Accelerated Batch Rendering: A Software Implementation

Hongyu Teng (ht919)
27 June 2023

For better visual effect (including animated images), check the online version:
https://docs.google.com/presentation/d/1ffdKxlpKJ1MEsorblpnoqY0HKOvmpQs-GNU2ygfQIqY/edit

https://github.com/JoeyTeng/jaxrenderer

Motivation

Applications of RL (Reinforcement Learning)

Robots

Self-driving

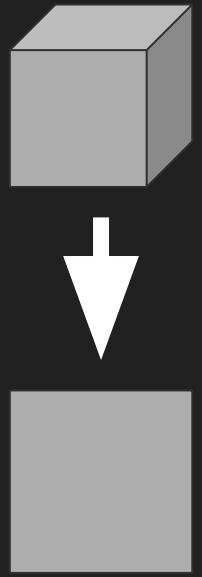
SayCan: Grounding Language in Robotic Affordances. https://say-can.github.io/

AutoPilot, Tesla: https://www.tesla.com/en_EU/autopilot

Applications in RL

Robots

Self-driving



Project 3D objects onto 2D sensor
(not just usual camera, maybe IR, LiDAR)

SayCan: Grounding Language in Robotic Affordances. https://say-can.github.io/

AutoPilot, Tesla: https://www.tesla.com/en_EU/autopilot

Applications in RL

- Use simulators to train agents
- Take **direct observation** from environment, or use internal states
  - Generate 2D observations using 3D env data?

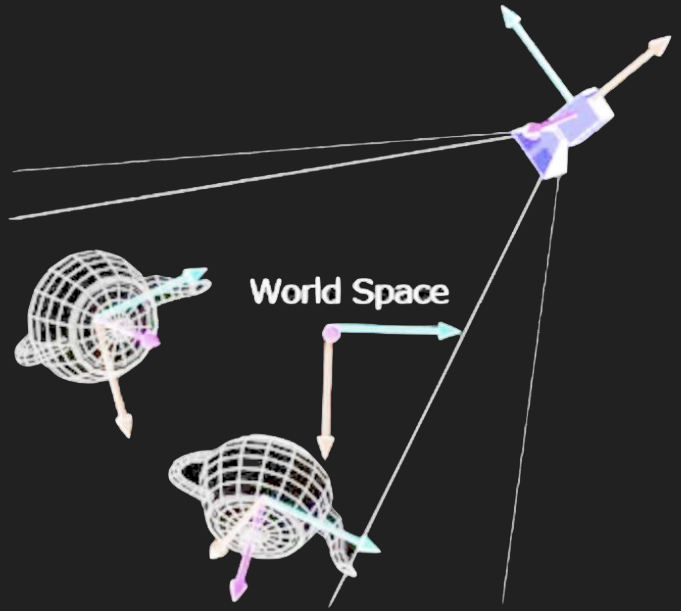
Applications in RL

- Use simulators to train agents
- Take **direct observation** from environment, or use internal states
  - Generate 2D observations using 3D env data?

Use Renderers!

Render

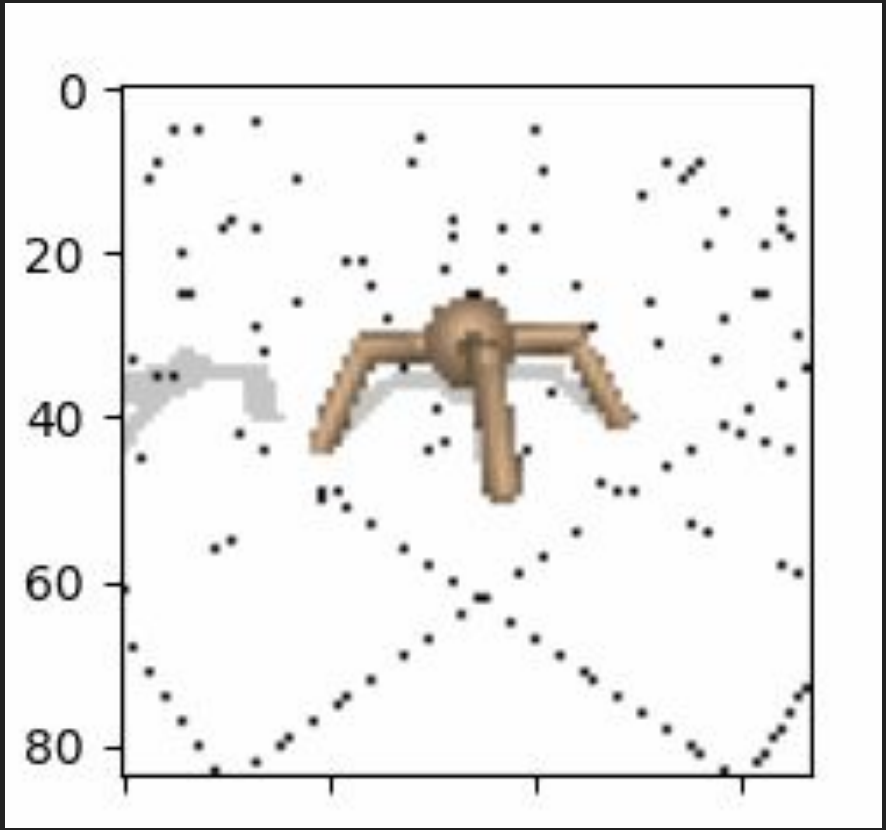
Project 3D objects onto 2D sensor,
then discretely sample them



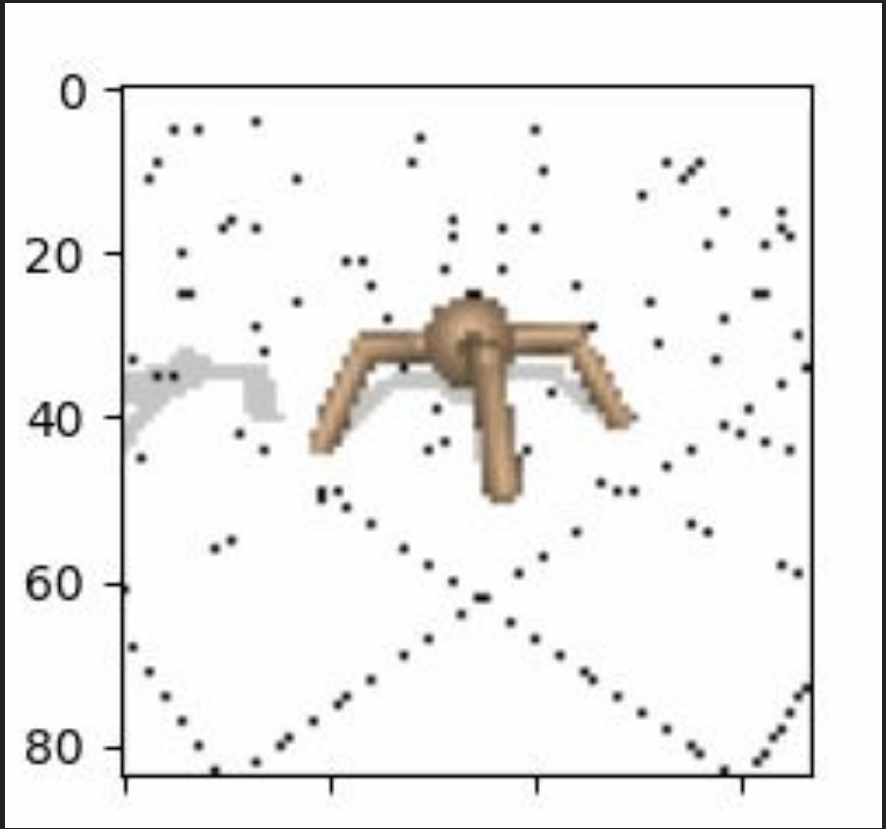
Similar to taking photos using
camera in the 3D scene

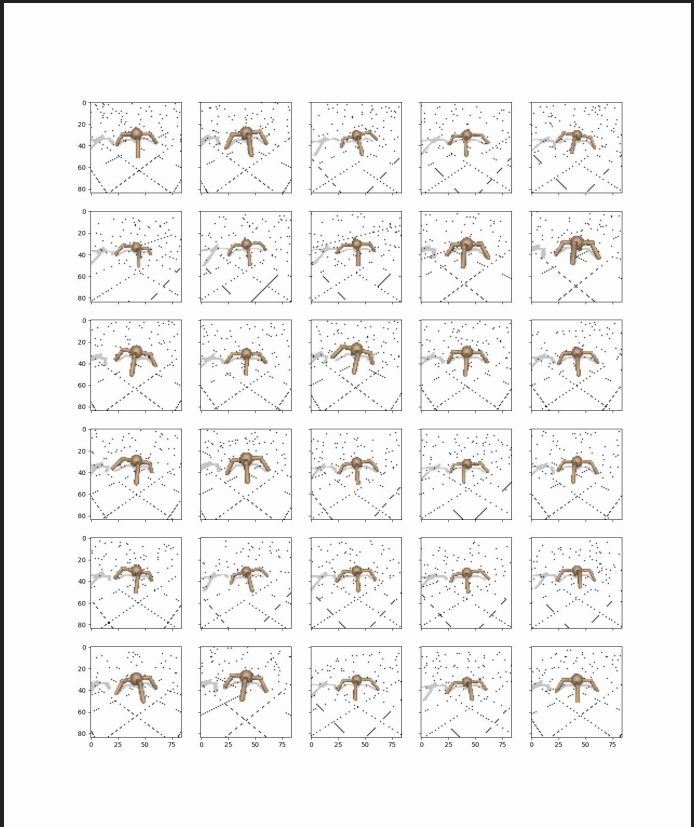
https://unsplash.com/photos/ItaAIuTh5Uc

Exciting Examples



90 Frames Animation, 84 x 84, in **1.91** s (**47fps**) (simulation + rendering)





30 Envs 90 Frames Animation, 84 x 84, in **5.26** s (**513fps**) (simulation + rendering)

Existing Solutions

- No batch rendering support, limit the throughput
- Not differentiable, limit the optimisation methods

Existing Solutions vs Our Solution

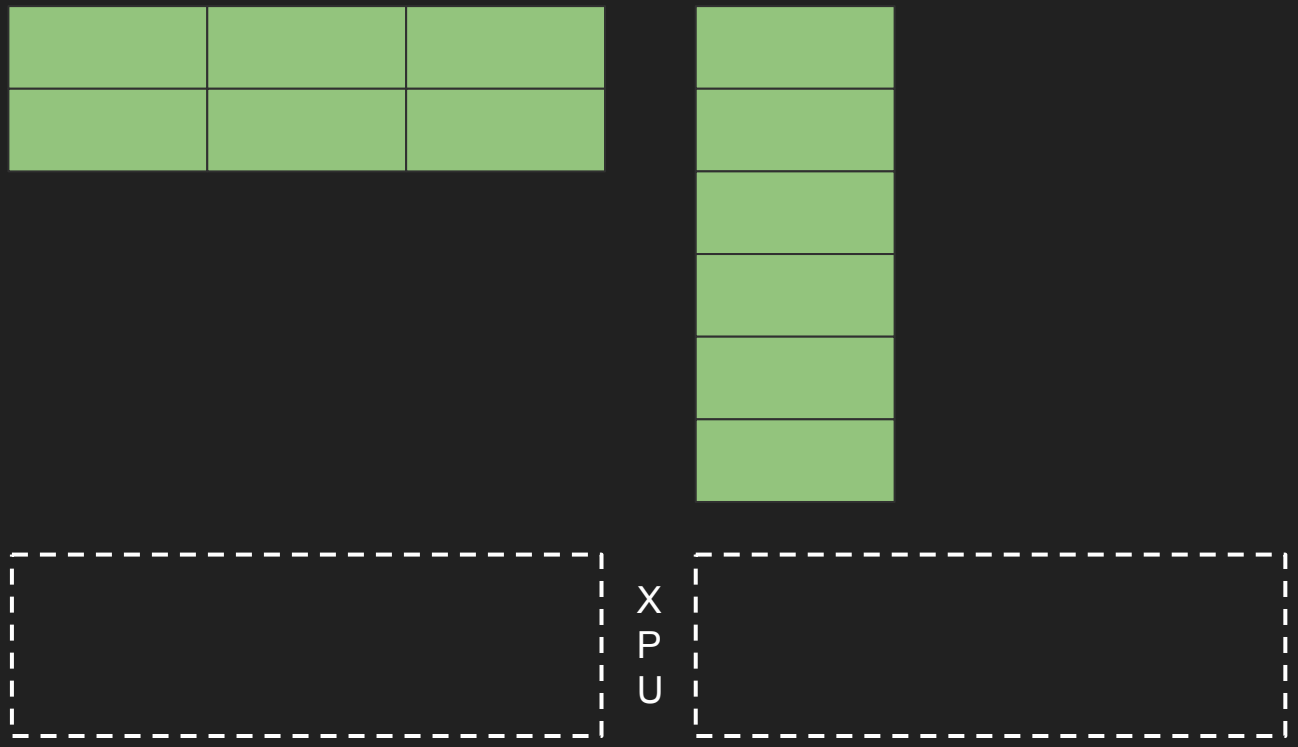
- No batch rendering support, limit the throughput
- Not differentiable, limit the optimisation methods

Using JAX to implement a software renderer

- Native batch computation & parallel/distributed computation support
- Automatic differentiable
- Natively run on accelerators (GPU, TPU)

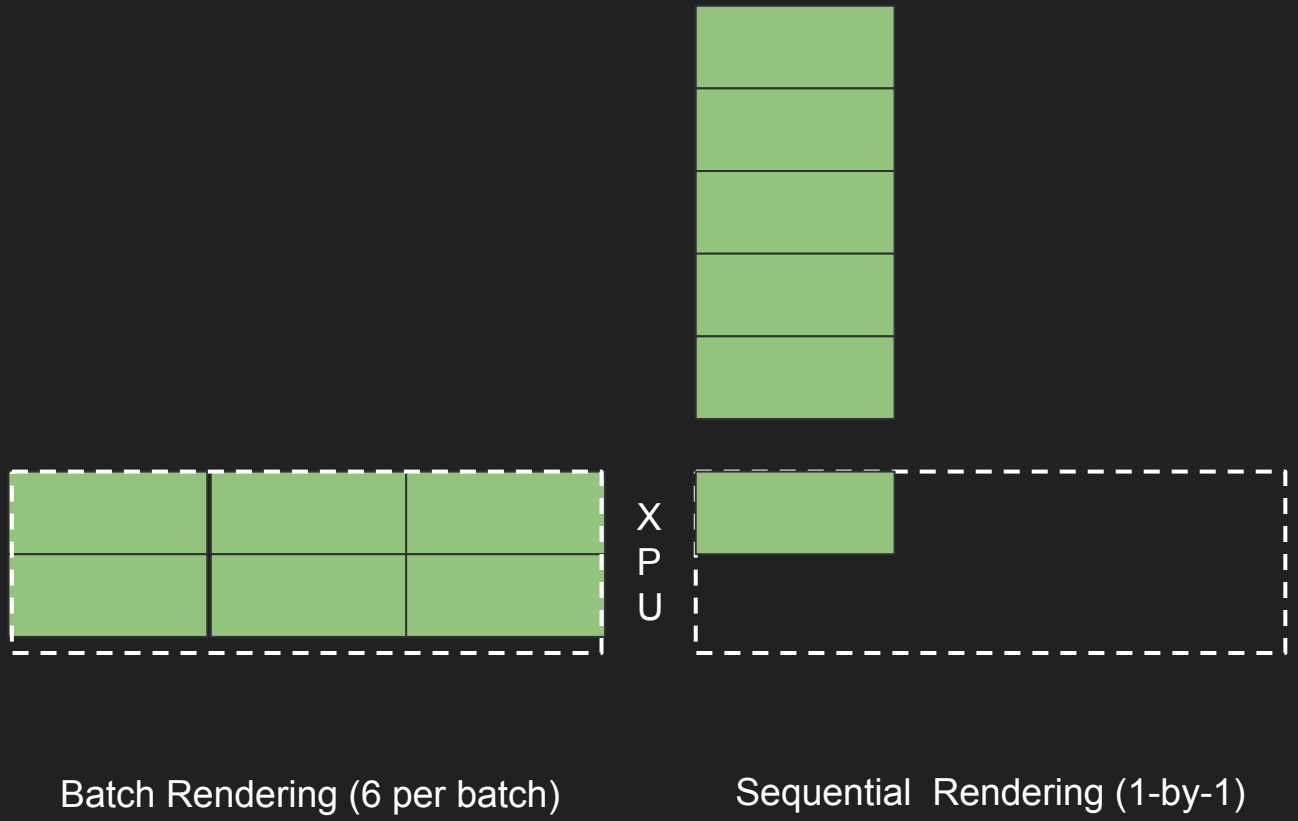
Batch Rendering

Render several frames together to increase utilisation and throughput



Batch Rendering

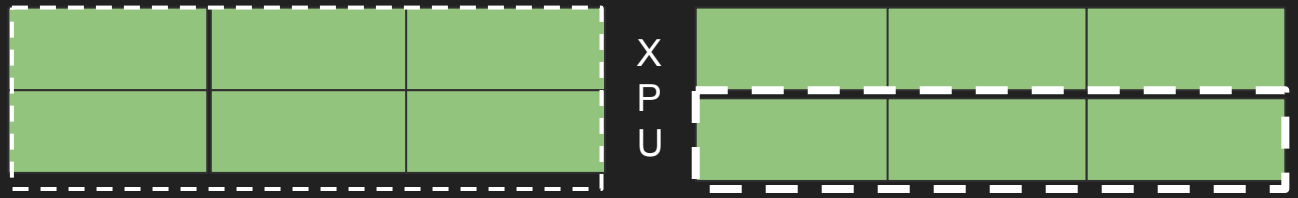
Render several frames together to increase utilisation and throughput



Batch Rendering

Render several frames together to increase utilisation and throughput

Compiler automatically divide big batch into smaller at op-level, and schedule them. This is done in compile time. (OOM still possible)



Capability 6 per batch

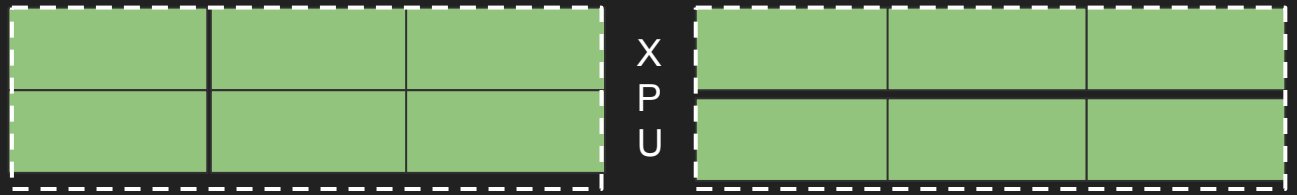
Capability 3 per batch

Batch Rendering

Render several frames together to increase utilisation and throughput

Compiler automatically divide big batch into smaller at op-level, and schedule them. This is done in compile time. (OOM still possible)

Easy distributed batch rendering on homogeneous XPUs at frame level



Render 12 frames in 2 XPUs in parallel, 6 frames each

JAX: Autograd and XLA

The Numerical Computation Framework Used

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, et al.. JAX: composable transformations of Python+NumPy programs; 2018. Available from: https://github.com/google/jax.

JAX: Advantages and Limitations

- ✅ Accelerator support (GPU, TPU)
- ✅ Simple JIT/AOT[1] compilation
- ✅ Simple automatic vectorisation
- ✅ Simple parallelisation & distribution
- ✅ Numpy-like API + some Sci-Py functions
- ✅ Autograd: automatic differentiation
- ✅ Type-hinting support[2]
- ~ Functional programming style:
  - Compostable function transformations
  - Pure functions + immutable data

- ❌ Heavy dispatch overhead (asynchronised)
- ❌ Long compilation time
  - Quadratic with the number of ops[3]
  - Must compile whole function at once
  - Change input shape causes re-compilation
- ❌ Statically shaped inputs & intermediate values
- ❌ Auto-parallelisation/distribution requires homogeneous device and data (shape)
- ❌ Limited data-dependent control flow
  - data must be static (known at compile time)
  - or, requires synchronisation with host
  - or, all branches are executed unconditionally[4]

[1]: JIT: Just-in-Time; AOT: Ahead-of-Time
[2]: Typing support through community solution jaxtyping: https://github.com/google/jaxtyping
[3]: Roughly estimation, see https://github.com/google/jax/discussions/3478
[4]: Under transformation. https://github.com/google/jax/pull/16335

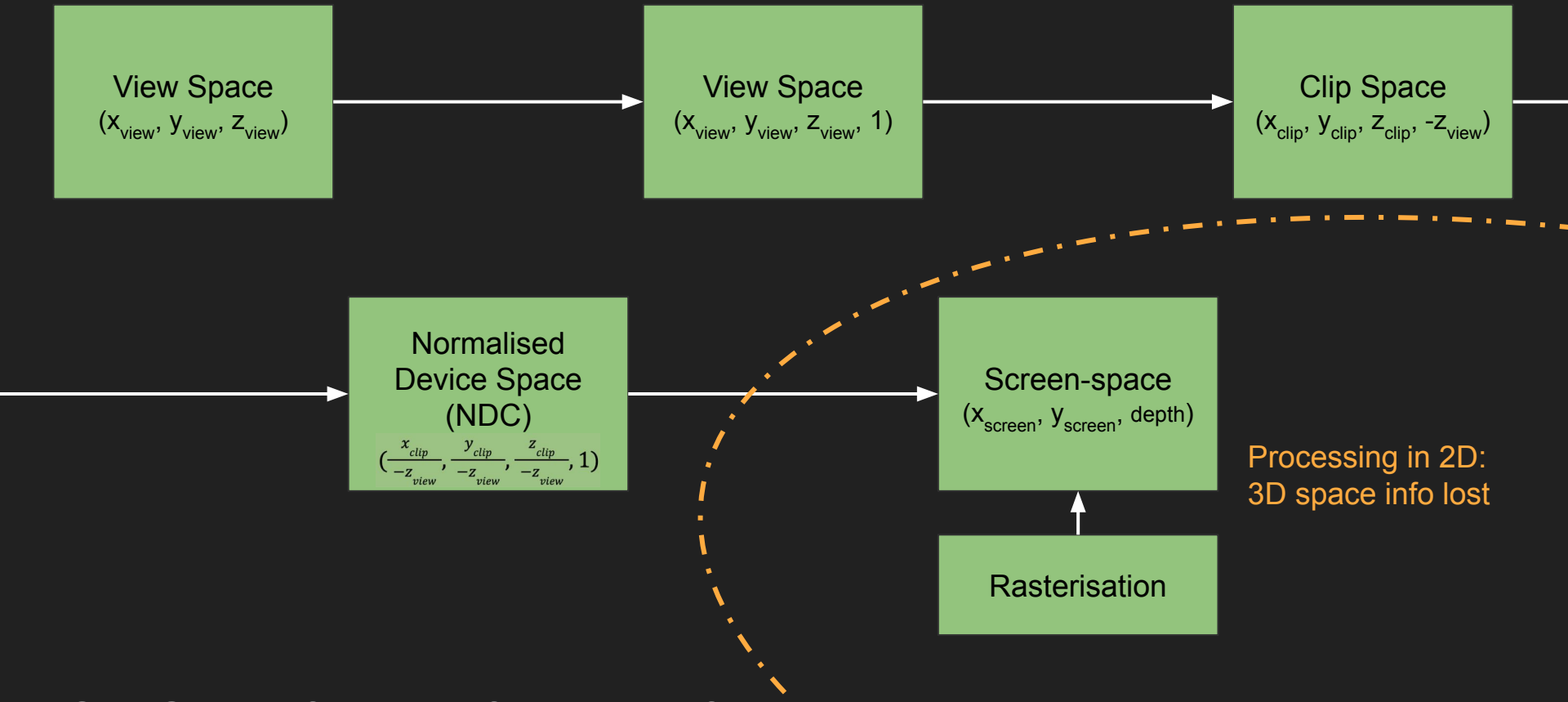
JAX: Advantages and Limitations (highlighted main ones)

- ✅ Accelerator support (GPU, TPU)
- ✅ Simple JIT/AOT[1] compilation
- ✅ Simple automatic vectorisation
- ✅ Simple parallelisation & distribution
- ✅ Numpy-like API + some Sci-Py functions
- ✅ Autograd: automatic differentiation
- ✅ Type-hinting support[2]
- ~ Functional programming style:
  - Compostable function transformations
  - Pure functions + immutable data

- ❌ Heavy dispatch overhead (asynchronised)
- ❌ Long compilation time
  - Quadratic with the number of ops[3]
  - Must compile whole function at once
  - Change input shape causes re-compilation
- ❌ Statically shaped inputs & intermediate values
- ❌ Auto-parallelisation/distribution requires homogeneous device and data (shape)
- ❌ Limited data-dependent control flow
  - data must be static (known at compile time)
  - or, requires synchronisation with host
  - or, all branches are executed unconditionally[4]

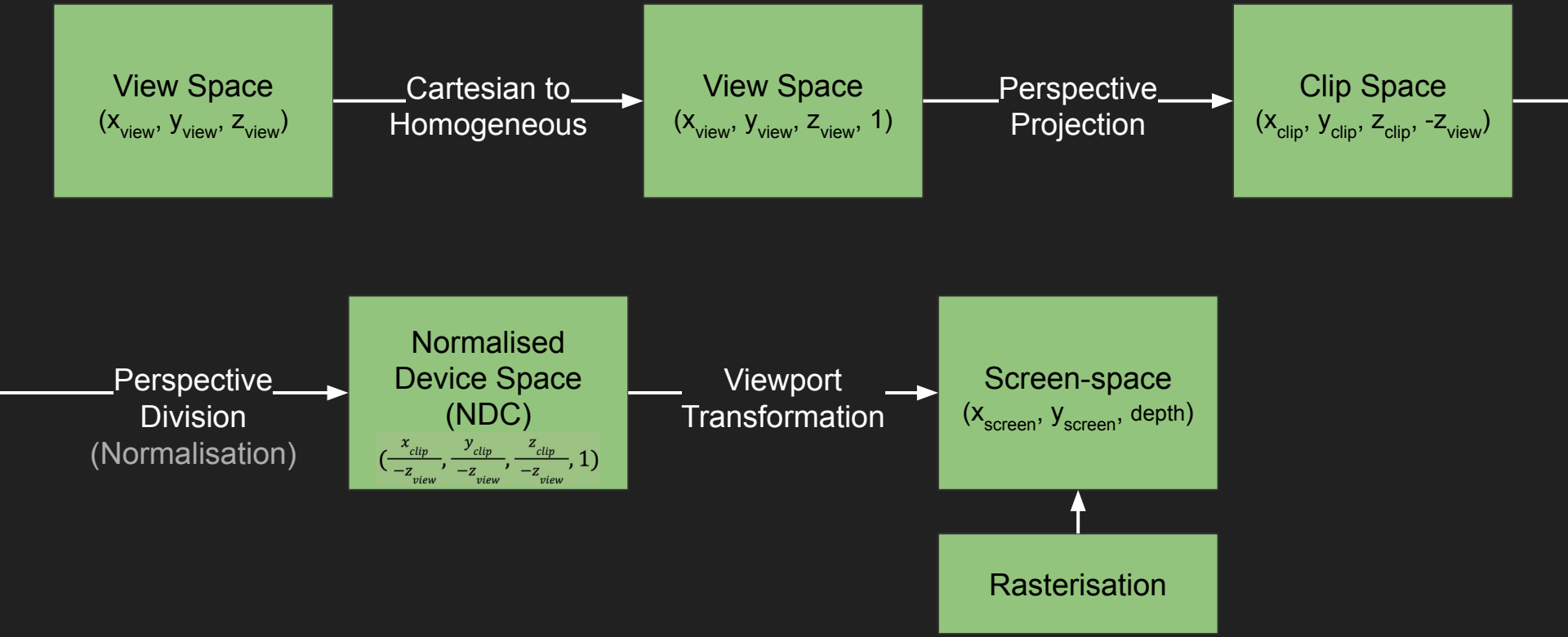
[1]: JIT: Just-in-Time; AOT: Ahead-of-Time
[2]: Typing support through community solution jaxtyping: https://github.com/google/jaxtyping
[3]: Roughly estimation, see https://github.com/google/jax/discussions/3478
[4]: Under transformation. https://github.com/google/jax/pull/16335

From 3D Models to Screen Pixels



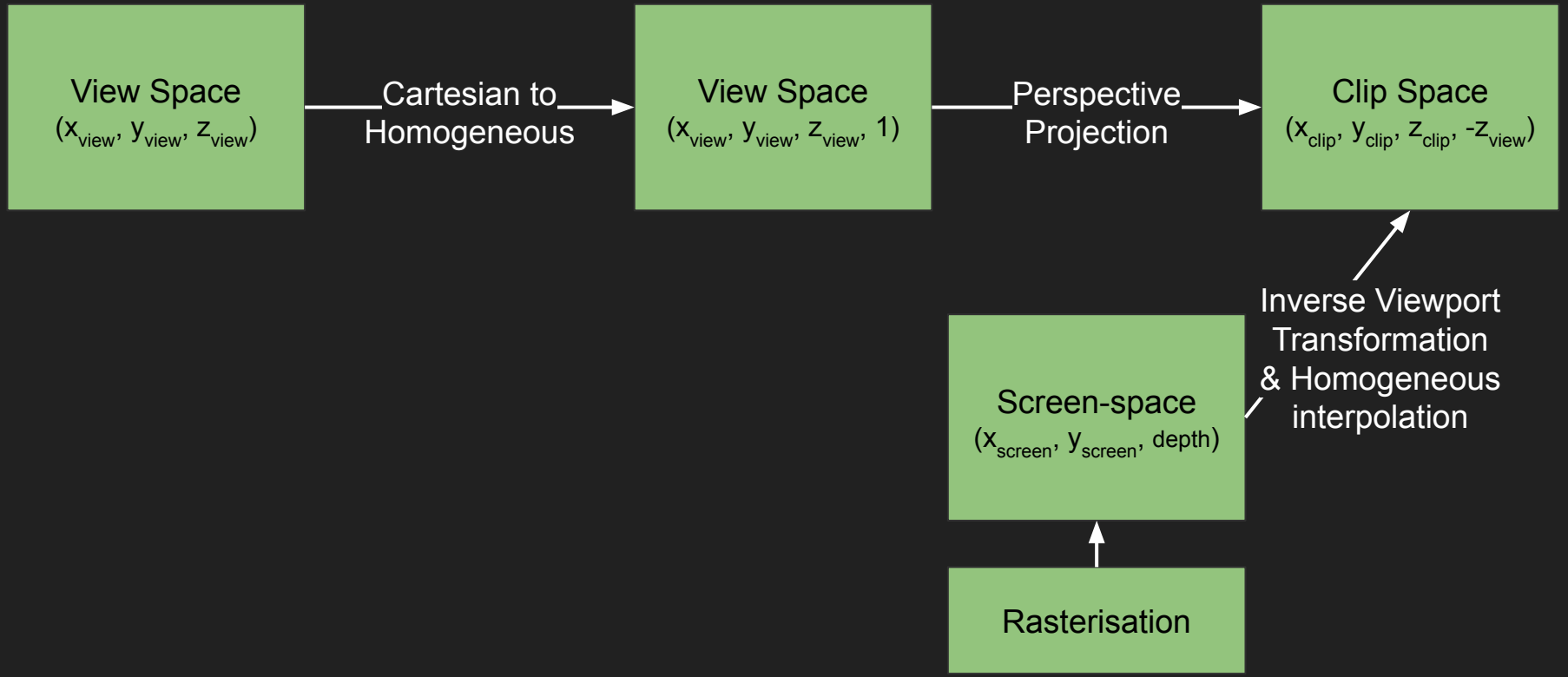
OpenGL Transformation of coordinates from View space to screen space

Typically the coordinates will be transformed from world space to view space via a *view matrix*, but that is defined by user in *vertex shader*, and is not part of OpenGL specification.



OpenGL Transformation of coordinates from View space to screen space

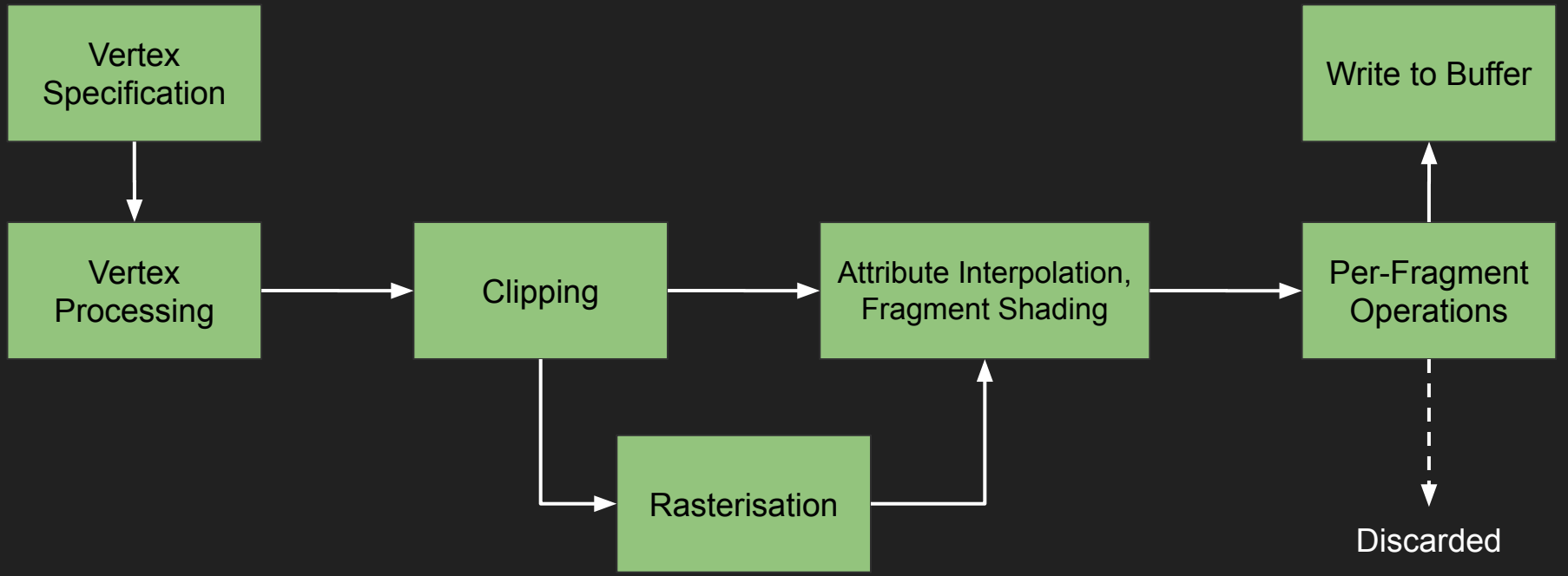
Typically the coordinates will be transformed from world space to view space via a *view matrix*, but that is defined by user in *vertex shader*, and is not part of OpenGL specification.



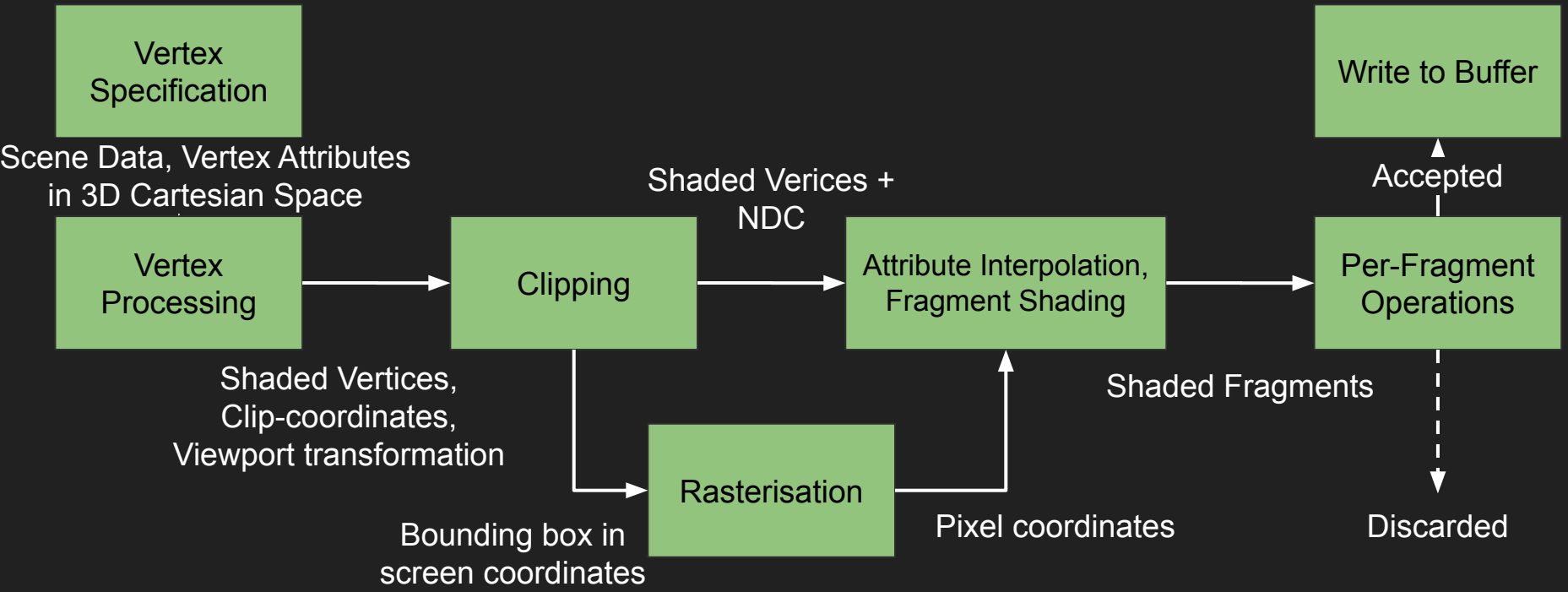
Our Transformation of coordinates

Convert pixels into 3D space to interpolate

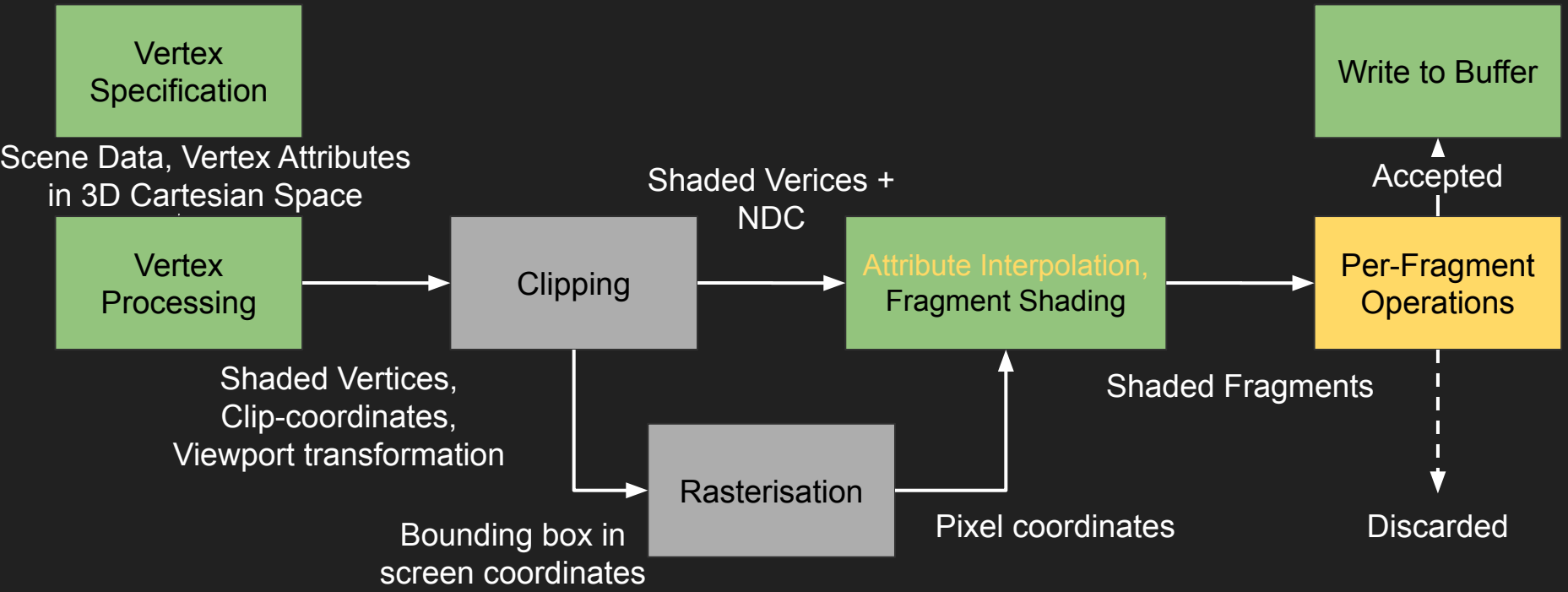
Graphics Pipeline (OpenGL)



Overview of OpenGL Pipeline (Compulsory Stages)

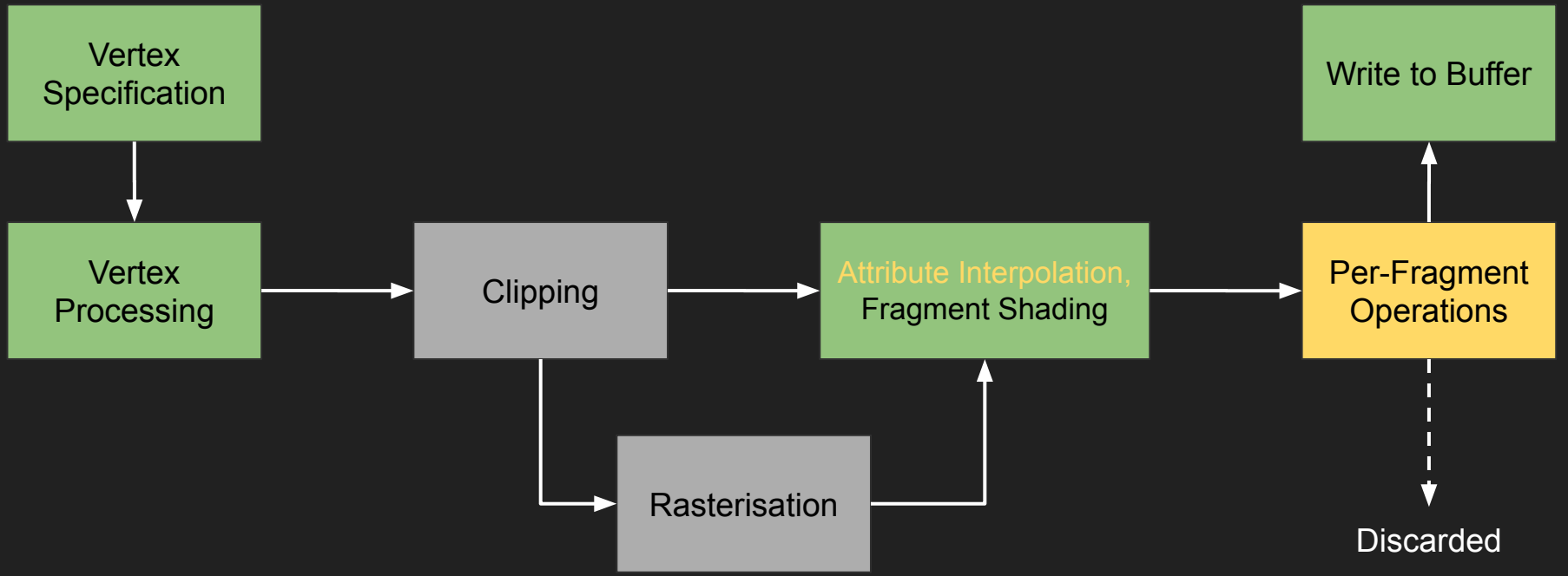


Overview of OpenGL Pipeline with Data Flow



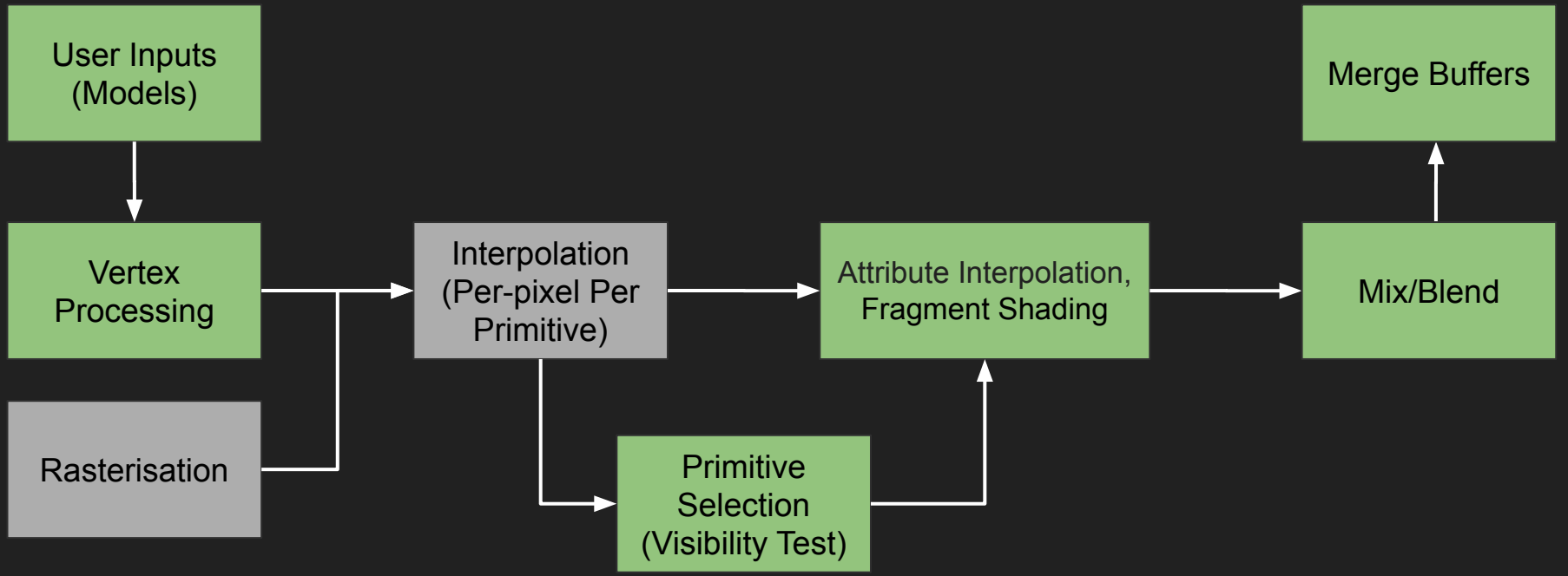
User-defined
Customisable by choosing between given options
Fixed, not customisable

Overview of OpenGL Pipeline with Data Flow



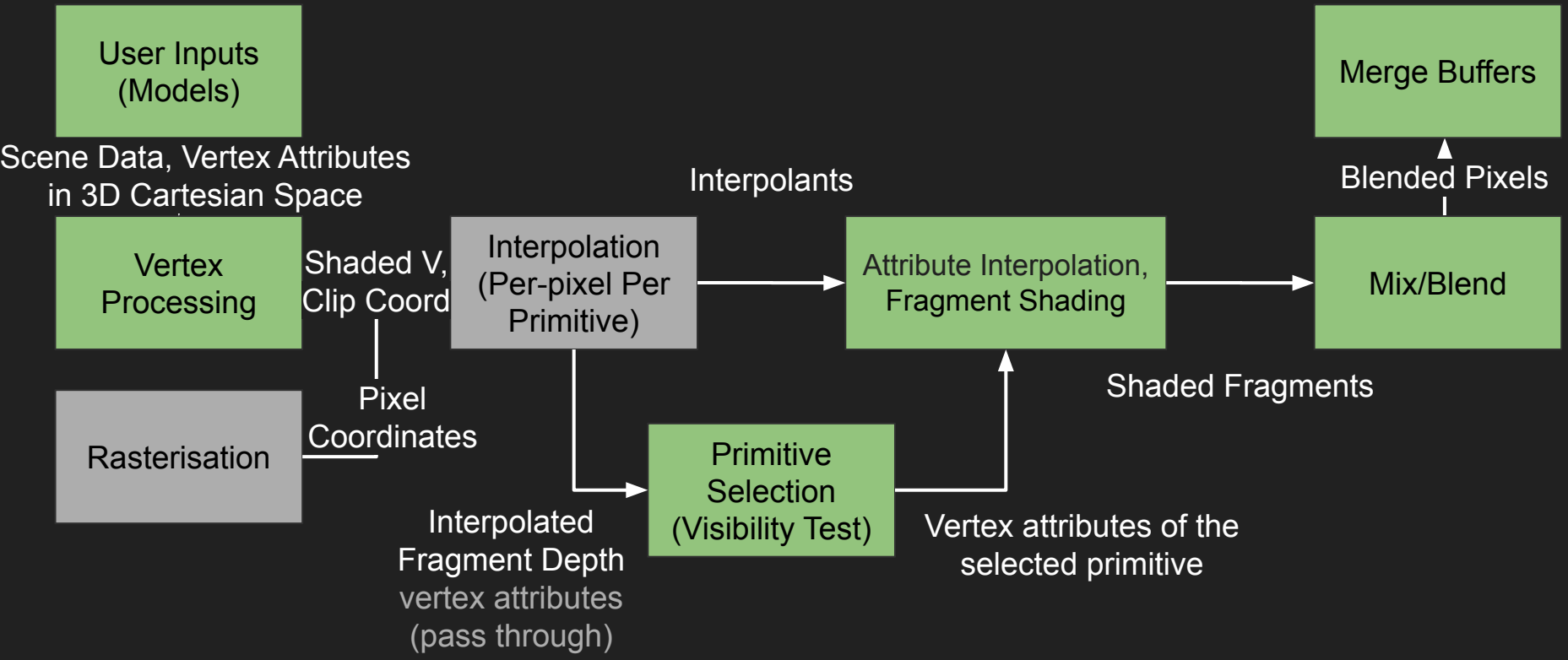
User-defined
Customisable by choosing between given options
Fixed, not customisable

Overview of OpenGL Pipeline (Compulsory Stages)



Overview of JaxRenderer Pipeline

User-defined with default given
Fixed, not customisable



Overview of JaxRenderer Pipeline

User-defined with default given
Fixed, not customisable

Changes Made

Ours

1. Use homogeneous interpolation + customisable interpolation shader

OpenGL

1. Perspective-corrected screen-space interpolation & clip stage + interpolation keywords

Changes Made

Ours

1. Use homogeneous interpolation + customisable interpolation shader

OpenGL

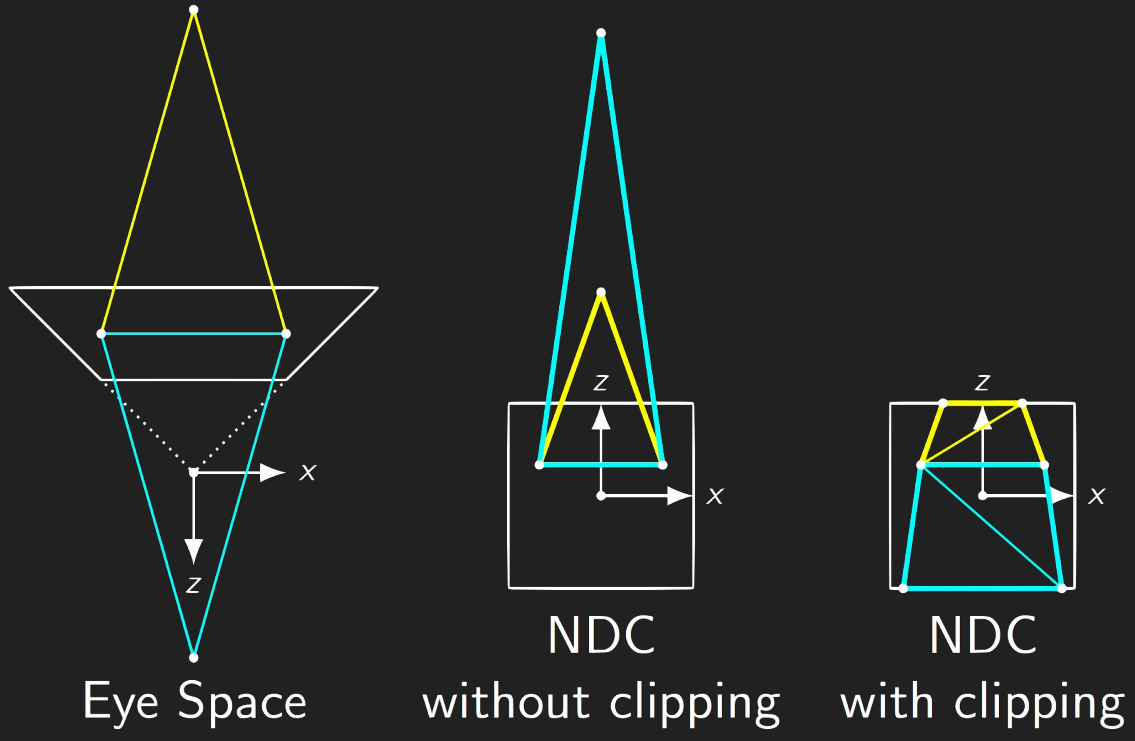
1. Perspective-corrected screen-space interpolation & clip stage + interpolation keywords

Why can we skip clipping stage?

Clipping

- Necessary for the correctness of perspective division (normalise homogeneous coordinates)

- Perspective projection:
- World space => Eye (clip) space
- The homogeneous coordinates must be normalised to extract the equivalent 3D cartesian coordinates



Triangles with vertex behind camera will have the sign of *z* flipped during *perspective division* (without clipping). Clipping solves this issue

Reference: https://zhuanlan.zhihu.com/p/102758967

Clipping

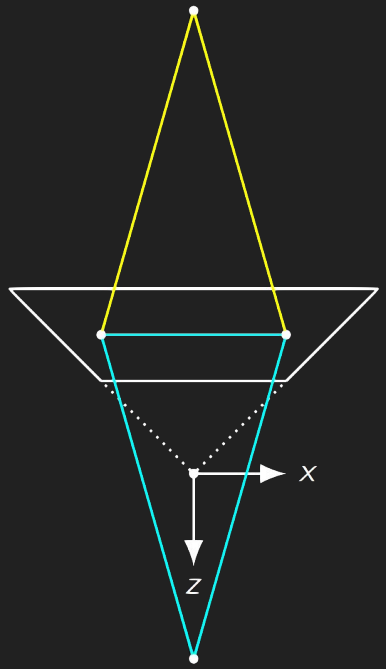
- Necessary for the correctness of perspective division (normalise homogeneous coordinates)

- Perspective projection:
- World space => Eye (clip) space
- The homogeneous coordinates must be normalised to extract the equivalent 3D cartesian coordinates

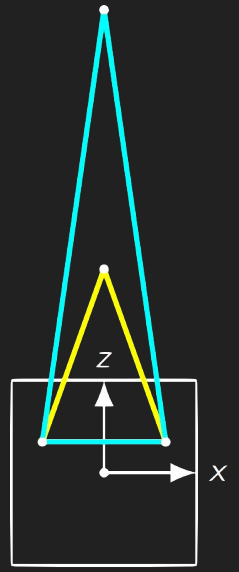
$$(x_{view}, y_{view}, z_{view}, 1)$$

$$\rightarrow (x_{clip}, y_{clip}, z_{clip}, -z_{view})$$

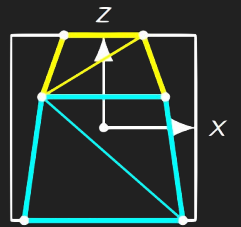
$$\rightarrow \left(\frac{x_{clip}}{-z_{view}}, \frac{y_{clip}}{-z_{view}}, \frac{z_{clip}}{-z_{view}}, 1\right)$$



Eye Space



NDC
without clipping



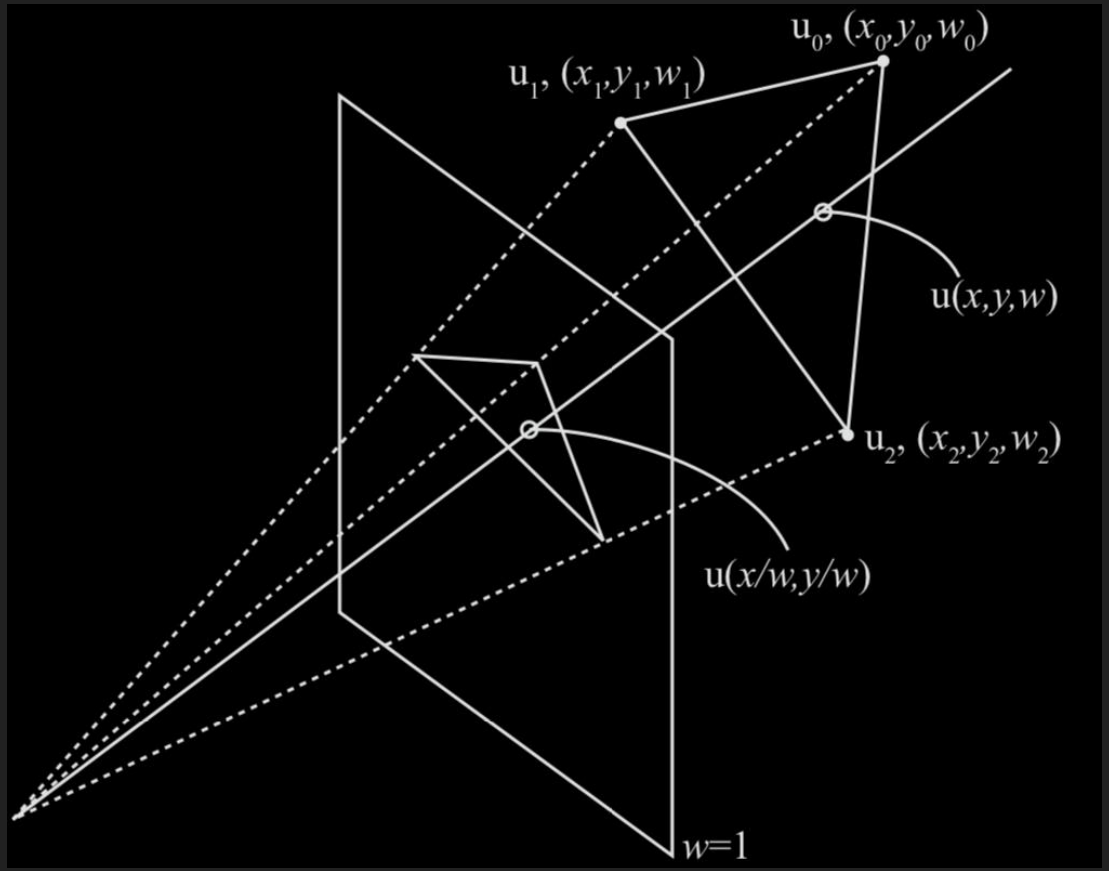
NDC
with clipping

Triangles with vertex behind camera will have the sign of *z* flipped during *perspective division* (without clipping). Clipping solves this issue

Reference: https://zhuanlan.zhihu.com/p/102758967

Homogeneous Interpolation

- Perspective-correct interpolation (interpolating in clip-space)
- Correctly handles all homogeneous coordinates without normalisation
- No clipping stage needed



Reference: https://www.cs.umd.edu/~zwicker/courses/computergraphics/Homogeneous_rasterization.pdf

Changes Made: Summary

Ours

1. Use homogeneous interpolation + customisable interpolation shader
   .
2. Add primitive selection stage to test visibility earlier, across all triangles for each pixel
3. Rasterise on whole canvas for all selected triangles per pixel altogether (static shape)
4. Customisable mixing/blending stage

OpenGL

1. Perspective-corrected screen-space interpolation & clip stage + interpolation keywords
2. Only do z buffering at the end, and has earlier triangles will not be rejected if further away than later shaded triangles
3. Rasterise on each triangle within their bounding box in tiles
4. Carefully design triangle rendering order and blending equation for desired mixing effect

Quick Demo

1. Simple API
2. Customised Shaders
3. Differentiability

Conclusion

Ours

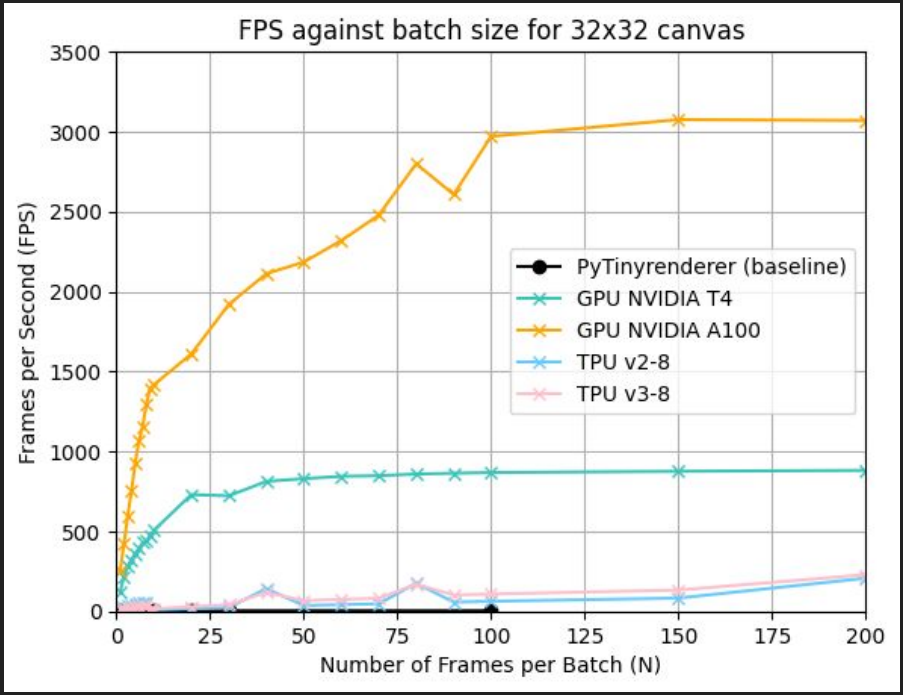
1. Use homogeneous interpolation + customisable interpolation shader
   .
2. Add primitive selection stage to test visibility earlier, across all triangles for each pixel
3. Rasterise on whole canvas for all selected triangles per pixel altogether
4. Customisable mixing/blending stage

OpenGL

1. Perspective-corrected screen-space interpolation & clip stage + interpolation keywords
2. Only do z buffering at the end, and has earlier triangles will not be rejected if further away than later shaded triangles
3. Rasterise on each triangle within their bounding box in tiles
4. Carefully design triangle rendering order and blending equation for desired mixing effect

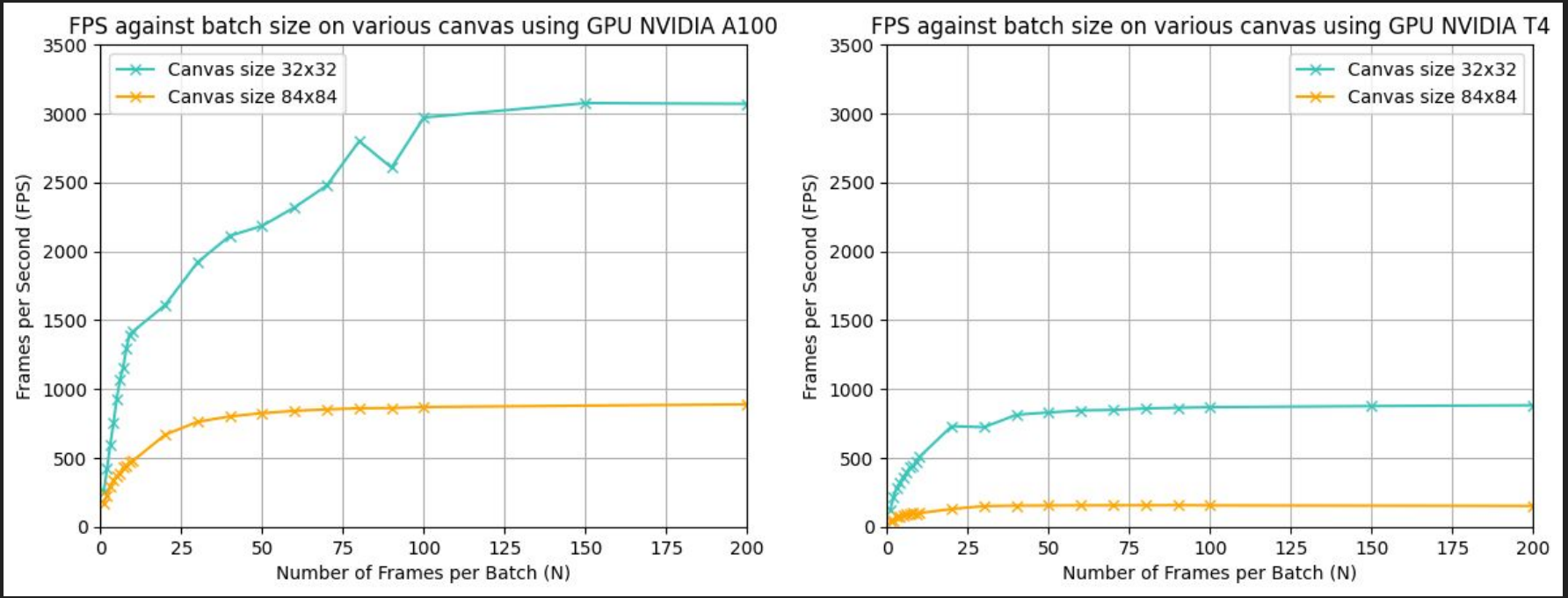
1. Batch Rendering (higher throughput)
2. More flexible, more convenient API
3. Differentiable Rendering
4. XPU support (TPU, GPU (CUDA, ROCm), Apple Silicon)

Performance: Batch Size (Ant, 3276 Triangles)



Far exceeds baseline (CPU rendering) throughput growth as batch size grow

Performance: Canvas Size (Ant, 3276 Triangles)



6.8x more pixels, 1/3 throughput

Large batch size is more meaningful with smaller canvas

Phong Reflection Model + Shadow, 30 frames 1920 x 1080, 2492 triangles, in 9.25s.

Thank you!

Backup Slides

JAX: Vectorisation vs Parallelisation

- Ⓥ Lowering the vectorisation into each lowest op (operator), then compile
- Ⓥ Execute in single device
- Ⓥ Do not support input with different shapes along batch axis
- Ⓥ Input batch size will be lowered to the op's batch size, may lead to unexpected high memory usage for intermediate values

- Ⓟ Replicate the compiled program to all devices, then execute them together, act in SPMD (Single-Program-Multiple-Data) pattern.
- Ⓟ Execute in multiple devices in parallel
- Ⓟ Input must be of same shape across devices
- Ⓟ Input batch axis size must be a multiple of the number of devices
- Ⓟ Devices must be homogeneous

Reference: Documentation of `pmap`: https://jax.readthedocs.io/en/latest/_autosummary/jax.pmap.html