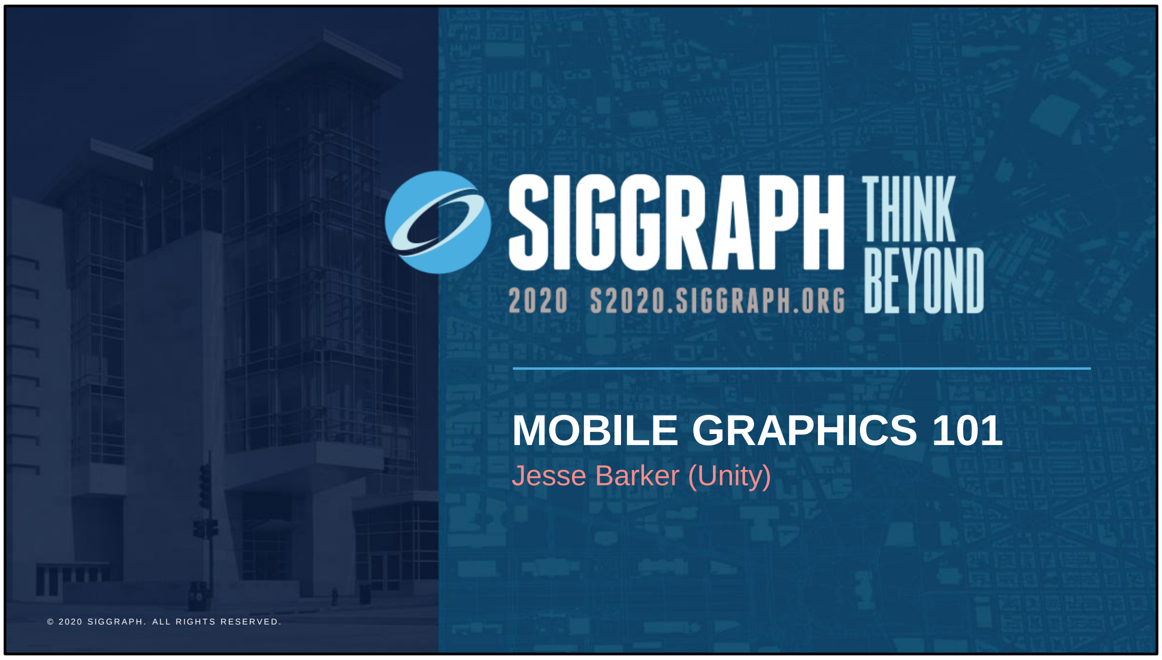
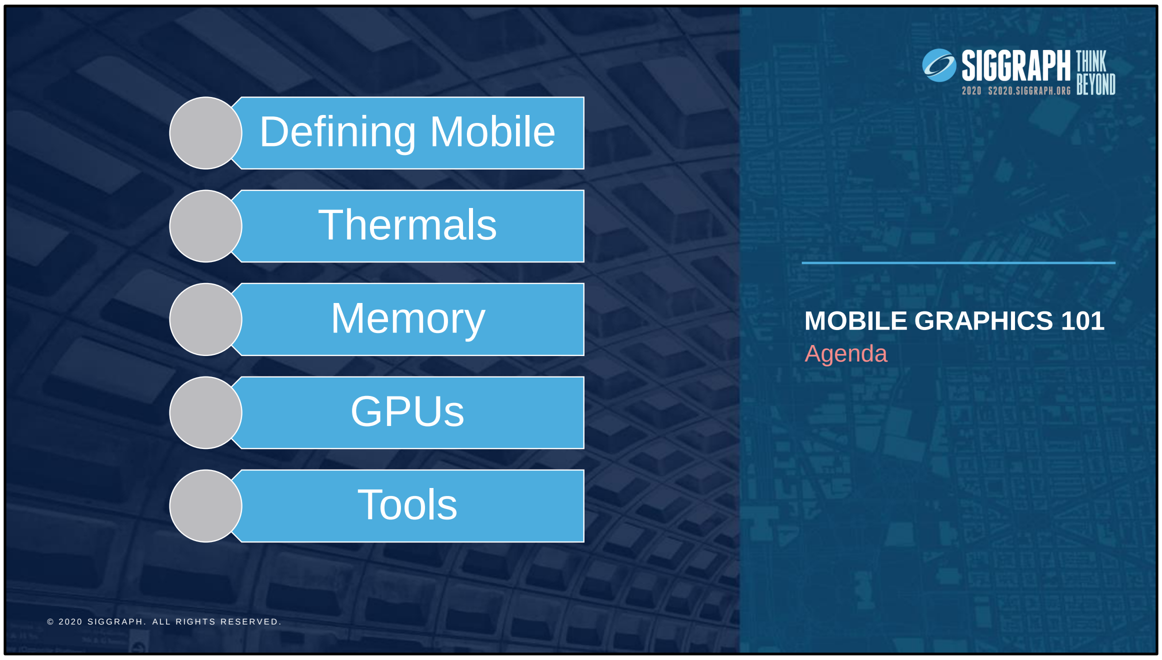


MOBILE GRAPHICS 101

Jesse Barker (Unity)

A little background on myself:

- Many years of low-level GPU software (drivers, advance feature development)
- Decade of mobile development
- Vulkan working group
- Lots of SIGGRAPH



Defining Mobile

Thermals

Memory

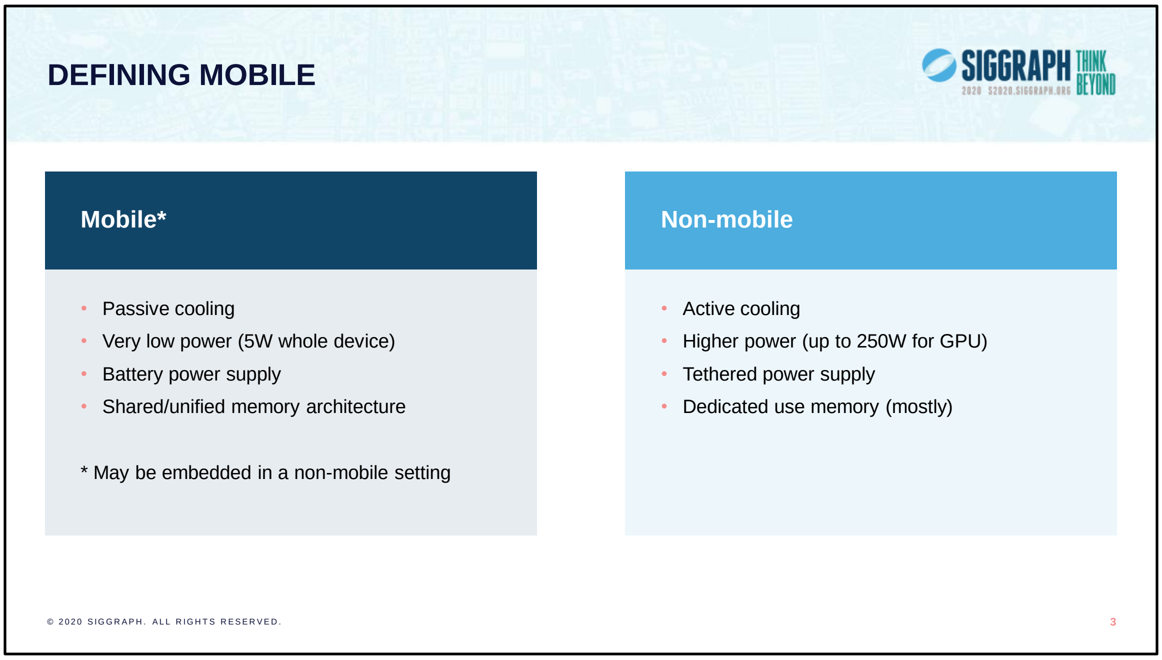
GPUs

Tools

MOBILE GRAPHICS 101

Agenda

We're going to start off with a quick definition of what we mean by the term "mobile" just for a baseline for what follows.
Following that, we will look at a few of the key factors that constrain (not just) graphics execution on these devices/platforms.
Then we will have a high-level overview of GPU styles and why mobiles tend to favor a particular approach (you may already know the answer by then).
Finally, we'll touch on a few tips and gotchas when analyzing the software you are running on these devices.


DEFINING MOBILE

Mobile*

- Passive cooling
- Very low power (5W whole device)
- Battery power supply
- Shared/unified memory architecture

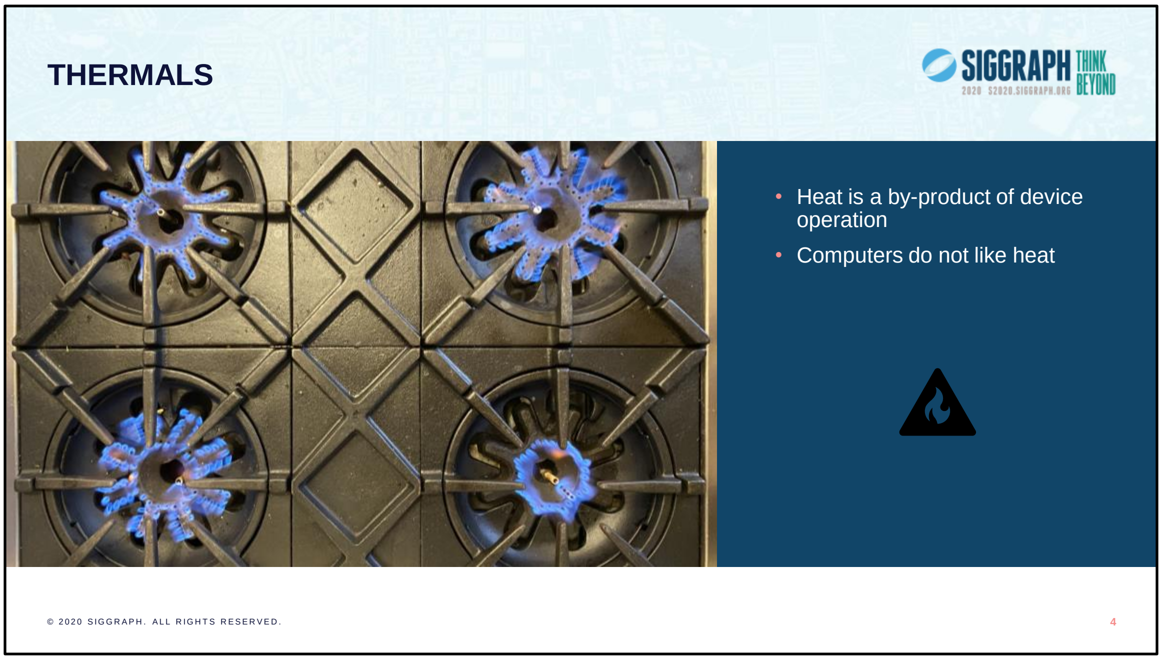
* May be embedded in a non-mobile setting

Non-mobile

- Active cooling
- Higher power (up to 250W for GPU)
- Tethered power supply
- Dedicated use memory (mostly)

I'm avoiding the "mobile vs desktop" high-level description here; we're talking about a phone-like device. (Anticipating questions like "what about laptops/ultraportables/chromebooks").
These are endpoints of a spectrum, and there are of course things that fall in between.
The landscape is evolving all the time.

Mobile is about thermals, power, and, well, size (your favorite PC graphics card chip is several times the size of any mobile SoC).
A device may use a "mobile" SoC, but if it has a tethered power supply, a fan (or even liquid cooling), I would argue it's no longer mobile,
though many of the considerations for programming them optimally still apply (GPU is the same, but some embedded environments disallow virtual memory or dynamically allocated memory, for example).

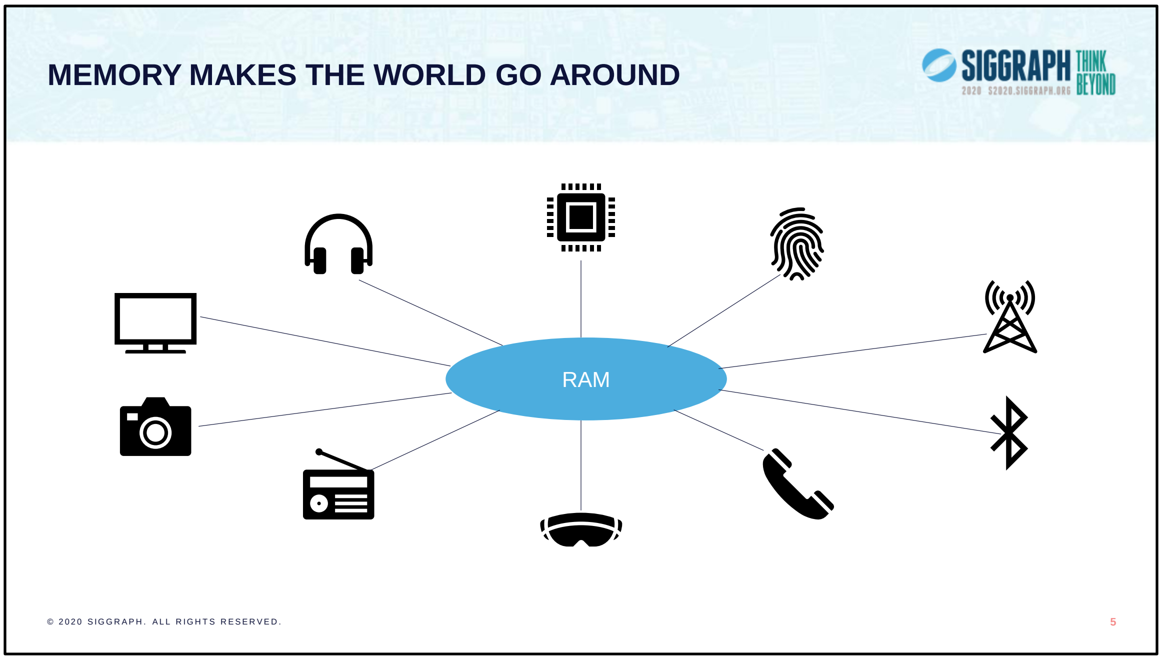


THERMALS

- Heat is a by-product of device operation
- Computers do not like heat

4

Every bit of active silicon in the SoC, as well as other active devices on the system board of the device generates heat. Computers don't like heat (they shut down and/or melt).
Phones and tablets are limited to passive cooling (whatever heat dissipation the device form factor allows inherently).
Tethered systems are able to run fans and even liquid cooling systems to address the thermal limits of the hardware.
Mobile devices must constantly monitor thermal conditions, and adjust the voltage and frequency of the portions of the chips that are too hot or drawing too much power.
This can have a huge impact on the behavior and performance of software on the device.
We all know that CPUs and GPUs (and other execution units in the system) draw power to operate, but the real culprit is…



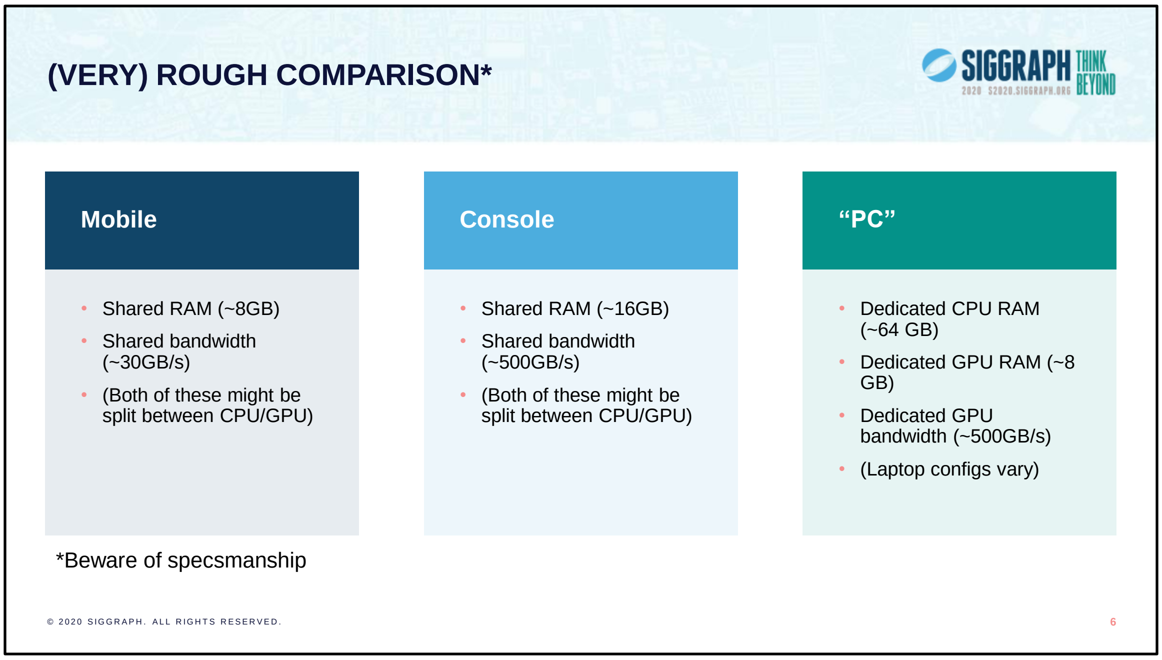
MEMORY MAKES THE WORLD GO AROUND

SIGGRAPH THINK BEYOND
2020 S2020.SIGGRAPH.ORG
RAM

5

Remember all of those “devices” inside an SoC that we talked about during the introduction? All of those require memory to be usable.

- Talking about memory is confusing. Quantity is significant, but tells you nothing about how useful that memory is
    - (e.g., what if you had 64GB of RAM, but only had an 8-bit bus running at 1MHz to access it).
- (I would argue) Memory is the single most expensive resource on a mobile device, including lighting up the display (dynamic memory must be constantly refreshed to preserve its contents).
- Memory configuration varies greatly based on system architecture (Dedicated, shared, split – other special cases as we’ll see)
- Systems may employ specialized memory types for specific use cases (e.g., GDDR memory for GPUs, and LPDDR memory for mobile).



(VERY) ROUGH COMPARISON*

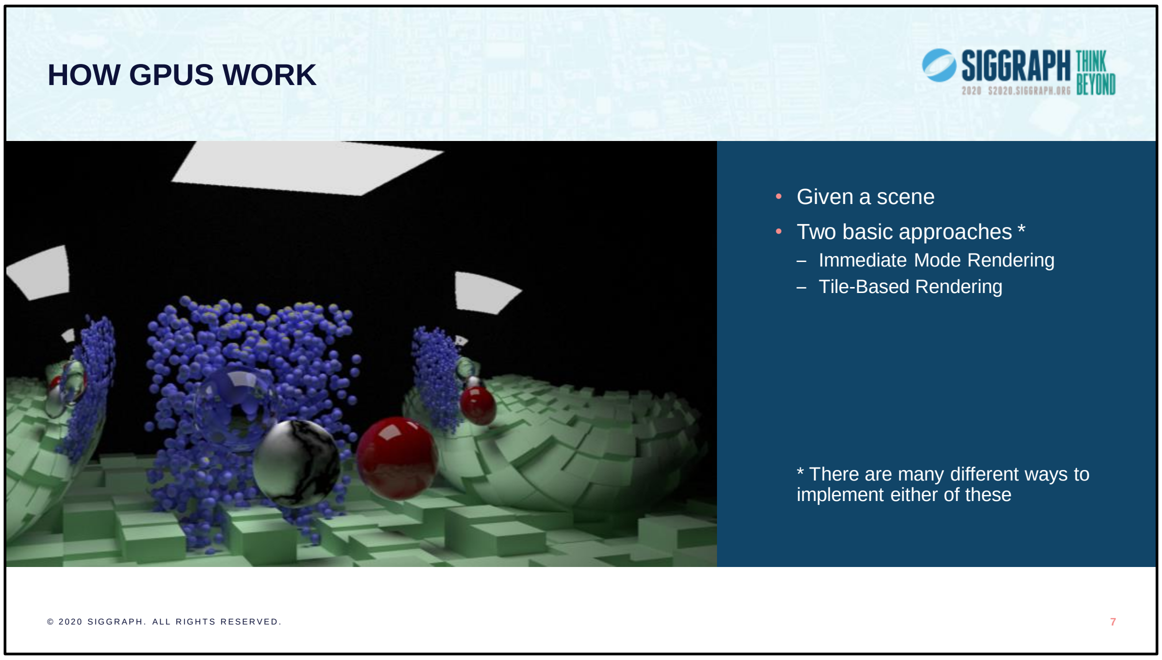
Mobile	Console	"PC"
• Shared RAM (~8GB) • Shared bandwidth (~30GB/s) • (Both of these might be split between CPU/GPU)	• Shared RAM (~16GB) • Shared bandwidth (~500GB/s) • (Both of these might be split between CPU/GPU)	• Dedicated CPU RAM (~64 GB) • Dedicated GPU RAM (~8 GB) • Dedicated GPU bandwidth (~500GB/s) • (Laptop configs vary)

*Beware of specsmanship

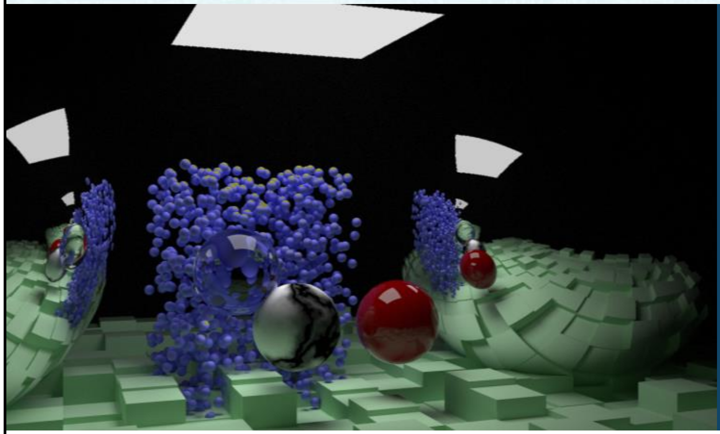
6

- Note: These numbers don't describe a single device in each category, but represent a rough overview of latest generation specs from SoC specifications, GPU vendor specifications, or review sites like "Tom's Hardware" or "AnandTech" (these latter will show more context by running benchmark workloads on a variety of devices)
- E.g. https://www.tomshardware.com/features/playstation-5-vs-xbox-series-x

- Discrete GPUs targeted at tethered systems (PCs, Macs, etc.) have lots of local/dedicated (on the GPU side of the bus) RAM with a huge pipe (lots of bandwidth) connecting the GPU with the memory.
- Mobile GPUs share a much smaller amount of memory with the rest of the System on a Chip (SoC) – a unified memory architecture - via the same pipe (much smaller than the dedicated pipe that discrete GPUs enjoy).
- With respect to bandwidth, there is typically at least an order of magnitude (and often much greater) difference between these devices.
- Raw hardware specifications aside, the available memory footprint and bandwidth are significantly reduced at runtime (power, thermals, other system overhead, and even the type of memory chips used).



HOW GPUS WORK



- Given a scene
- Two basic approaches *
  - Immediate Mode Rendering
  - Tile-Based Rendering

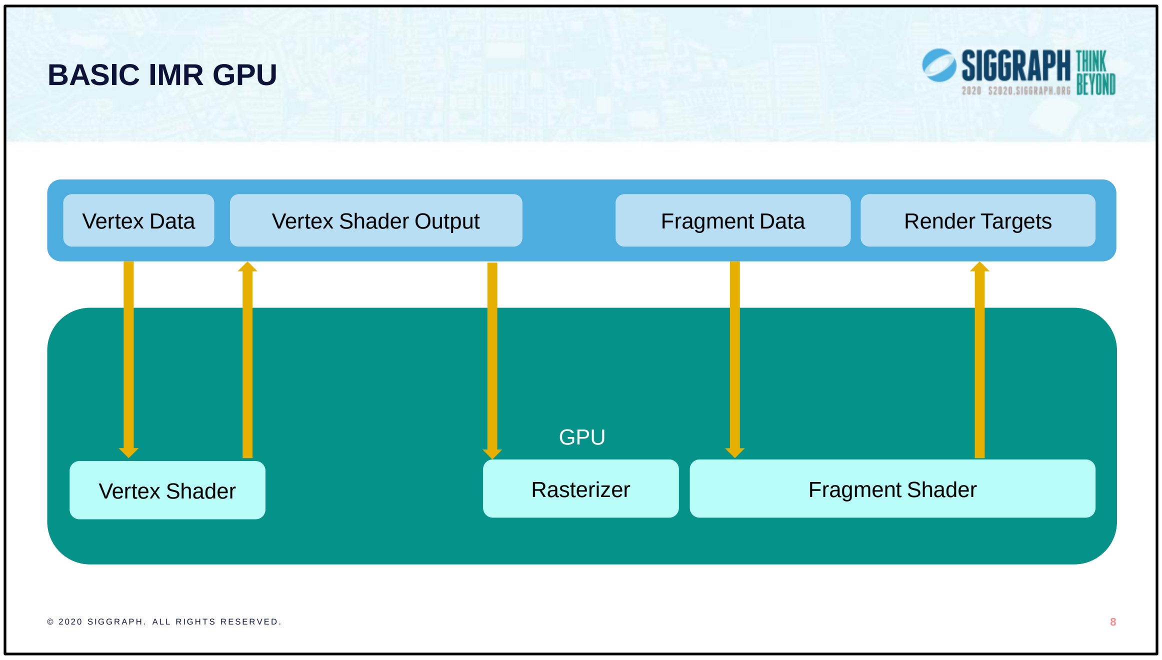
* There are many different ways to implement either of these

7

We're not doing a deep dive here, so I won't be covering low-level implementation details for a specific vendor architecture.

Every vendor's hardware is different, and indeed, each generation/architecture is likely different from its predecessor (or even a "sibling" within a generation).

This is intended to give you a basic grounding in how GPUs work.



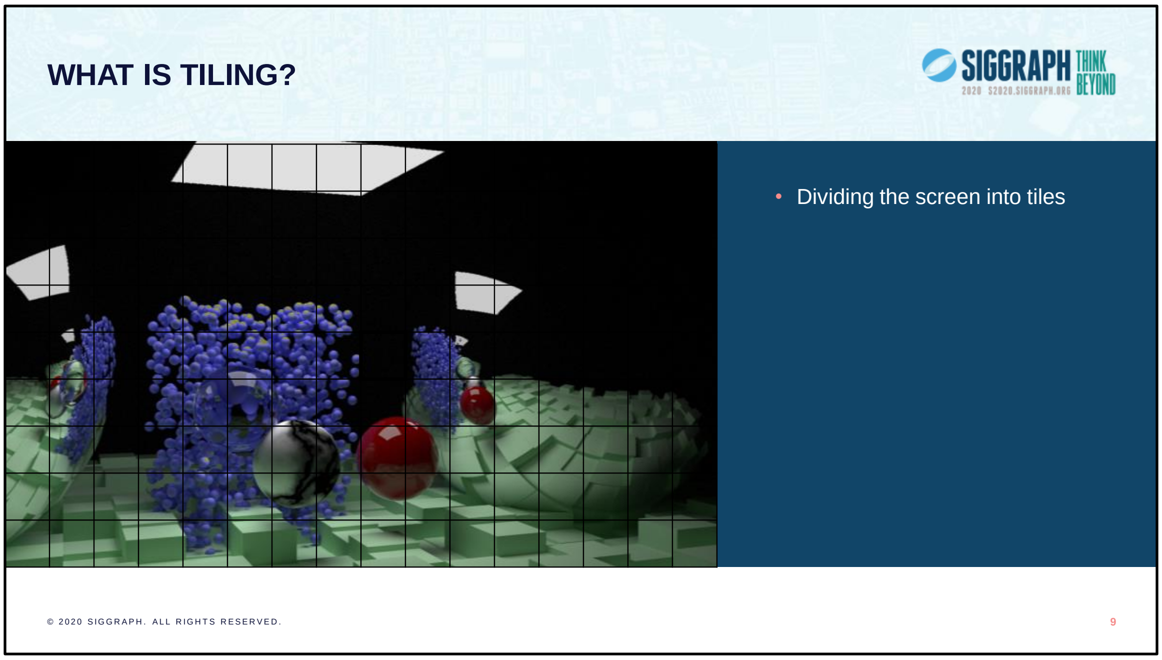
BASIC IMR GPU

8

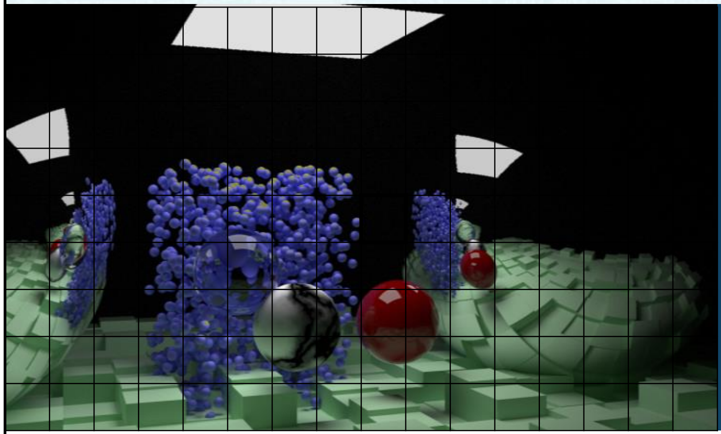
This is a very over simplified diagram of what an immediate mode rendering GPU looks like. (Green block GPU silicon, blue block RAM)
(if you want more detail of what happens inside each block, your favorite PC GPU vendor will have better breakdowns of how they process low-level work)

This type of GPU ”simply” (there are lots of hardware optimizations that make this not so simple):
- fetches primitive (typically triangles) data from a buffer
- transforms them into screen space (additional vertex shader output may go out to RAM, but may have an on-chip buffer)
- rasterizes (interpolates the screen-space vertex data) them
- shades all of the fragments/pixels that the rasterizer spits out (including any blending, etc. – “fragment data” contains textures, and other buffers containing per-fragment information)

Plenty of opportunities for the GPU to discard work, but this is _very_ memory intensive

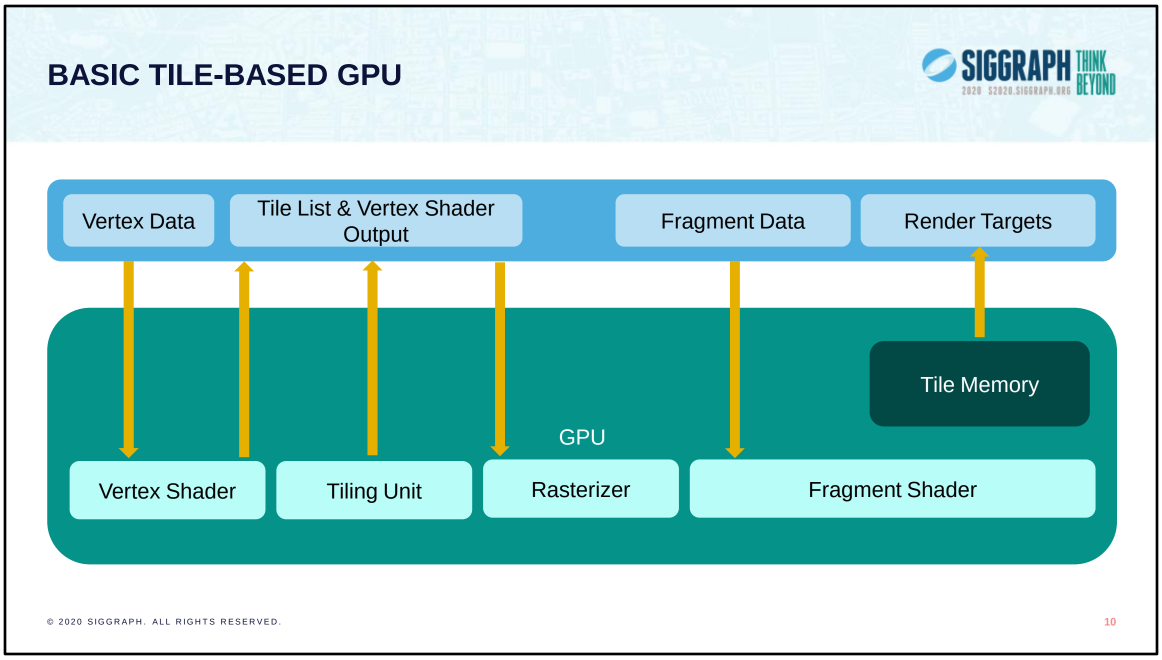


WHAT IS TILING?



- Dividing the screen into tiles

When we talk about a tile-based rendering GPU, this is what we mean. Each render target is divided into a rectangular grid of tiles.
Now let's look at how that happens and why it is so important for mobile GPUs.



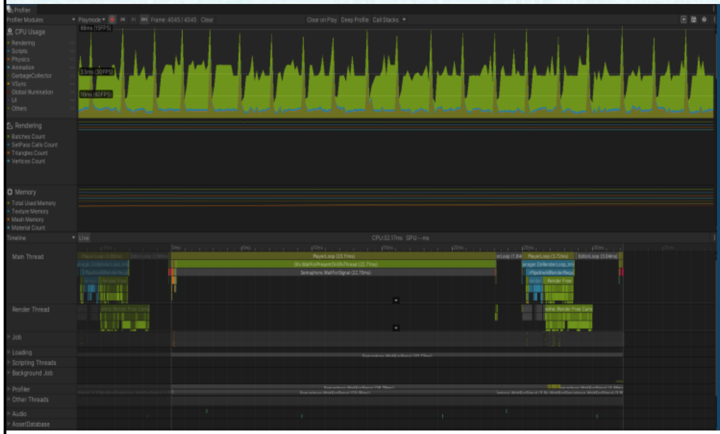
BASIC TILE-BASED GPU

This is a very simplified diagram of a tile-based GPU (again, how this is all done is very vendor-specific). (Green block GPU silicon, blue block RAM)
You may also hear "tile-based deferred renderer" or "tile-based immediate renderer". These fall into the
category of variations on a theme, and refer to implementation details that may or may not allow a GPU to avoid work.

You'll notice some similarity here to the IMR diagram (I made the diagrams that way on purpose ☺). This type of GPU "simply":
 - Fetches primitives from a buffer
 - Transforms them into screen space
 - (Tiling Unit) Sorts them into "bins" (one bin per "tile" of screen space)
   - This is the key: all scene/frame data is processed this way before we continue down the pipeline.
 - Once all of the geometry has been binned, each bin, the bin is rasterized and sent off for fragment shading
 - The work for each bin (including overdraw and depth testing) is performed in very fast on-chip memory, the results of which are finally written out to RAM
Note about the "Tile Memory" block here:

it is typically large enough to store multiple multisampled render targets (the size of the tile is adjusted to account for this)
This means that the only memory that needs to be allocated is the single-sample color output
  – MSAA color, depth, and stencil are typically not backed by RAM on these devices (Yay, big memory win!).
Other possible optimizations for such GPUs:
 - Splitting parameter interpolation from position transform (less work in vertex shader)
 - Forward pixel kill (remove fragments from the queue based on "early Z" tests – reduces number of times each fragment is shaded)
 - Hidden surface removal (similar, also based on "early Z" tests, but never issues the fragment rather than killing it after)
I mention these here as they are hardware optimizations that you have no control over (we'll talk about software optimization next)

DEBUGGING & OPTIMIZATION



- Mobile considerations:
  - Texture Compression
  - Data Types
  - Screen resolution

Platform & vendor tools are your friend. Game development tools (like the Unity timeline profiler seen here) can show you a lot if you're using them, but they have their limitations being fundamentally cross-platform.
There are also platform specific tools, as well as tools from GPU vendors that may show you a deeper view of your workloads, and you should familiarize yourself with them.

High-level optimizations:

Compact texture encodings are your friend. Even if you are already using compact representations for texture data
(RGBM light maps, octahedral normal maps), you want to compress them if possible. Bandwidth savings can be huge.
Downside is cross-platform availability (ETC/ETC2 pretty universal, ASTC is getting there and also includes HDR support).

Use fp16 (as well as lower precision integer representations) where possible both for shader input/output (bandwidth), as well as for operands (ALU performance often 2x fp32).

Reduced precision comes with some drawbacks like limited texture addressability and results like NaNs, so these optimizations must be carefully validated
(debug options like shaders with NaN elimination can help).

(Almost) Never render at native device resolution (you are outputting to a “small”, probably smudged screen and you won’t notice the difference).
Some devices have a small enough native resolution, but if you are rendering at QHD+, you are asking way too much out of your CPU, GPU, and memory system.
Render at a fractional resolution (testing to see what looks acceptable), and upscale and composite with a full resolution UI.

Important note when analyzing and optimizing graphics applications on tile-based GPUs:
 - Per-draw statistics are nearly impossible due to the way the draws are processed (as we’ve just seen).
 - Some vendor tools help with this, but those can be synthetic approximations (depending on whether the replay replicates the active power/clock state from the capture).
 - Combine live timeline analysis with frame/shader replay debugging to validate that optimizations are actually helpful.

THANKS!