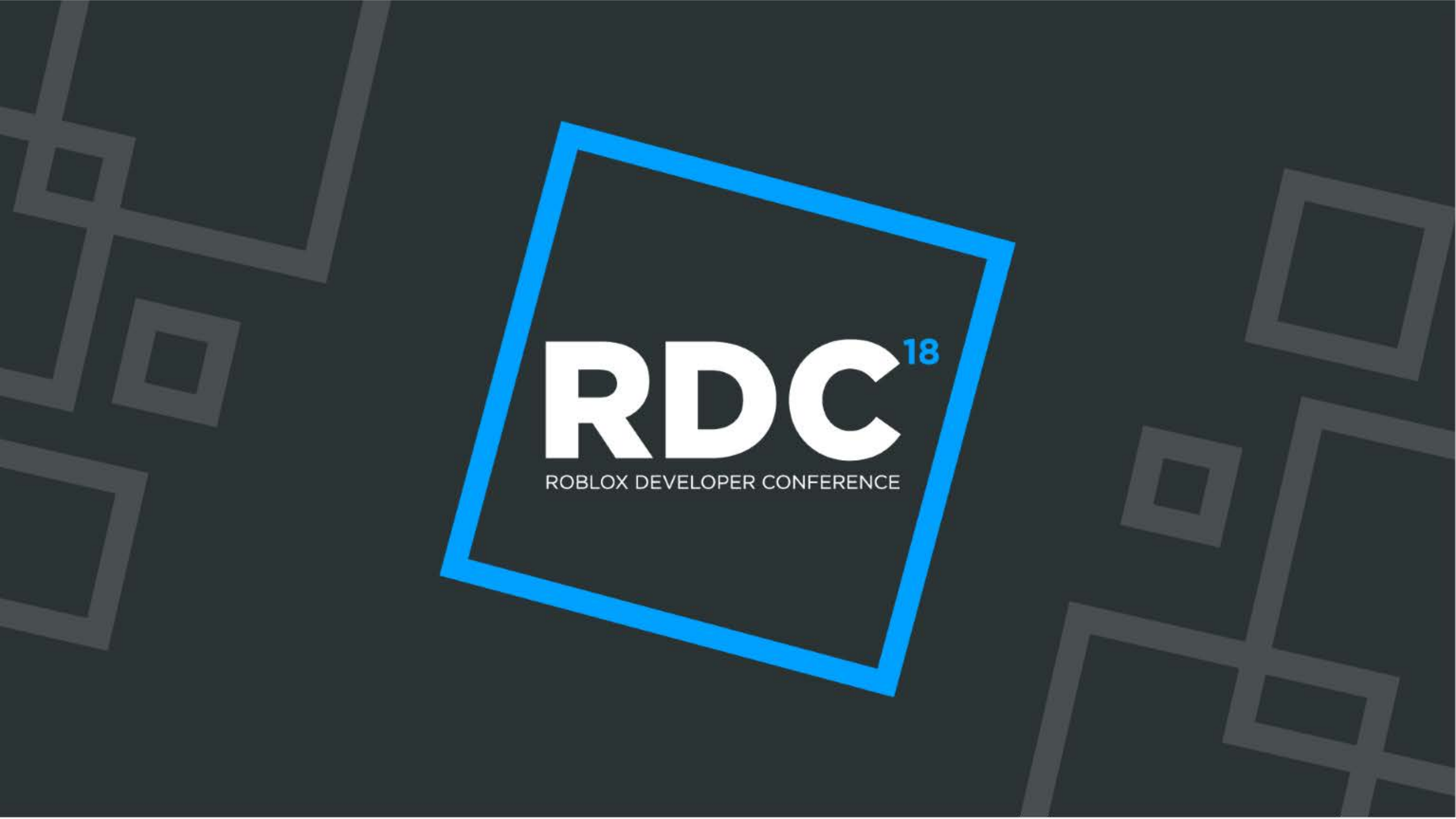

RDC 18
ROBLOX DEVELOPER CONFERENCE

Designing for Performance

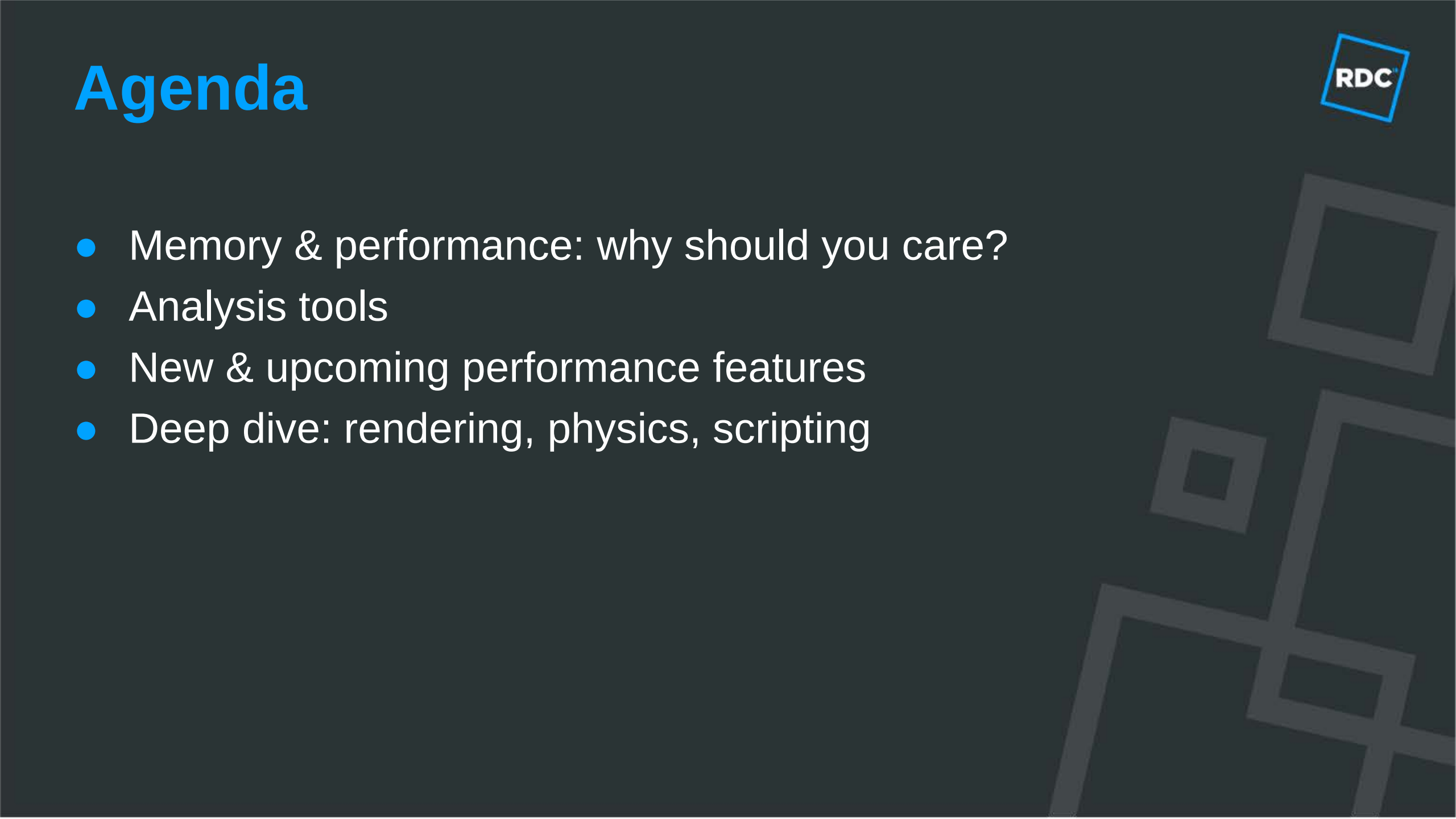
Arseny Kapoulkine

Who I Am

Arseny Kapoulkine

zeuxcg

Agenda



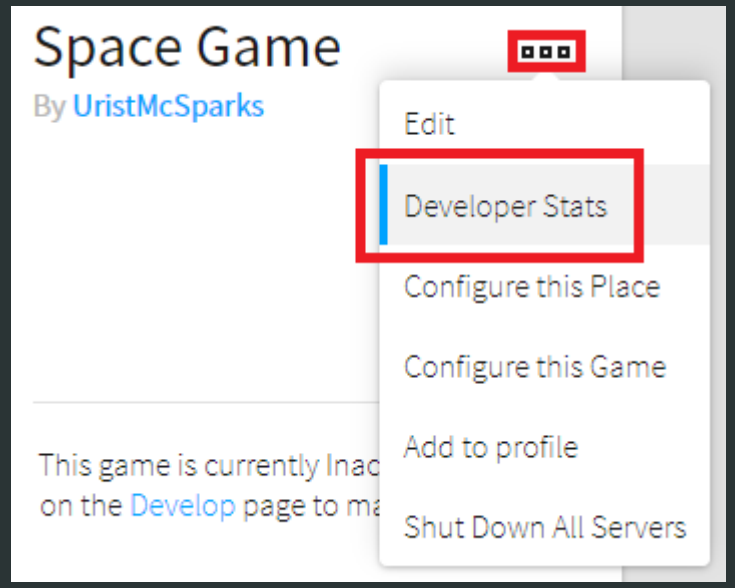
- Memory & performance: why should you care?
- Analysis tools
- New & upcoming performance features
- Deep dive: rendering, physics, scripting

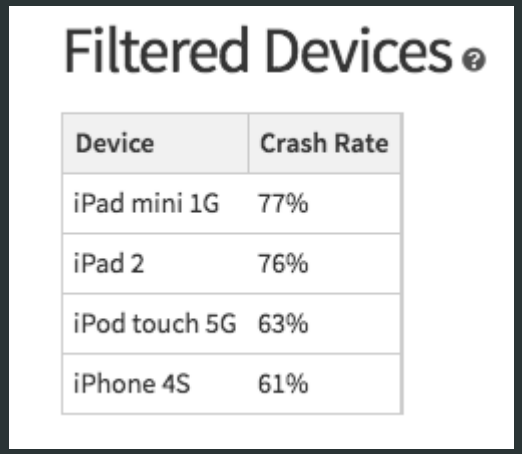
Memory & performance: Memory

- Number of bytes used by your game to store game state
- Includes EVERYTHING: objects, textures, Lua tables…
- Measured in megabytes (MB)
- Allowance: hundreds of MB on mobile

Memory & performance: Memory

- If you’re out of memory, the game crashes
- OOM is the largest source of crashes on mobile!



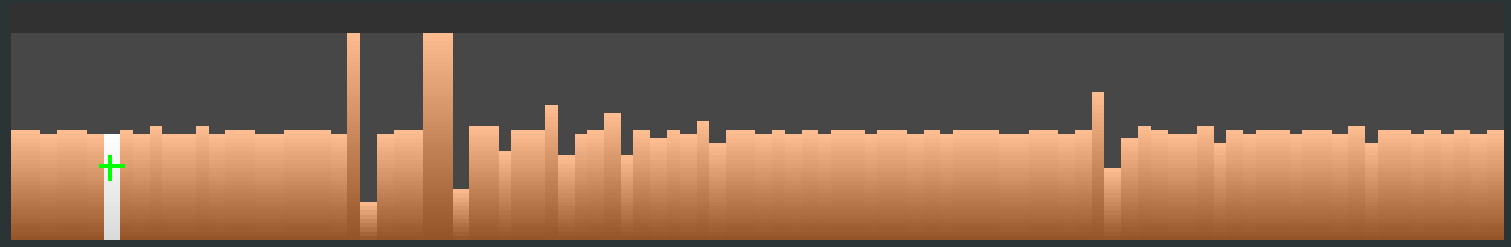


Memory & performance: Join time

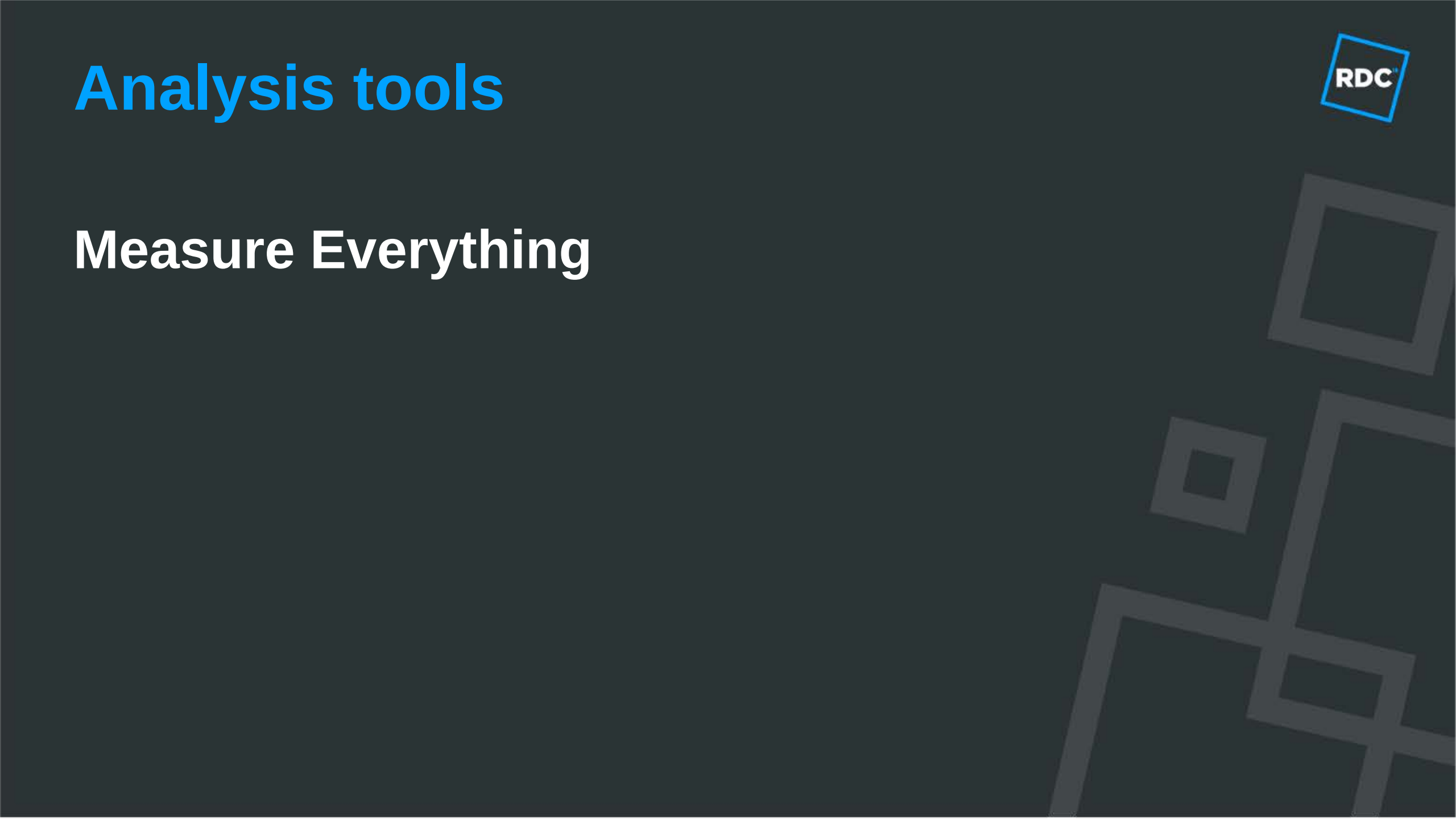
- How much time does it take a player to join your game
- Measured in seconds
- "Allowance": 10-20 seconds?
  - Median desktop: 4 seconds
  - Median mobile: 5 seconds

Memory & performance: Frame time

- How much time does one frame of your game take
- Measured in milliseconds
- "Allowance": 15-30 ms
- Beware spikes!



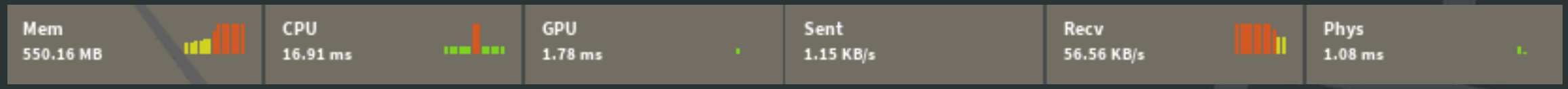
Analysis tools



Measure Everything

Analysis tools: Performance Stats

- Memory, frame time, network health (soon: ping time)
- At-a-glance “are things okay?” view
- Desktop: Ctrl+F7
- Mobile: Settings -> Performance Stats



Analysis tools: Client/Server memory

- Memory broken down by categories
- Available on client and server (for game owners)
- Caveat: categories vary per platform, measure on device!

Analysis tools: Microprofiler

- Precise frame timing measurement in a timeline view
- Internal engine work + Lua scripts (debug.profilebegin/end)
- Desktop: Ctrl+F6 / Ctrl+P
- Mobile: Settings -> Micro Profiler

Analysis tools: Stopwatch

- The best tool to measure & understand join time :(
- Load time is simple(r): a function of # of instances/voxels
- Watch out for server delays!

New features

Imagine Faster

New features: Overview

- Memory/performance is a shared responsibility
  - You, as a game creator
  - Us, as a platform creator
- We try to work on performance so that you don’t have to
- But you have to drive it by giving us feedback

New features: Instancing

- Old part clustering system: “Featherweight” parts (2012)
  - Dynamically merge all objects in a region
  - 10 MeshParts = 10x memory cost
  - Updating any non-CFrame property triggers geometry rebuild
- New part clustering system: instancing (late 2017)
  - Rebuilt around capabilities of “modern” GPUs
  - 10 MeshParts = 1x + ε memory cost
  - Updating any property is fast*

New features: Instancing

- Automatically enabled when HW supports it
  - 100% Xbox, 95% Windows/macOS, 90% iOS, 20%* Android
- Already live for CSG & MeshParts
- Will soon be live for other part types
- Usage guidelines
  - Disable Lighting.Outlines
  - Use Humanoids sparingly

Upcoming features: Level of detail

- Short term (2018)
  - Automatic Mesh/CSG part simplification
  - On low-end devices: use lower-poly mesh
  - On high-end devices: use lower-poly mesh at a distance
- Long term (soon™)
  - Model level-of-detail
  - Streaming level-of-detail

Upcoming features: Texture transcoding

- 1024x1024 texture
  - 32bpp + mipmaps => 5.3 MB
  - PNG size is misleading!
- We automatically encode textures into HW specific formats
  - No alpha: 4bpp + mipmaps => 0.7 MB
  - Alpha: 8bpp + mipmaps => 1.3 MB
- Expected quality degradation: check on device
  - We will support correct preview in Studio emulator

Upcoming features: UI caching

- Old UI rendering
  - For each visible container (SurfaceGui/ScreenGui)…
  - For each descendant Instance…
  - Generate geometry to render…
  - … and send geometry to GPU
- New UI rendering
  - … send geometry to GPU
- Put related UI elements into separate ScreenGUI objects!

Deep dive

Dig

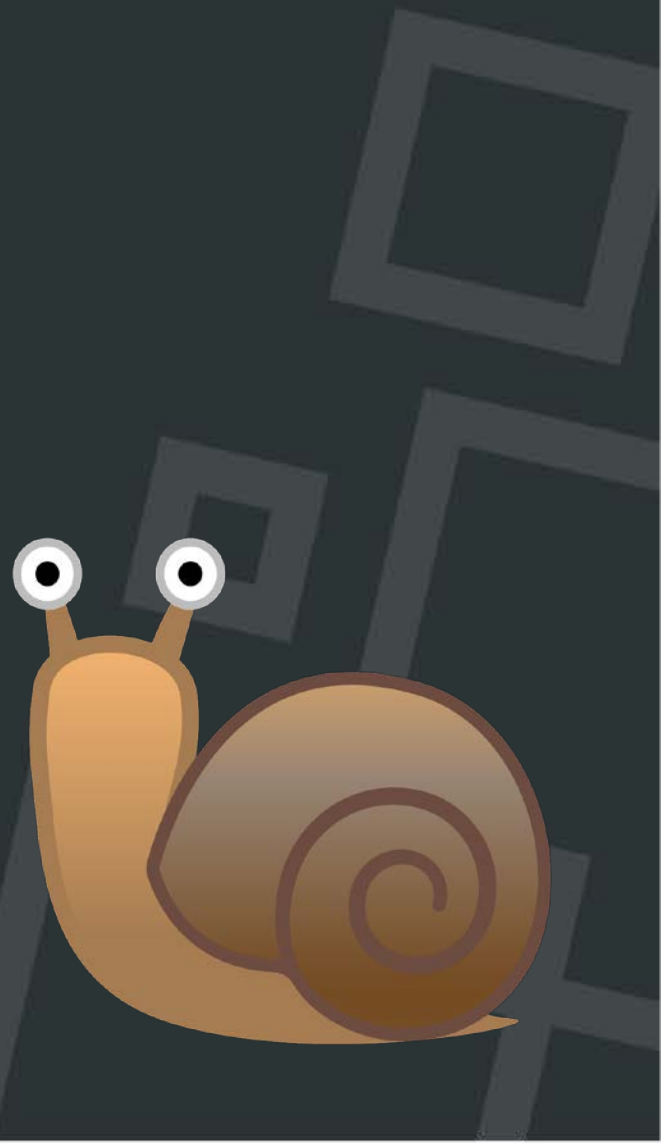
DEEPER

Scripting in depth

- When do you run Lua code?
- Are you using Lua efficiently?
- Are you using Roblox engine efficiently?

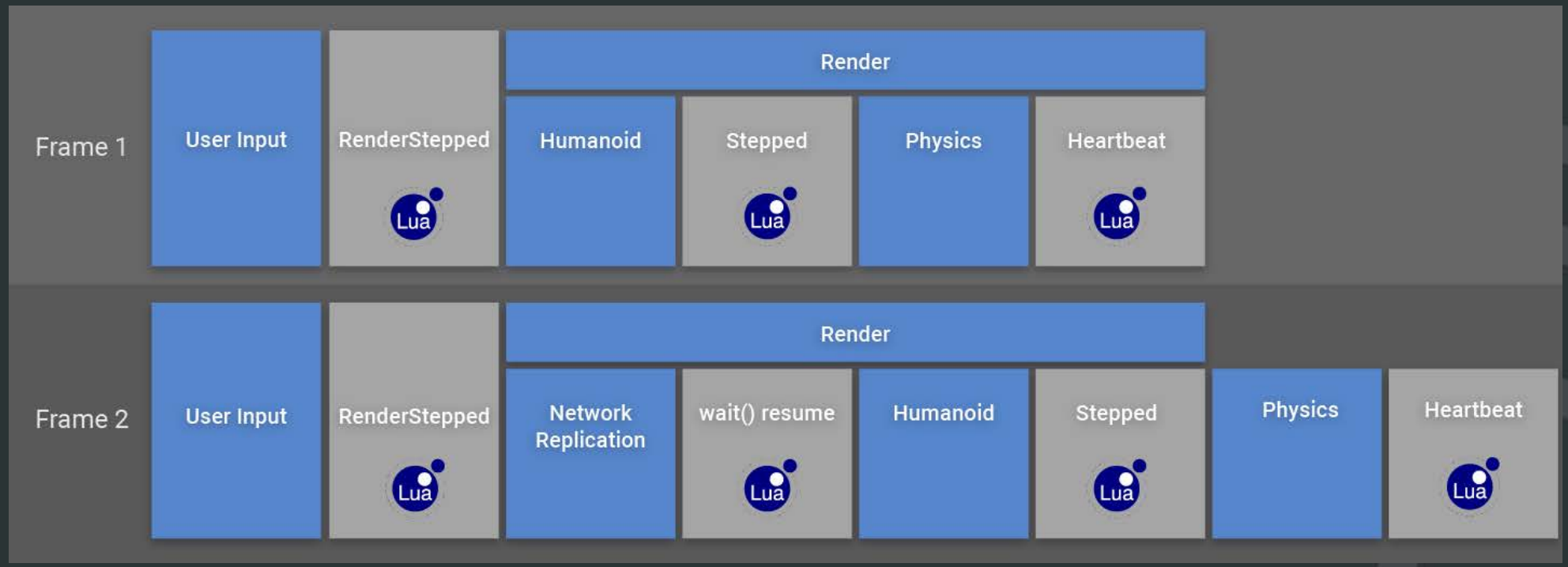
Scripting in depth: When?

- Do you NEED to run code every frame?
- Prefer running code infrequently
  - E.g. wait() loops with long delay (1+ seconds?)
- Prefer event-based code
  - Listen to user input, Touched, etc.
- Avoid infrequent but long running code
  - Use wait() to time-slice



Scripting in depth: Heartbeat et.al.

- Heartbeat vs RenderStepped vs Stepped?
  - All of these run every frame!

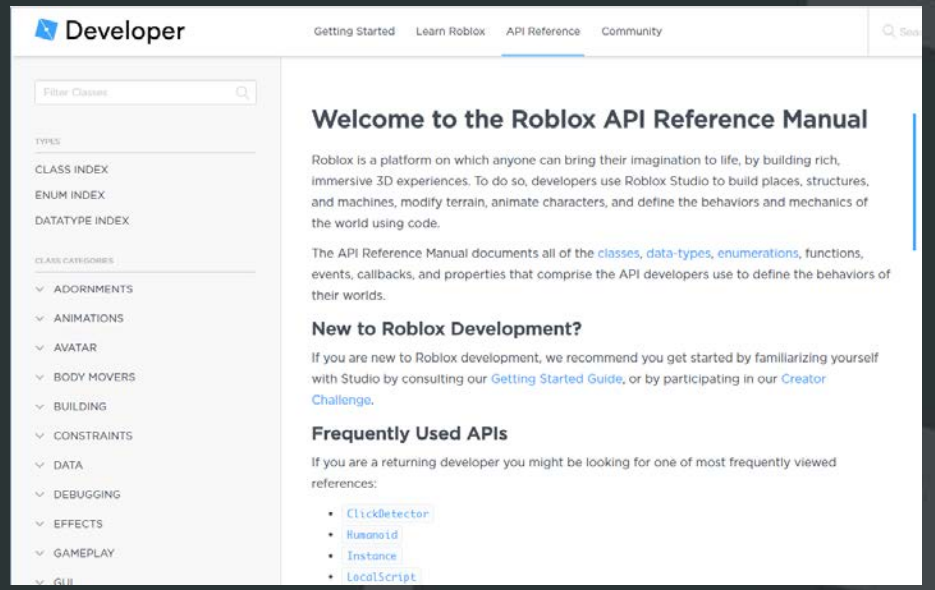


Scripting in depth: Lua

- Not all Lua code is equally fast
- Beware of:
  - Creating objects
  - Accessing properties too frequently
  - Calling methods too frequently
- General Lua performance advice applies!
  - E.g. “Lua Performance Tips” by R. Ierusalimschy

Scripting in depth: Roblox APIs

- Instance.new
  - Don’t pass “parent” argument to Instance.new
  - Don’t use table-based wrappers for Instance.new
- Custom animation engines
- UI Layout objects vs roll-your-own
- Help us help you!



Building in depth: What do I use?

- Terrain
- Basic parts
- CSG
- MeshParts

Building in depth: Terrain

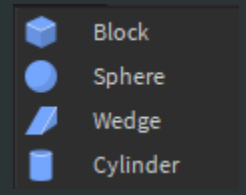
- Peak efficiency per unit of detail
  - Automatic level of detail
  - Minimum memory cost per voxel
  - Fast dynamic updates
- Stable guaranteed performance
- Make sure you need detail!
  - Hard to compete with 1 giant part

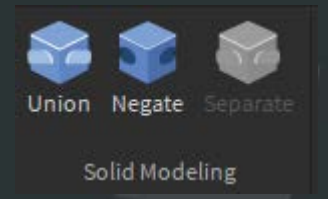
Building in depth: Terrain guidelines

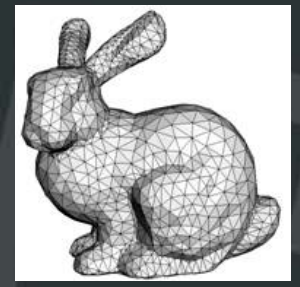
- Do:
  - Use different materials – they cost the same (exc. water)
  - Use varied terrain shapes
  - Use StreamingEnabled* (future: streaming LOD)
- Don’t:
  - Blindly use terrain if your map is enormous
    - 20k stud radius is pushing it
  - Use 1-voxel thick shell for mountains
    - Measure!
    - Need around 4 voxels to maintain LOD integrity

Building in depth: Part types

- Basic parts
  - Low efficiency per unit of detail
  - Can manipulate independently!
  - Low-end mobile: target tens of thousands of parts
    - … or use Streaming
- CSG parts
  - Medium efficiency per unit of detail
- MeshParts
  - Good efficiency per unit of detail
  - Must know how to Blender

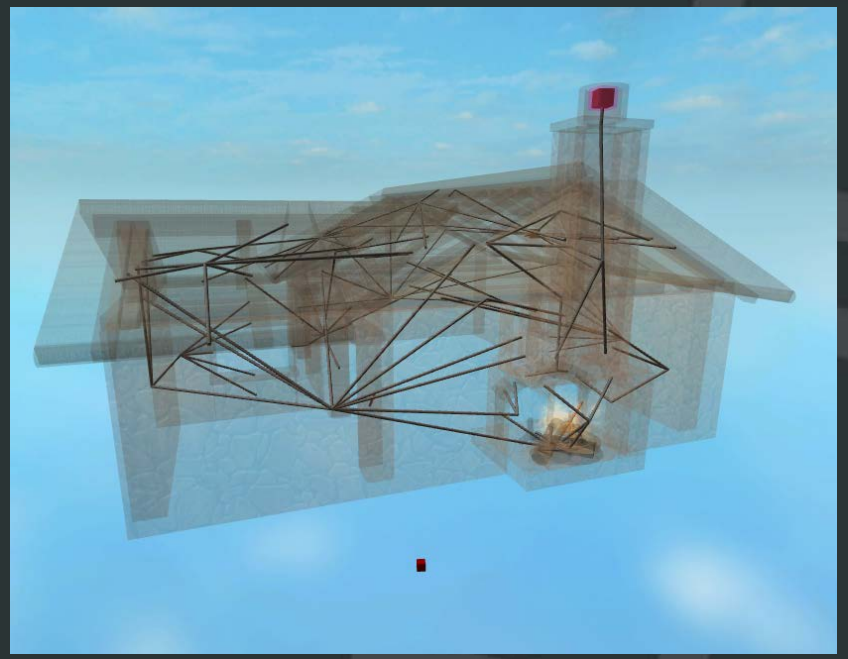






Building in depth: Physics

- Basic parts
  - Peak physics performance for single parts
  - Large welded clumps are not optimal
- CSG & MeshParts
  - Physics performance is identical!
  - Use CollisionFidelity
    - Box – no collision, or object looks like a block
    - Hull – simplified collision, convex
    - Default – anything else
- CanCollide=False *only* helps for interactive regions



Building in depth: Rendering

- Basic parts
  - Prefer Roblox materials for optimal performance
- CSG parts
  - Prefer Roblox materials for optimal performance
  - Use reasonably complex CSG parts (dozens of parts in a union)
- MeshParts
  - Use Roblox materials or MeshPart.TextureId
  - Use reasonably complex meshes
    - 1 200-triangle MeshPart is faster than…
    - 1 1000-triangle MeshPart is faster than…
    - 10 100-triangle MeshParts

Building in depth: Transparency

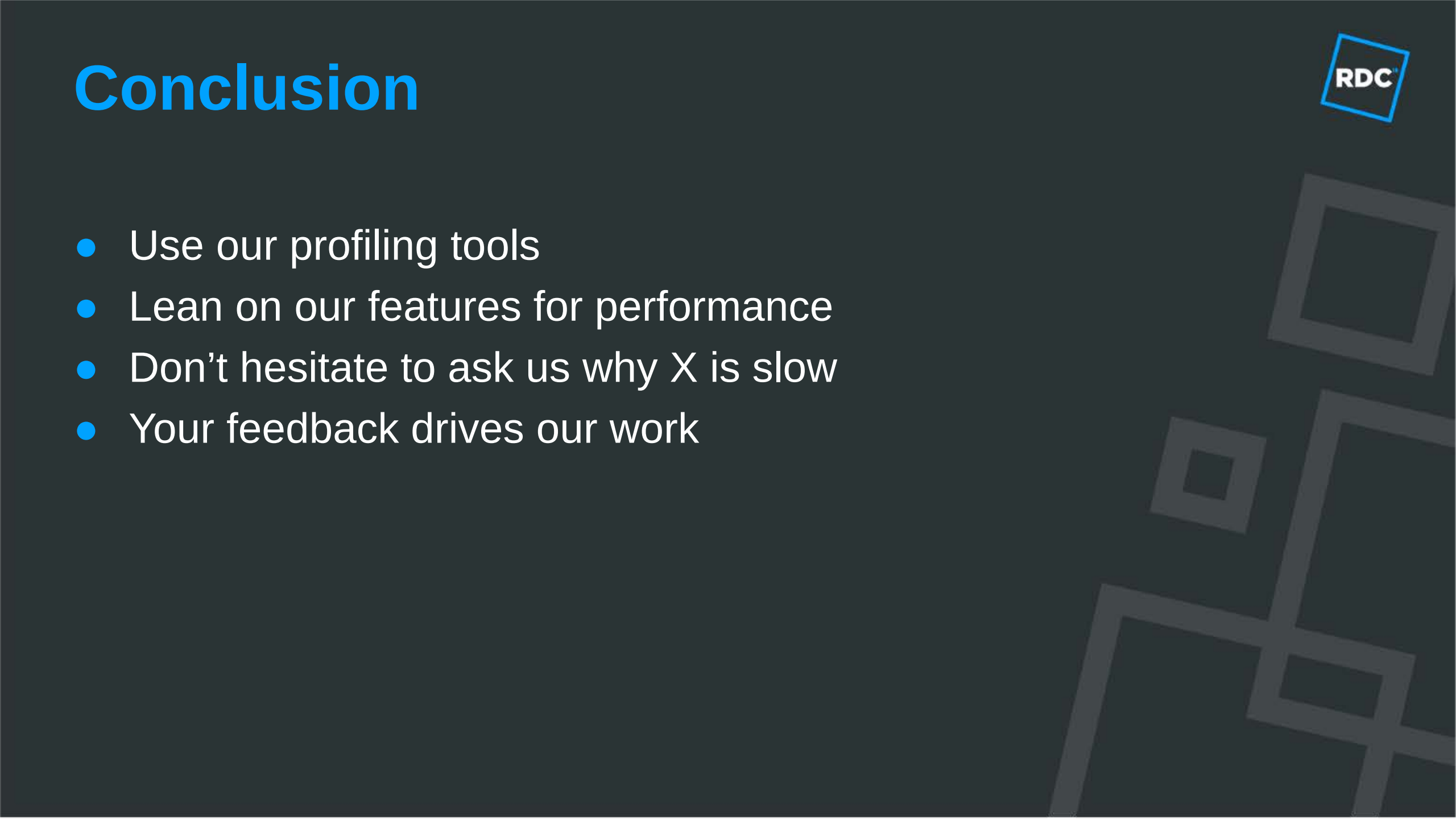
- Transparency is bad for performance
  - Unless .Transparency = 1 ;)
  - Merge transparent parts with CSG
- Use Textures & Decals sparingly
  - Prefer Roblox materials for performance
  - Use MeshPart.TextureId
- Part.Transparency = 1 & Decals/Textures
  - Please don’t do this ☹

Building in depth: What do I use?

- Terrain
- Basic parts
- CSG
- MeshParts

- All of the above!
- MEASURE EVERYTHING

Conclusion



- Use our profiling tools
- Lean on our features for performance
- Don’t hesitate to ask us why X is slow
- Your feedback drives our work

Q & A