High-Performance Software Rasterization on GPUs

Samuli Laine Tero Karras

NVIDIA Research

HIGH-PERFORMANCE GRAPHICS
VANCOUVER, CANADA AUGUST 5-7, 2011

Graphics and Programmability

- Graphics pipeline (OpenGL/D3D)
  - Driven by dedicated hardware
  - Executes user code in shaders
- What's next in programmability?

Programmable Graphics Pipeline

- But what does it mean?
  - Another API?
  - More programmable stages?
  - Coupling between CUDA/OpenCL and OpenGL/D3D?
  - Or ”it’s just a program”?

Our Approach

- Try and implement a full pixel pipeline using CUDA
  - From triangle setup to ROP
- Obey fundamental requirements of gfx pipe
  - **Maintain input order**
  - Hole-free rasterizer with correct rasterization rules
- Prefer speed over features

Project Goals

- Establish a firm data point through careful experiment
- Provide fully programmable platform for exploring algorithms that extend the hardware gfx pipe
  - Programmable ROP
  - Stochastic rasterization
  - Non-linear rasterization
  - Non-quad derivatives
  - Quad merging
  - Decoupled sampling
  - Compact after discard
  - etc.
- Ideas for future hardware
  - Ultimate goal = flexibility of software, performance of fixed-function hardware

Previous Work: FreePipe [Liu et al. 2010]

- Very simple rasterization pipeline
  - Each triangle processed by one thread
  - Blit pixels directly to DRAM using atomics
- Limitations
  - Cannot retain inputs order
  - Limited support for ROP operations (dictated by atomics)
  - Large triangles → game over
- We are 15x faster on average
  - Larger difference for high resolutions

Design Considerations

- Run everything in parallel
  - We need a lot of threads to fill the machine
- Minimize amount of synchronization
  - Avoid excessive use of atomics
- Focus on load balancing
  - Graphics workloads are wild

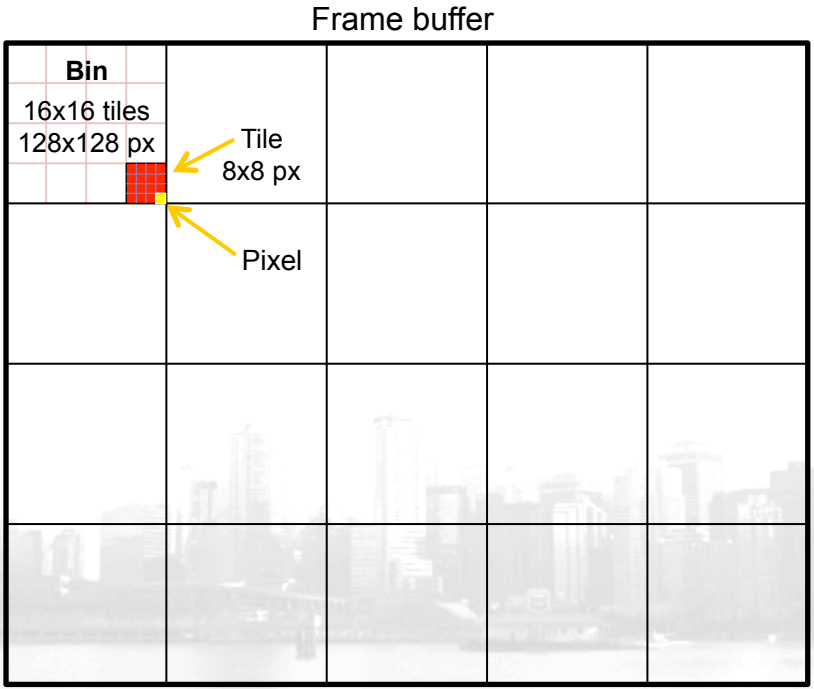
Pipeline Structure

- Chunker-style pipeline with four stages

  Triangle setup → Bin raster → Coarse raster → Fine raster

- Run data in large batches
  - Separate kernel launch for each stage
- Keep data in input order all the time

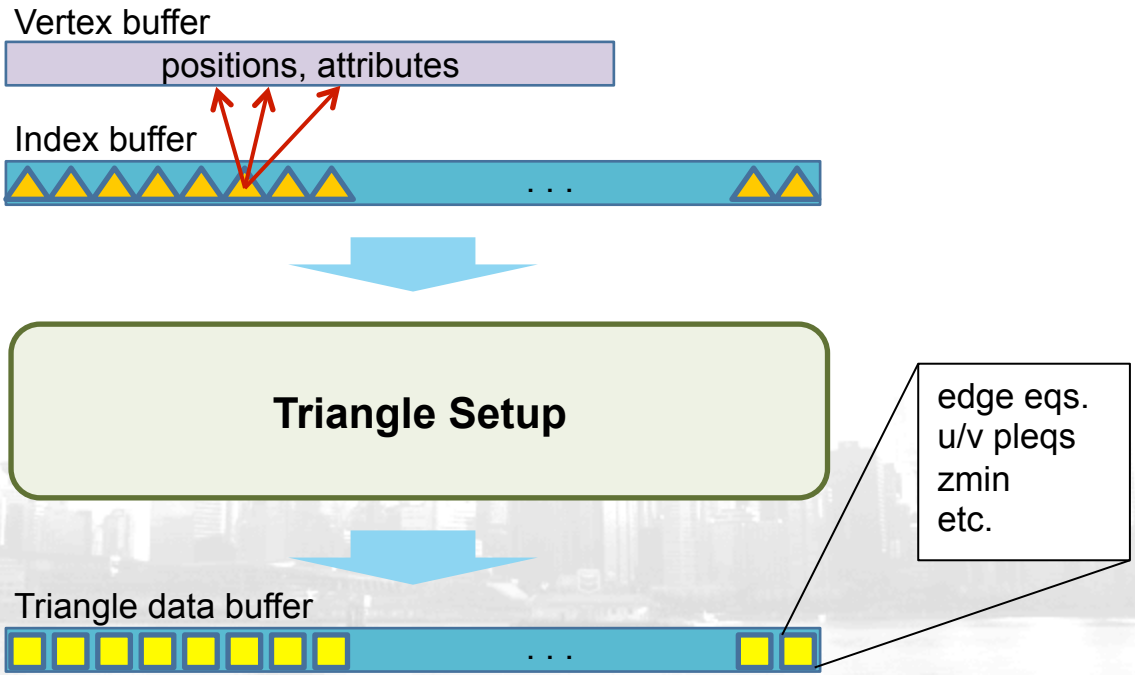
Chunking to Bins and Tiles



Pipeline Stages

- Quick look at each stage
- More details in the paper

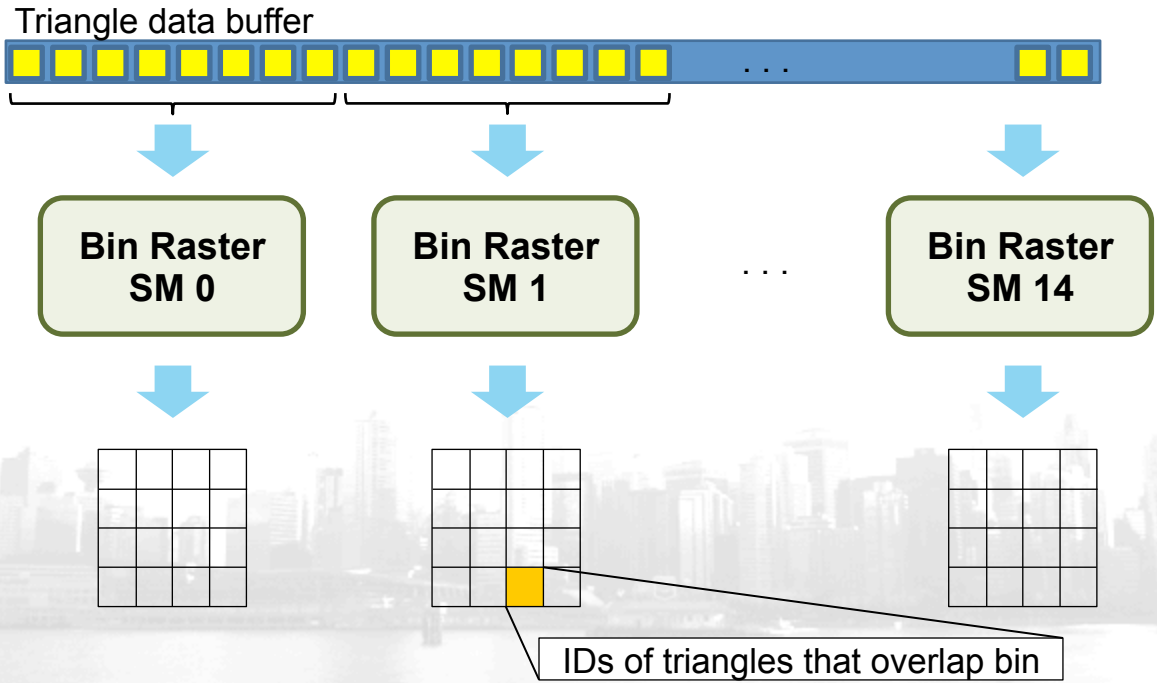
Triangle Setup



Triangle Setup

- Fetch vertex indices and positions
  - Clip if necessary (has guardband)
  - Frustum, backface and between-samples cull
  - Setup screen-space pleqs for u/w, v/w, z/w, 1/w
  - Compute zmin for early depth cull in fine raster
- One-to-one mapping between input and output
  - Trivial to employ full chip while preserving ordering

Bin Raster


Triangle data buffer
. . .
Bin Raster
SM 0
Bin Raster
SM 1
. . .
Bin Raster
SM 14
IDs of triangles that overlap bin

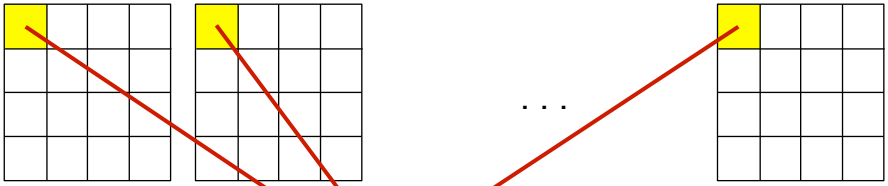
Bin Raster, First Phase

- Pick triangle batch (atomic, 16k tris)
- Read 512 set-up triangles
- Compact/expand according to culling/clipping results
  - Efficient within thread block
- Repeat until has enough triangles to utilize all threads

Bin Raster, Second Phase

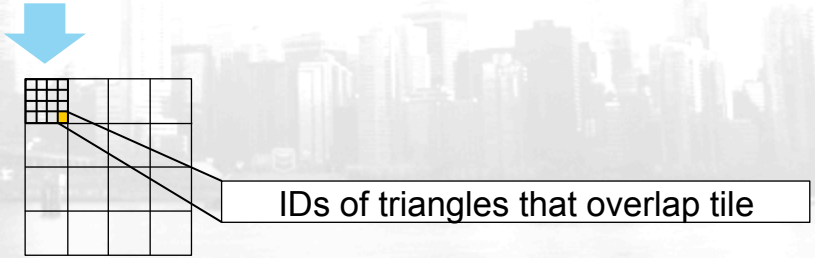
- Rasterize
  - Determine bin coverage for each triangle (1 thread per tri)
  - Fast paths for 2x2 and smaller footprints
- Output
  - Output to per-SM queue → no sync between SMs

Coarse Raster



Coarse Raster SM *n*

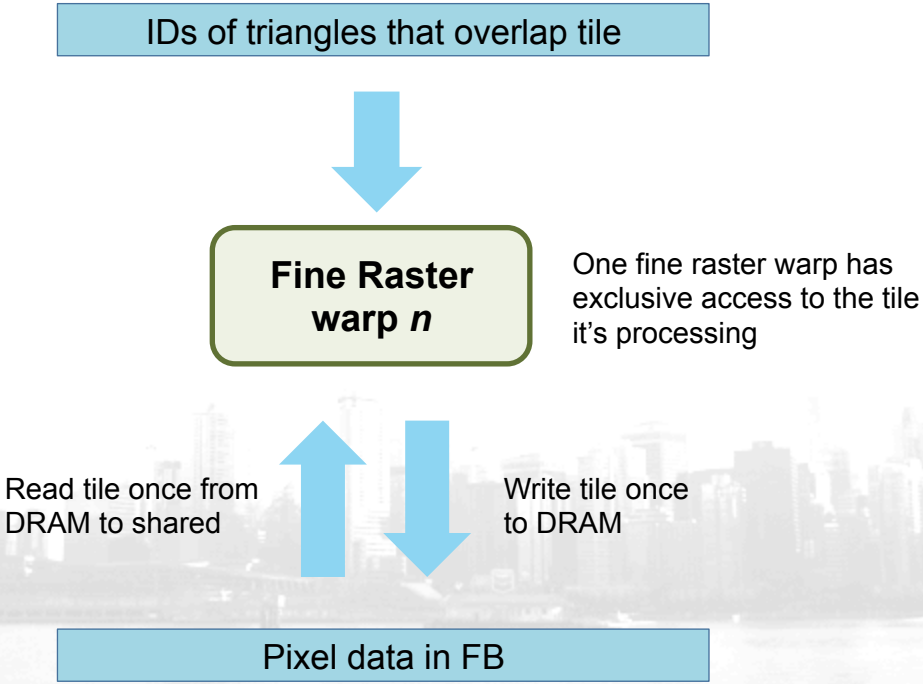
One coarse raster SM has exclusive access to the bin it's processing



Coarse Raster

- Input Phase
  - Merge from 15 input queues (one per bin SM)
  - Continue until enough triangles to utilize all threads
- Rasterize
  - Determine tile coverage for each triangle (1 thread per tri)
  - Fast paths for small / largely overlapping triangles
- Output
  - Highly varying number of output items → divide evenly to threads
  - Only one SM outputs to tiles of any given bin → no sync needed

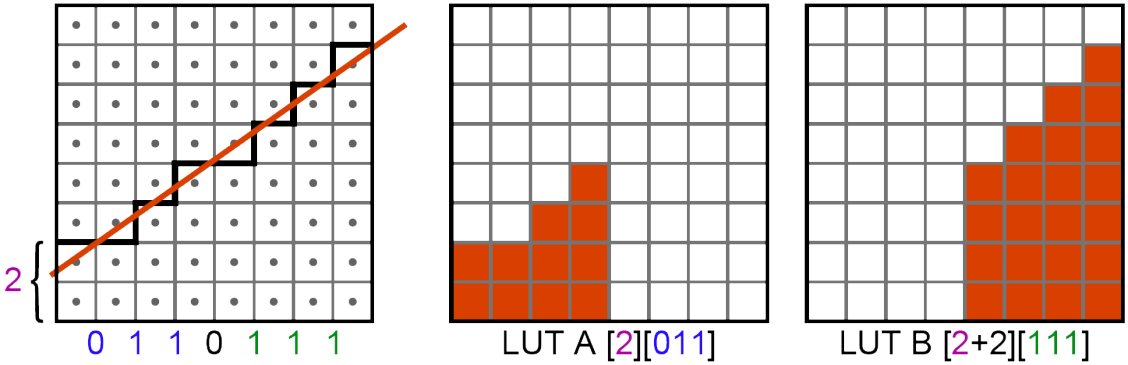
Fine Raster


IDs of triangles that overlap tile
Fine Raster warp *n*
One fine raster warp has exclusive access to the tile it's processing
Read tile once from DRAM to shared
Write tile once to DRAM
Pixel data in FB

Fine Raster

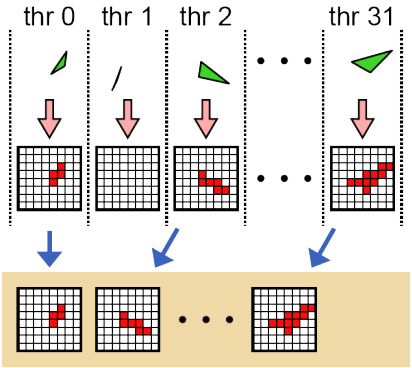
- Pick tile, read FB tile, process, write FB tile
- Input
  - Read 32 triangles in shared memory
  - Early z kill based on triangle zmin and tile zmax
  - Calculate pixel coverage using LUTs (153 instr. for 8x8 stamp)
  - Repeat until has at least 32 fragments
- Raster
  - Process one fragment per thread, full utilization
  - Shade and ROP

Tidbit 1: Coverage Calculation

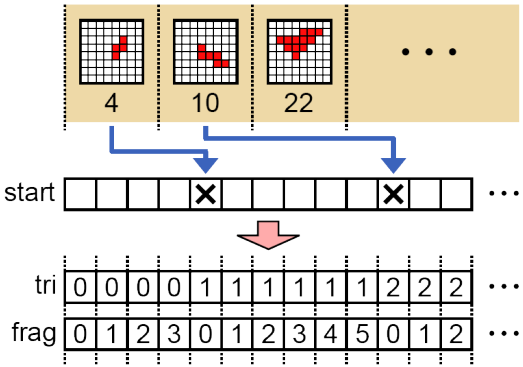


- Step along edge (Bresenham-like)
- Use look-up tables to generate coverage masks
- ~50 instructions for 8x8 stamp, one edge

Tidbit 2: Fragment Distribution



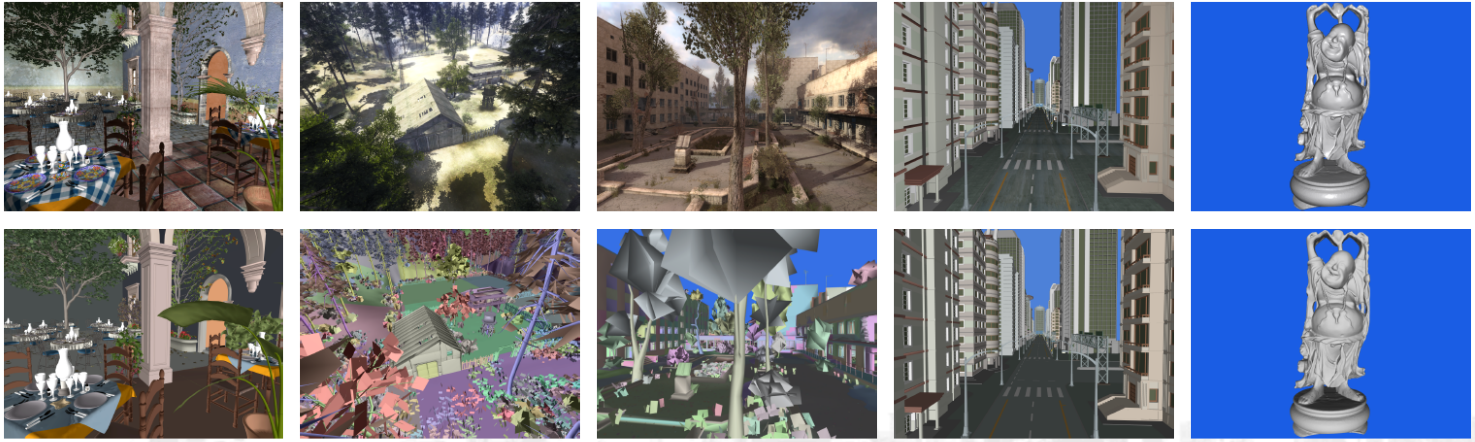
Input Phase



Shading Phase

- In input phase, calculate coverage and store in list
- In shading phase, detect triangle changes and calculate triangle index and fragment in triangle

Test Scenes



San Miguel, 189MB
5.44M tris, 25% visible
2.4 pixels / triangle

Juarez, 24MB
546K tris, 37% visible
14.6 pixels / triangle

Stalker, 11MB
349K tris, 41% visible
14.1 pixels / triangle

City, 51MB
879K tris, 21% visible
16.3 pixels / triangle

Buddha, 29MB
1.09M tris, 32% visible
1.4 pixels / triangle

Call of Juarez scene courtesy of **Techland**
S.T.A.L.K.E.R.: Call of Pripyat scene courtesy of **GSC Game World**

Results: Performance

Scene	Resolution	HW	Our (SW)	FreePipe (FP)	SW:HW ratio	FP:SW ratio
SAN MIGUEL	512×384	5.37	7.82	130.14	1.46	16.65
	1024×768	5.43	9.48	510.20	1.74	53.84
	2048×1536	5.86	15.44	1652.52	2.64	107.06
JUAREZ	512×384	0.59	2.71	5.34	4.56	1.97
	1024×768	0.67	3.28	18.63	4.87	5.69
	2048×1536	1.03	7.06	72.45	6.84	10.26
STALKER	512×384	0.31	1.81	23.47	5.91	12.96
	1024×768	0.39	2.31	92.73	5.96	40.14
	2048×1536	0.67	5.41	386.07	8.10	71.36
CITY	512×384	0.93	2.16	64.56	2.32	29.88
	1024×768	1.04	3.13	251.86	3.01	80.54
	2048×1536	1.42	6.79	1032.83	4.77	152.13
BUDDHA	512×384	1.06	2.09	2.14	1.98	1.02
	1024×768	1.07	2.66	3.08	2.50	1.16
	2048×1536	1.11	4.01	6.96	3.62	1.73

Frame rendering time in ms (depth test + color, no MSAA, no blending)

Results: Memory Usage

	San Miguel	Juarez	Stalker	City	Buddha
Scene data	189	24	11	51	29
Triangle setup data	420.0	42.2	26.9	67.9	84.0
Bin queues	4.0	1.5	1.2	0.9	2.0
Tile queues	4.4	2.9	2.2	2.2	1.5

Memory usage in MB

Comparison to Hardware (1/3)

- Resolution
  - Cannot match hardware in raster, z kill + compact
  - Currently support max 2K x 2K frame buffer, 4 subpixel bits
- Attributes
  - Fetched when used → bad latency hiding
  - Expensive interpolation
- Antialiasing
  - Hardware nearly oblivious to MSAA, we much less so

Comparison to Hardware (2/3)

- Memory usage, buffering through DRAM
  - Performance implications of reduced buffering unknown
  - Streaming through on-chip memory would be much better
- Shader complexity
  - Shader performance theoretically the same as in graphics pipe
- Frame buffer bandwidth
  - Each pixel touched only once in DRAM

Comparison to Hardware (3/3)

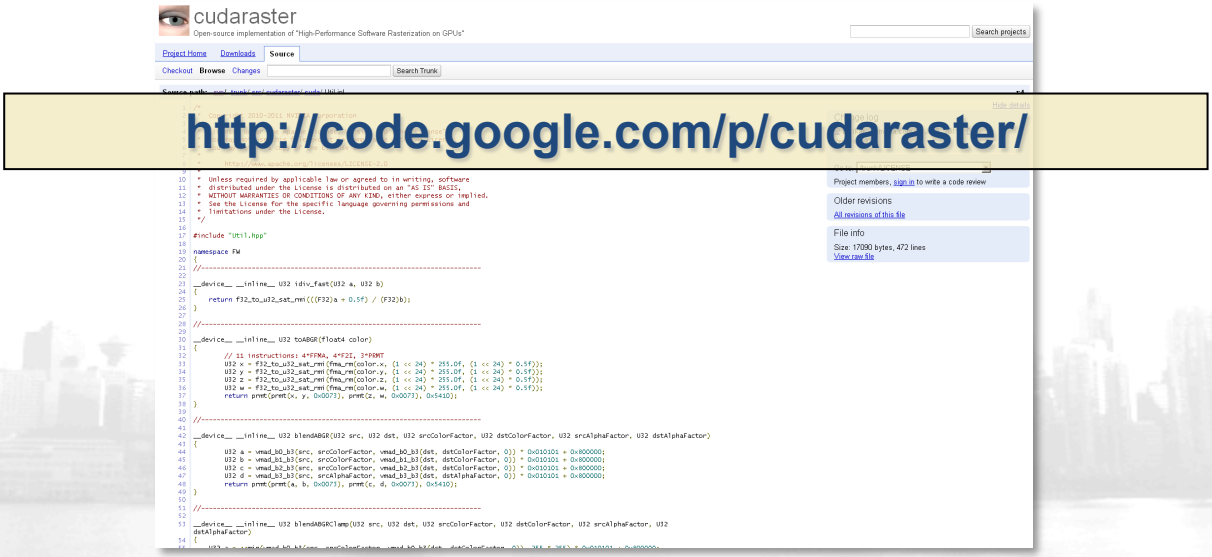
- **Extensibility**
  - Need one stage to do something extra?
  - Need a new stage altogether?
  - *You can actually implement it*
- **Specialization to individual applications**
  - Rip out what you don't need, hard-code what you can

Exploration Potential

- Shader performance boosters
  - Compact after discard, quad merging, decoupled sampling, …
- Things to do with programmable ROP
  - A-buffering, order-independent transparency, …
- Stochastic rasterization
- Non-linear rasterization
- (Your idea here)

The Code is Out There

- The entire codebase is open-sourced and released



http://code.google.com/p/cudaraster/

Thank You

- Questions