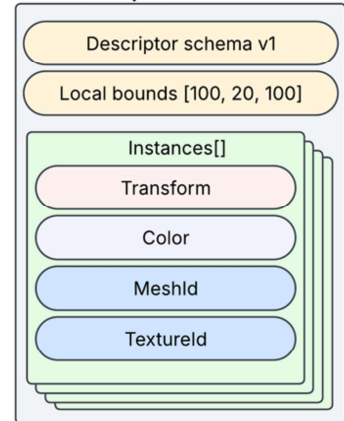


World transcoder

1. SLIM descriptor
- ↓
2. Fetch referenced assets
- ↓
3. Build aggregate mesh
- ↓
4. Remove hidden interior surfaces
- ↓
5. Simplify geometry
- ↓
6. SLIM ACR



SLIM Descriptor



We'll use this six-step diagram to follow the van through the process.

We have already covered step one - creating the descriptor - in detail. Just remember that it stays small: transforms, colors, and asset references, but never the payloads. That keeps every input explicit and reproducible.

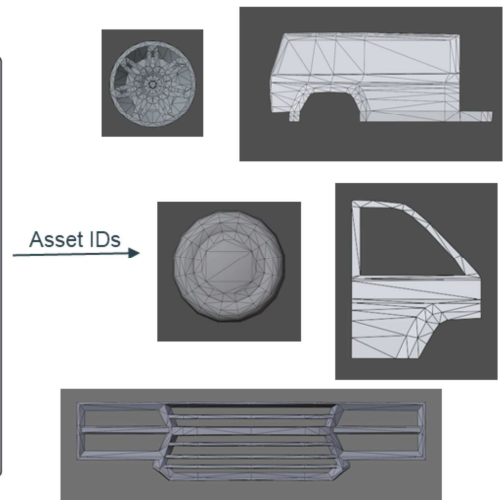
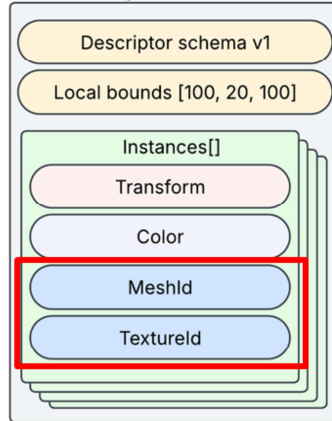
Example: from descriptor to SLIM ACR



World transcoder

1. SLIM descriptor
2. **Fetch referenced assets**
3. Build aggregate mesh
4. Remove hidden interior surfaces
5. Simplify geometry
6. SLIM ACR

SLIM Descriptor



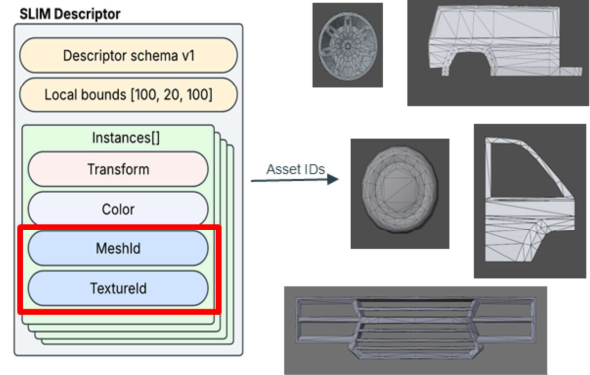
The next step is to fetch the actual asset data...

Example: Fetch referenced assets



The transcoder is sandboxed game-engine code - not a separate implementation

- Deserialize and validate the SLIM descriptor
- Walk through all instance records
- Collect unique asset IDs
- Resolve each asset ID to an exact asset version
- Check the shared local disk cache
- Fetch cache misses from CDN
 - Source assets or required ACR variants
- Parse the resolved assets and load them into memory



One thing worth calling out: the world transcoder is sandboxed game-engine code. It's not a separate reimplement of every Roblox format. It runs the same code as the runtime.

That code validates the descriptor, walks the instances, collects the unique asset IDs, and resolves each one to an exact source version. It checks the shared local disk cache first; anything missing comes from the CDN, either as a source asset or as an asset ACR. Then it loads everything into memory.

The descriptor is small, but every dependency is exact and explicit.

Example: from descriptor to SLIM ACR



World transcoder

1. SLIM descriptor
↓
2. Fetch referenced assets
↓
3. **Build aggregate mesh**
↓
4. Remove hidden interior surfaces
↓
5. Simplify geometry
↓
6. SLIM ACR



With the assets loaded, the next step is to build the aggregate...

Example: Build aggregate mesh

SIGGRAPH



- Bake position, rotation, and scale into model space
 - Allows simplification across the original object boundaries
- Create renderable geometry
 - Mesh assets, primitives, CSG, and skinned geometry
- Group and merge compatible geometry
 - Match material and shader state
 - Start a new batch only when the target ACR's mesh limits are reached
- Preserve existing motion
 - Combine existing skeletons into a shared skeleton
 - Preserve skin weights and bone influences
- Add control to rigid geometry
 - Create virtual bones and assign rigid skin weights
 - Selected subsets can move or animate without re-transcoding



First, we bake every part's position, rotation, and scale into a shared model space. That erases the original object boundaries and lets the mesh simplifier work across the whole composition.

Then we group and merge geometry with compatible material and shader state, starting a new batch only when platform limits force us to do so. For example, we start a new batch when we hit the 16-bit index buffer limit.

Also, note that aggregation does not make everything static. For skeletal meshes, we merge the input skeletons into one shared skeleton, preserving skin weights and bone influences. For static meshes, we add one virtual bone for each input mesh. So we can still rotate a wheel or open a door while using the aggregate.

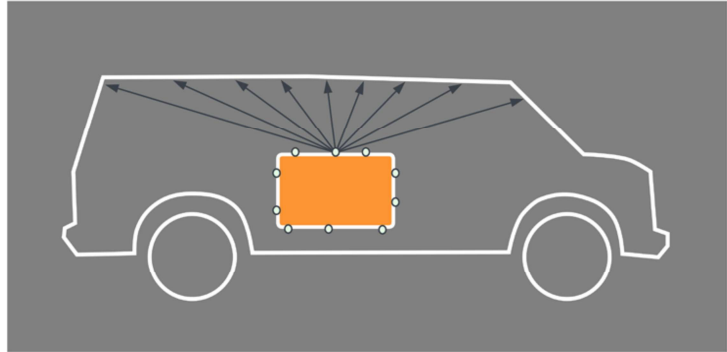
The rule of thumb is to optimize the structure while keeping the mapping to the live data model.

Example: from descriptor to SLIM ACR



World transcoder

1. SLIM descriptor
↓
2. Fetch referenced assets
↓
3. Build aggregate mesh
↓
4. **Remove hidden interior surfaces**
↓
5. Simplify geometry
↓
6. SLIM ACR

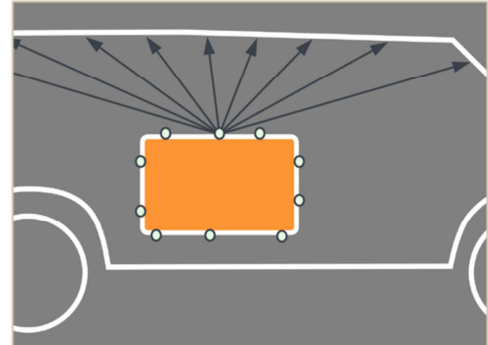


Next, we remove geometry that never contributes to the image.

Example: Remove hidden interior surfaces



- Build BVH over the aggregate
 - Accelerates ray tests against all other triangles
- Sample points across every face
 - Larger faces receive more samples
 - Always include the three vertices
- Search for a path to open space
 - Cast several directions and their opposites
 - Ignore the face currently being tested
- Remove the face only if no sample finds a path to open space
 - Any escape path marks the face as potentially visible and stops the test



The transcoder builds a BVH for the entire aggregate because it performs many visibility tests while transcoding.

For each triangle in the aggregate, we sample points across its area. Larger triangles get more sample points. From every sample point, we cast rays in several directions and check whether any of them escape to open space.

The rule is intentionally conservative: if any ray escapes, the triangle is potentially visible, and we keep it. We remove a triangle only when no sample finds a way out.

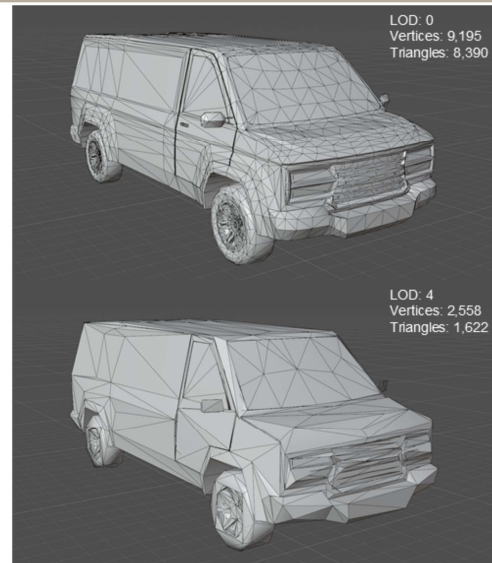
That eliminates interior and overlapping triangles without creating visible holes in the model.

Example: from descriptor to SLIM ACR



World transcoder

1. SLIM descriptor
↓
2. Fetch referenced assets
↓
3. Build aggregate mesh
↓
4. Remove hidden interior surfaces
↓
5. **Simplify geometry**
↓
6. SLIM ACR



After that, we simplify what remains.

Here's the van presented at two detail levels. The most detailed version has about 9,200 vertices and 8,400 triangles. At level four, that drops to about 2,600 vertices and 1,600 triangles.

Example: Simplify geometry

SIGGRAPH



- Prepare arbitrary topology
 - Weld coincident positions, but keep separate UV and normal values at seams
 - Store vertex→edges, vertex→faces, and edge→faces explicitly
 - Allow one, two, or many faces around an edge
 - Simplify disconnected components without joining them
- QEM that remembers the source but adapts to the current mesh
 - Accumulate geometry quadrics so removed source faces still influence later collapses
 - Rebuild area and boundary penalties from the current neighborhood
 - Increase collapse cost at sharp creases detected from adjacent face angles
 - Include UV and other attribute distortion in the collapse cost
- Collapse the lowest-cost valid edge
 - Solve for the best position of the merged vertex
 - After each collapse, update only adjacent candidates - not the whole mesh
- Preserve attribute seams
 - One geometric position may have several UV or normal values
 - Simplify attributes without welding seams

See: Liu, Zhang & Yuksel, *Simplifying Textured Triangle Meshes in the Wild*, 2025

Simplifying Textured Triangle Meshes in the Wild

HSUEH-TI DEREK LIU, Roblox, Canada
XIAOTING ZHANG, Roblox, USA
CEM YUKSEL, University of Utah & Roblox, USA

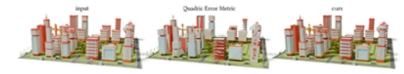


Fig. 1. Given an artist-created city scene (left) with high resolution, we simplify it down to 10% of its original resolution. Our method (right) better preserves geometrically significant components (e.g., buildings) compared to the quadric error simplification (left) and Hsiao et al. (2005) (middle), which prioritize small components (e.g., trees).



Fig. 2. Comparison between our textured mesh simplification method and a representative prior technique (left) and Hsiao et al. (2005) (middle) using the implementation from [Zhang et al. 2025]. Our single component and manifold inputs both methods produce excellent results (compare baselines, top row), as does top-right baseline for challenging “self” inputs, often characterized by non-manifold and multiple components geometry. The baseline approach frequently yields undesirable results (bottom center). In contrast, our method preserves visual fidelity and geometric structure more effectively on these self-meshes (bottom right).

ACM Reference Format:
Hsiao, T.-D., Zhang, X., and Yuksel, C. 2025. Simplifying Textured Triangle Meshes in the Wild. *ACM Trans. Graph.* 44, 4 (December 2025), 19 pages. <https://doi.org/10.1145/3694277>

INTRODUCTION
Mesh simplification reduces different levels of detail (LOD) for 3D objects while maintaining their visual fidelity. It plays a critical role in interactive graphics to achieve target performance in rendering and simulation across various hardware platforms. Its importance has been recognized for decades of development with a plethora of solutions, such as the widely used QEM by Garland and Heckbert [1997]. While these solutions produce high-quality results for manifold surface inputs, we demonstrate that they often fail to generate acceptable results on meshes obtained from online repositories (see

A quick word about mesh simplification: we had to develop our own algorithm because user-generated content rarely has clean, manifold topology.

The simplification algorithm explicitly stores the relationships between vertices, edges, and faces without using the traditional half-edge data structure. It handles edges with one, two, or many adjacent faces and keeps disconnected parts disconnected.

The quadric error metric preserves the original surface, and we added a penalty for area loss to improve the results. Sharp creases and distortion in UVs or other vertex attributes raise the edge-collapse cost. Each simplification step collapses the cheapest valid edge, solves for the new vertex position, and updates only the nearby candidates. At attribute seams, we keep multiple attribute values instead of welding them together.

The result is a set of useful detail levels that don't require clean topology - because we rarely get it.

We published a detailed paper on this algorithm last year.

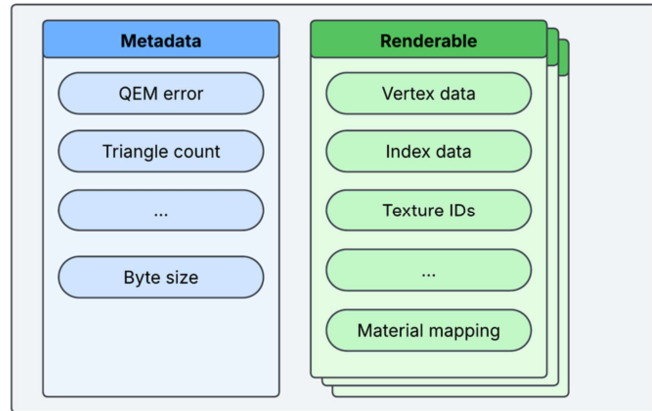
Example: from descriptor to SLIM ACR



World transcoder

1. SLIM descriptor
- ↓
2. Fetch referenced assets
- ↓
3. Build aggregate mesh
- ↓
4. Remove hidden interior surfaces
- ↓
5. Simplify geometry
- ↓
6. **SLIM ACR**

SLIM ACRs



The final step packages the detail levels into multiple SLIM ACRs.

Each one holds the renderable data - vertices, indices, texture IDs, and material mappings - plus the metadata the runtime needs to choose between levels: geometric error, triangle count, and byte size.



Runtime orchestration

That brings us to the third pillar: runtime orchestration.

The transcoder created the fidelity levels. Now each client decides how to spend its budgets on them.

Quality within memory and performance budgets



Maximize visible quality across the whole scene

Memory

- One fixed byte budget across all SLIM meshes
- Store each shared mesh only once
- Evict unneeded detail
- Load the detail levels that improve quality the most

Performance

- One fixed rendering budget shared across the scene
- Visibility and screen size set each instance's ideal LOD level
- Rendering cost limits the selected level
- Each instance gets no benefit above its ideal level

At runtime, the client solves two connected problems: performance and memory.

The performance budget controls how much detail each visible instance gets. Visibility and screen coverage define the ideal LOD, and the rendering budget pulls that target down when the scene becomes too expensive to render. Detail above the ideal level buys us nothing - the object simply doesn't cover enough pixels to show it.

The memory budget controls which LODs stay resident in GPU memory. There is one fixed byte budget across all SLIM meshes. Each shared mesh is stored once, unneeded levels are evicted, and missing but useful levels compete for the available memory.

Performance determines which detail is useful.

Memory determines which detail can remain resident.

Together, those limits determine the best visible quality we can achieve across the whole scene.

Inside the orchestrator: two budgets, one quality target



- Set the limits
 - Device capabilities and quality tier set both budgets
 - Scene complexity changes the competition - not the budgets
 - Rendering: triangles/frame target in this example
 - Desktop **4.78M** · Tablet **2.4M** · Phone **1.1M**
 - Memory: fixed GPU-byte pool for resident SLIM LODs
- Compute useful detail
 - Visibility and screen coverage set each instance's ideal LOD
 - Detail above that LOD provides no visible benefit
- Evict unnecessary detail
 - Choose a target LOD for each model
 - Evict resident LODs above the new target first
- Spend the recovered memory
 - Rank all below-ideal LODs by score
 - Load the highest-scoring upgrade that fits

$$Score_{a,r} = \frac{\sum_{i \in I_a} v_i c_i \Delta Q_{i,r}}{\Delta M_{a,r}}$$

Score : upgrade score
a : shared mesh
r : candidate LOD
I_a : instances using mesh *a*
v_i : visibility of instance *i*
c_i : screen coverage of instance *i*
 $\Delta Q_{i,r}$: QEM error removed for instance *i*
 $\Delta M_{a,r}$: added GPU bytes

Higher score = more visible error removed per added GPU byte

Inspired by Shoham & Gersho (1988)

Let's make that precise.

Device capability and quality settings establish two fixed budgets. Here, the rendering budget is a triangle target - 4.7 million on desktop, 2.4 million on tablet, and 1.1 million on phone. The memory budget is a fixed pool of GPU bytes. Scene complexity changes the competition for these budgets, not the budgets themselves.

Every frame, visibility and screen coverage set the ideal LOD for each instance, and the rendering budget sets the actual target.

For memory, we first evict the levels above their new targets. Then we score every useful, missing level by the visible error it removes per added byte, and we load the best-scoring upgrade that fits into the available memory.

If that sounds familiar, it's because this is rate-distortion optimization. We spend bytes where they remove the most visible error.

One world, scaled to each device budget



Metric	Baseline	Desktop		Tablet		Phone	
Client instances	172,531	16,582	-90%	13,750	-92%	5,213	-97%
Triangles / frame	59.4M	4.78M	-92%	2.4M	-96%	1.1M	-98%
Draw calls / frame	3,526	2,432	-31%	1,535	-56%	917	-74%
GPU memory (MB)	204	186	-9%	169	-17%	137	-33%

*Same scene and camera, real-time engine capture

Back to the demo. This table compares the same scene and camera across several devices - one device per column.

Compared with the naive baseline, we cut up to 97 percent of client instances, 98 percent of triangles, and 74 percent of draw calls, while reducing GPU memory by 33 percent.

The takeaway is that every number in this table comes from the same source world but reflects different devices and device budgets. There is no handmade version for desktop, tablet, or phone. Everything is generated on demand, and each device selects the best representation possible.

One world, scaled to each device budget



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

This is the naive baseline - 59.4 million triangles per frame.

One world, scaled to each device budget



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

And this is the desktop with SLIM enabled - nearly identical visuals at 4.7 million triangles. It's the same scene and the same camera.

[switch back and forth]

As you can see, the difference is visible but small. Most of the work we eliminated had almost no effect on the final image.

Let's check the debug view and see where the runtime spent the detail.

One world, scaled to each device budget



Least detailed LOD



Most detailed LOD

Desktop - 4.78M triangles



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

Uncolored objects use the regular rendering path. The colors show the SLIM detail level selected for each object.

Notice that the selected detail levels don't form simple distance rings. Nearby objects get less detail when they cover only a small part of the screen or when their geometric error is small. Farther objects can get more detail, especially when one resident mesh serves many visible instances at once.

Visibility, screen coverage, reuse, triangle cost, byte cost, and the global budgets - all of it feeds the orchestrator.

One world, scaled to each device budget



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

Now let's look at the tablet target - 2.4 million triangles.

The runtime removes detail where the visual loss is smallest. So the scene keeps its silhouette and density, even at roughly half the desktop budget.

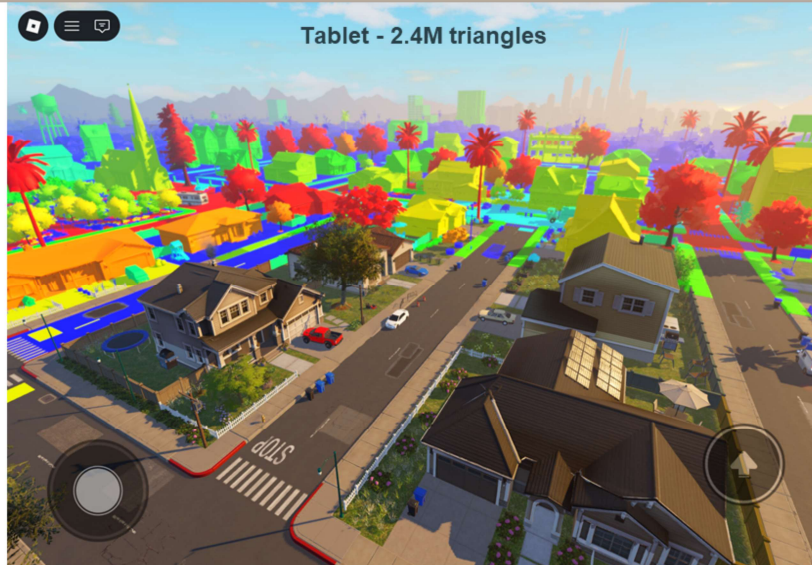
One world, scaled to each device budget



Least detailed LOD



Most detailed LOD



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

In the debug view, we can see that the LOD selection is more aggressive and that more objects render through SLIM.

But the decision is still global and value-driven - we didn't flip one big quality switch. Every resident level still competes on visible benefit per byte.

Let's drop the budget one more time.

One world, scaled to each device budget



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

And this is the phone target - 1.1 million triangles.

I never authored this demo to be mobile-friendly. Honestly, I ran it on a phone exactly once - to capture this screenshot. The platform derived the optimized form automatically, and the phone picked the version that fit its budget.

This is the same scene you saw at the start of the talk.

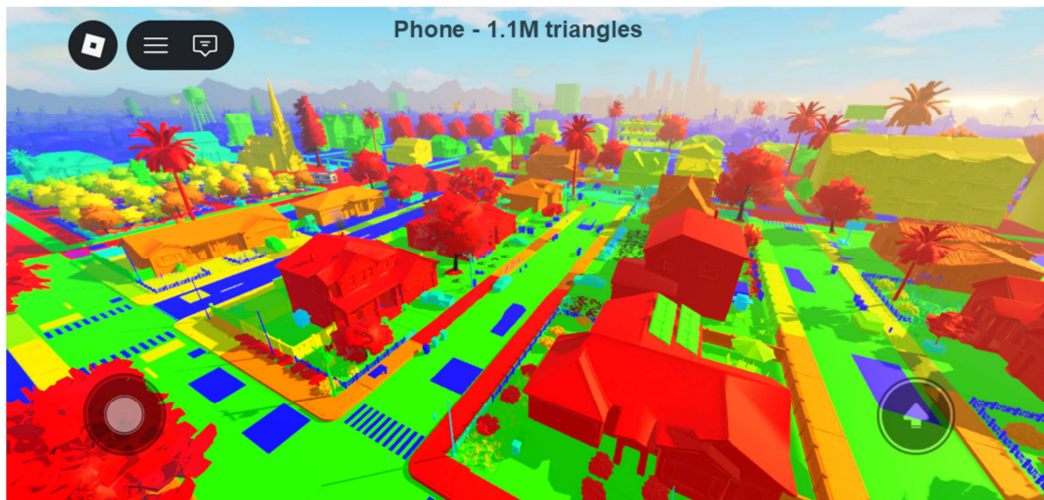
One world, scaled to each device budget



Least detailed LOD



Most detailed LOD



© 2020 SIGGRAPH Advances in Real-Time Rendering in Games course. All Rights Reserved.

ROBLOX

And here's the allocation behind that phone image. Almost everything is SLIM now, and the LOD selection is even more aggressive.

I know where to look, so I can see what was removed. But can you really tell the difference?

We have one live world, one source, many derived forms, and three different devices, each selecting what's optimal for that exact frame.



Dynamic SLIM

So far, everything has looked mostly static. But Roblox worlds move, animate, and change.

That brings us to dynamic SLIM.

Immutable artifact, dynamic scene?



Server - live data model

- Authoritative instances and properties
- Position, rotation, scale
- Color and animation state
- Changes continuously

Network replication

- Parameter mapping
- State snapshots
- Root-bone transform updates

Client - immutable SLIM artifact

- Aggregate geometry
- Detail levels
- Material mapping
- Parameter slots

The artifact stays immutable - the client renders it with the latest replicated state

And here's the split.

The server owns the live data model - instances, properties, transforms, colors, animation state, and the scripts that drive them. The client holds an immutable SLIM artifact - aggregate geometry, detail levels, material mappings, and parameter slots.

Do you remember the virtual bones we inserted into the aggregate and the mapping back to the data model? That's exactly what lets an immutable artifact react to transform and color changes.

The server takes compact snapshots of those parameters and replicates them to clients, and the renderer applies them to the SLIM models. Colors keep changing, and the wheels keep spinning, even though all the geometry is baked into one aggregate and rendered in a single draw call.

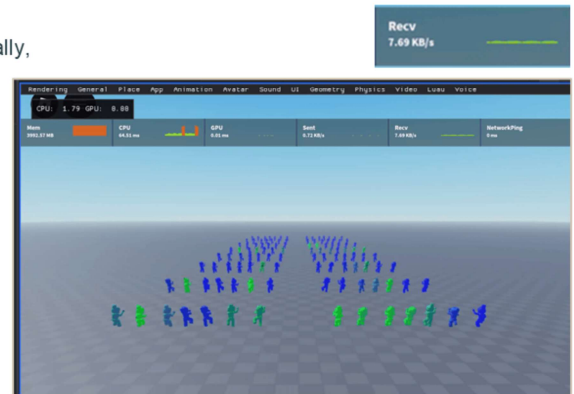
The client always renders the same artifact, parameterized with the latest state replicated from the server. The client has an immutable structure driven by a live stream from the server.

The latest visual state matters - not every update



The artifact already holds the structure; only prioritized parameter snapshots cross the network

- **Filter**
 - No network update when simulation state is already available locally, the model is invisible, or changes stay below their thresholds
- **Prioritize**
 - Age + proximity determine urgency
- **Budget + send**
 - Leaky bucket, compact quantized snapshots, unreliable delivery
- **Smooth**
 - Quintic inertialization, latest snapshot wins



Inertialization inspired by David Bollo (GDC 2018)

Snapshot throttling in action - the green models are receiving updates

Since the SLIM artifact already contains the structure and the mapping, we don't send every property update to every client.

If the client already has the simulation state available, we use it directly to parameterize the SLIM model - at zero network cost.

For models the client doesn't have, the server must replicate the state. We use importance-based prioritization, and we throttle updates to stay within a fixed network-bandwidth budget. And yes, network bandwidth is budget-based too, like everything else in this system.

We also track the time since the last update, so distant models don't freeze in the same pose forever. Nearby models update more often, but eventually, every SLIM model gets its update.

Network delivery is intentionally unreliable, and it's very simple: there is no retransmit queue and there are no acknowledgments. A newer snapshot makes every older one obsolete anyway.

On the receiving end, snapshots arrive sparsely, so the client smooths them with quintic inertialization, turning sparse network packets back into smooth, render-rate motion.



Here's another example: 1,200 live, animated Roblox avatars.

[play video]

Can you tell where SLIM is in this video?

The server stays authoritative the whole time. The client renders lightweight artifacts and receives compact snapshots for the motion that matters on screen.

SLIM works best when structure is stable and reusable



Artifact churn, one-off content, strict latency requirements, and over-aggregation can reduce the savings

- **Frequent invalidations**
 - Structural changes invalidate the artifact
 - Repeated transcoding and cache churn may cost more than SLIM saves
- **Unique content**
 - One-off geometry gets no benefit from hash deduplication
 - LOD and streaming may still help, but reuse disappears
- **Low-latency simulation**
 - Highly dynamic state may need immediate local updates
 - Remote snapshots may not meet the latency requirement
 - Use local simulation state when available
- **Over-aggregation**
 - Many small objects create large, spatially sparse aggregates
 - Many materials increase texture-array memory pressure
 - Frustum and occlusion culling operate at aggregate granularity
 - One visible piece may cause the entire aggregate to render

SLIM is extremely efficient at optimizing aggregates, but let's be honest about where it can be improved.

SLIM pays off when structure stays stable long enough or is reused enough to cover the transcoding and storage costs. Frequent structural changes create new descriptors and flood the system with repeated jobs. In that case, we prefer to fall back to traditional rendering.

Unique geometry that is used only once still benefits from LOD and streaming, but deduplication gives us almost nothing, and the memory cost grows quickly. That can push memory use over budget. To stay within the limit, the orchestrator must evict high-resolution LODs even when we need them. This causes a severe drop in quality.

Latency is another hard problem. Even a correct remote snapshot can arrive too late for a highly dynamic simulation. For extremely latency-sensitive cases, SLIM may not be the best fit yet.

And aggregation itself has a cost: large but spatially sparse models make frustum and occlusion culling less efficient.

So the contract is explicit: use SLIM where stable, reusable structure makes the trade worthwhile, and fall back when it doesn't.

Fallback is not a failure mode. It's what keeps the optimization correct.



- **Preserve meaning; derive representation**
 - Keep the source and its semantics authoritative
 - Re-transcode instead of re-authoring
- **Optimize with context and explicit contracts**
 - Normalize, discover reuse, and aggregate only when it helps
 - Fall back when a contract no longer holds
- **Let runtime budgets choose the detail**
 - Allocate memory, rendering cost, and bandwidth across the whole scene
 - Spend resources where they improve visible quality
- **Separate immutable structure from live state**
 - Artifacts contain geometry, materials, and parameter slots
 - Compact state updates keep them dynamic

Let me summarize what we have learned while building SLIM.

First, preserve the source of truth and derive the representation. The source stays authoritative, while optimized forms are disposable. We can improve the algorithms and transcode the content again later instead of asking creators to reauthor 20 years of content.

Second, optimize with context and explicit contracts. Normalize, discover reuse automatically, aggregate only while the assumptions hold, and treat fallback as part of the contract.

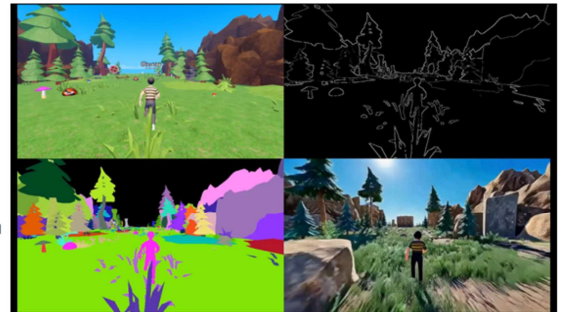
Third, let runtime budgets determine the level of detail. The transcoder creates options; each client spends its memory, rendering, and bandwidth budgets only where they buy the most visible quality.

And finally, separate immutable structure from live state. Geometry and parameter slots live in the immutable artifact. Compact, dynamic server snapshots keep it all moving.

Together, these ideas let a user-generated platform scale from desktop to phone with no extra work from creators.



- **Better SLIMs**
 - Richer parameterization and partial invalidation → fewer retranscodes
 - Non-triangle representations for distant content
 - Hierarchical, spatial SLIMs for very large worlds
- **Fully distributed engine**
 - Run any rendering work anywhere - client, server, rack, or cloud
 - Cache and reuse derived lighting, occlusion, visibility, etc.
- **Gen AI upsampling**
 - Transcoders can enhance content - not only simplify it
 - Reconstruct or enrich geometry, textures, materials, lighting, and animation
- **Roblox Reality**
 - World models need persistent spatial and semantic memory
 - Combine live data model state with compact SLIM descriptors



Roblox Reality - hybrid engine architecture

I'd also like to highlight some future work.

First, we want better SLIMs: richer parameterization and partial invalidation to survive small structural changes without retranscoding; representations beyond triangles for distant SLIMs, including point clouds, Gaussian splats, and impostors; and hierarchical, spatial SLIMs - think SLIMs made of SLIMs - to support even larger worlds.

Second, we want a fully distributed rendering engine. We want rendering tasks to run anywhere - on the client, a game server, a server rack, or in the cloud - using the same contract and producing outputs that can be cached and reused. Lighting, occlusion, visibility - you name it. For example, we could generate an ACR for global illumination.

Third, we want content upsampling. A transcoder doesn't have to simplify; it can enhance. Geometry, textures, materials, lighting, animation - all of it can be improved with generative AI.

And finally, Roblox Reality - a hybrid rendering architecture. AI world models need persistent spatial and semantic memory, and we already have the data model for that. Imagine a SLIM ACR that stores the output from a vision model or represents the world in latent space. It would be cacheable and reusable, so the world model wouldn't repeat that work over and over again.

To be clear, these are research directions, not shipping features. But they all share the same foundation: a stable source, movable computation, and reusable derived results.



Thank you!

smakeev@roblox.com

And that's all I have.

Thank you very much. I'm happy to take your questions.