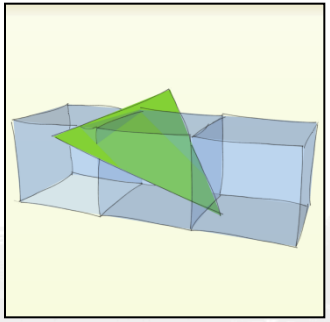
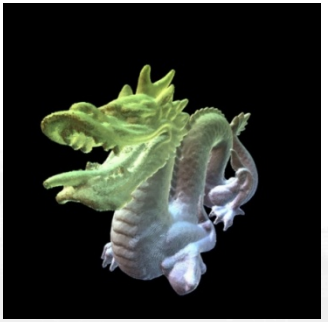
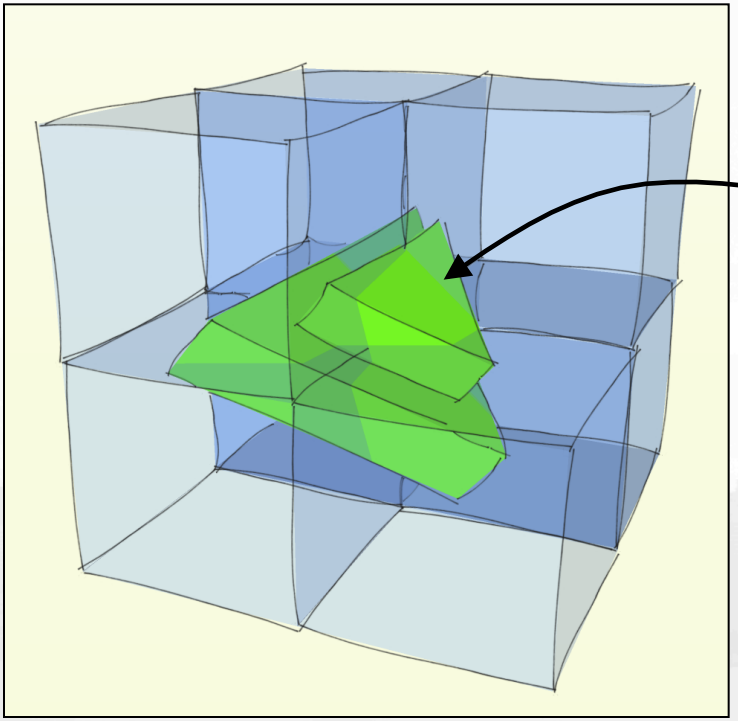
VoxelPipe:
A Programmable Pipeline for 3D Voxelization





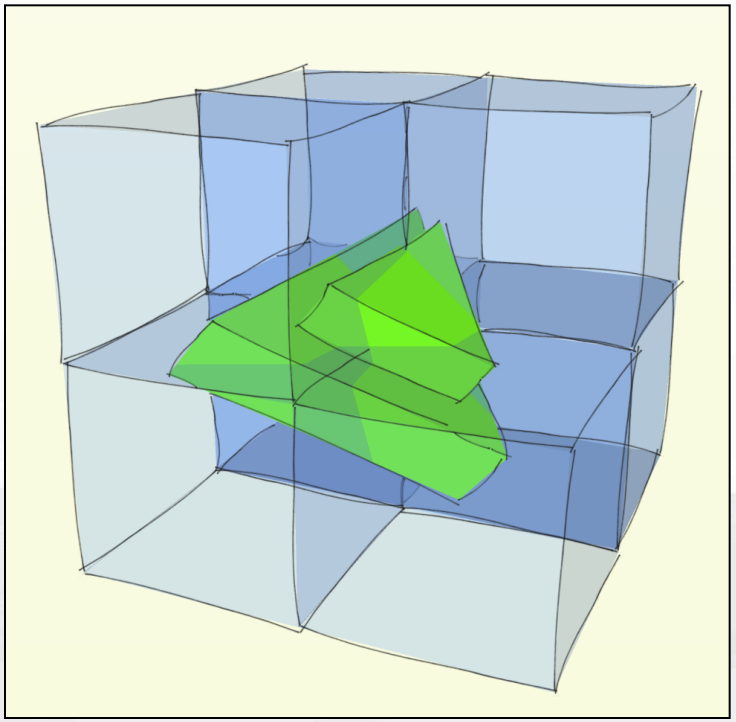
Surface Voxelization



Voxelization =

Finding all voxels overlapped by each triangle in a mesh

Surface Voxelization



Why is it useful?

- Shape Matching
- Collision Detection
- Fluid / Soft-body Sim
- Stress Analysis
- Level of Detail
- Ray Tracing

Surface Voxelization

Interactive Indirect Illumination and Ambient Occlusion using Voxel Cone Tracing

Cyril Crassin

(I3D 2011)

Rationale

building a full-featured pipeline for voxelization,

analogous to OpenGL for 2d rasterization

- fully conservative and *thin** rasterization
- arbitrary frame-buffer types
- many blending modes (additive,max,min,and,or...)
- multiple render targets
- vertex shaders
- fragment shaders

Rationale

Extended support for rendering modes:

- conventional blending-based rasterization
- A-buffer / bucketing

Challenges

- Previous research mostly concerned with binary output [Schwartz and Seidel 2010]

  => no Shading, no ROP

- State-of-the-Art had poor load balancing

  => Huge performance hit for mixed triangle sizes

Challenges

- Previous research mostly concerned with binary output [Schwartz and Seidel 2010]

  => no Shading, no ROP

- State-of-the-Art had poor load balancing

  => Huge performance hit for mixed triangle sizes

Challenges

- Previous research mostly concerned with binary output [Schwartz and Seidel 2010]

  => no Shading, no ROP

- State-of-the-Art had poor load balancing

  => Huge performance hit for mixed triangle sizes

Challenges

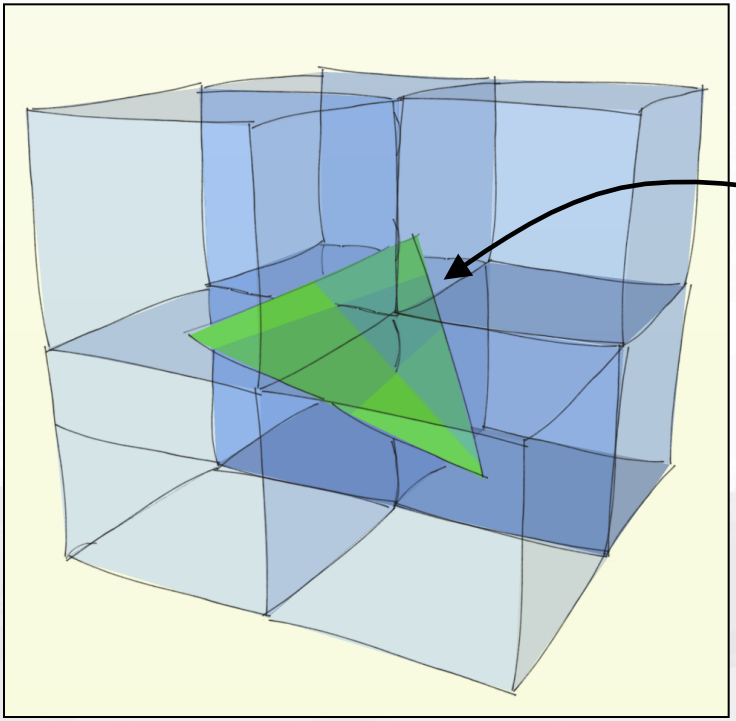
- Previous research mostly concerned with binary output [Schwartz and Seidel 2010]

  => no Shading, no ROP

- State-of-the-Art had poor load balancing

  => Huge performance hit for mixed triangle sizes

The Basics

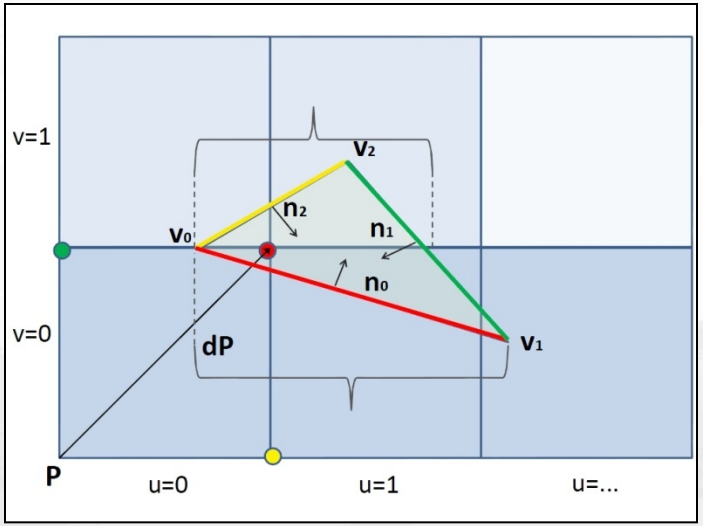


Voxelization =

Finding all voxels overlapped by each triangle

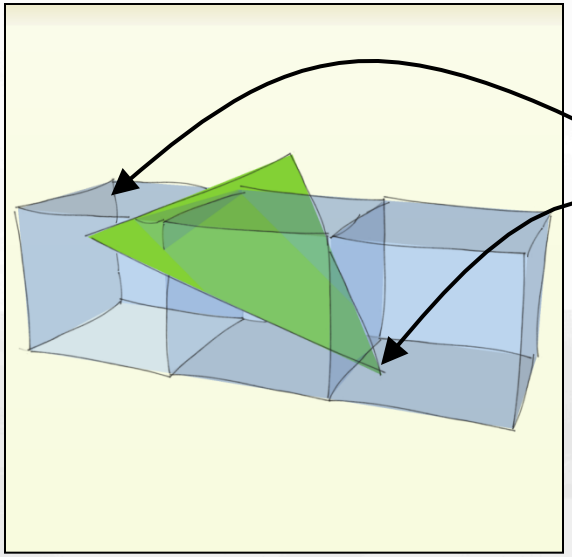
The Basics (2)

projecting along the major axis of the triangle, it becomes **conservative rasterization**



The Basics (3)

projecting along the major axis of the triangle, it becomes **conservative rasterization**

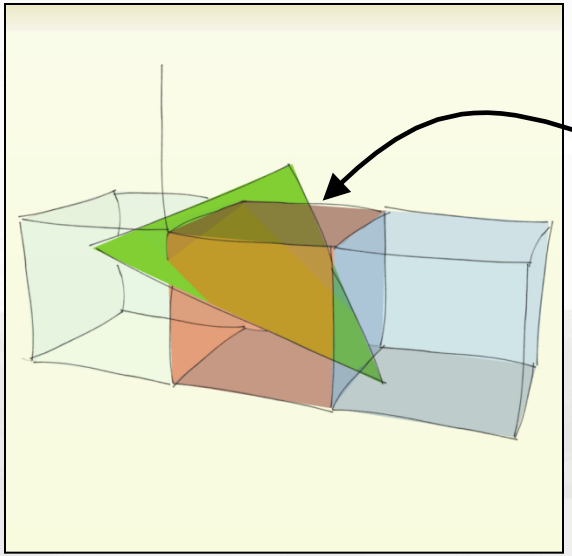


Except we need to find the **column of voxels** spanned by each 2d fragment

=> at most **3!**

The Basics (4)

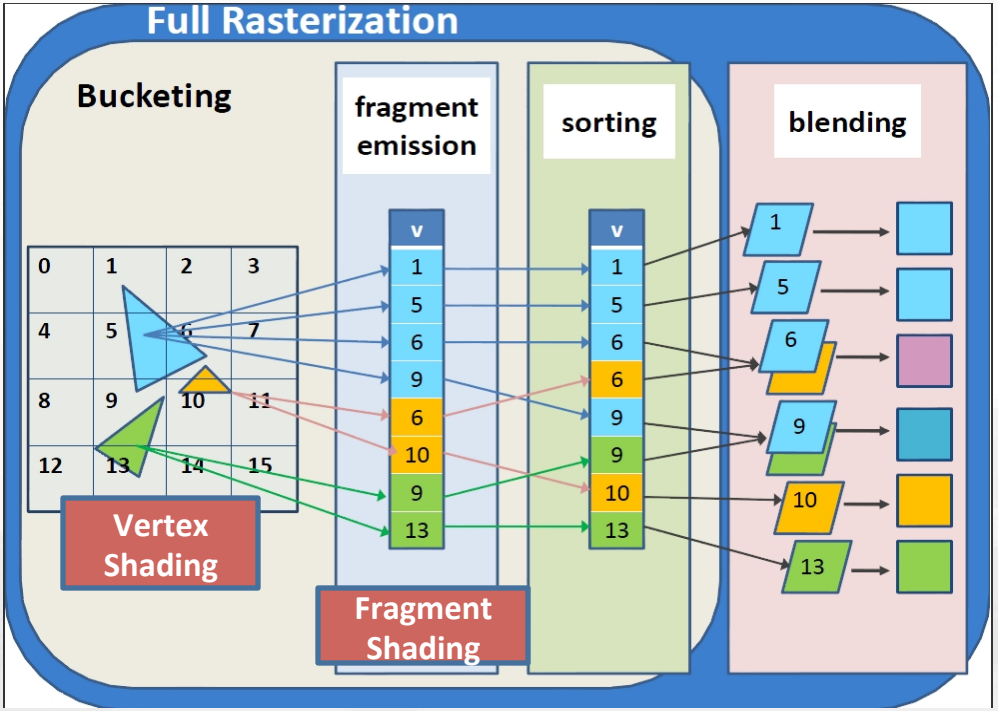
projecting along the major axis of the triangle, it becomes **conservative rasterization**



Or we can select the **single** voxel overlapped by the **fragment ‘s center**

=> **Thin** Voxelization

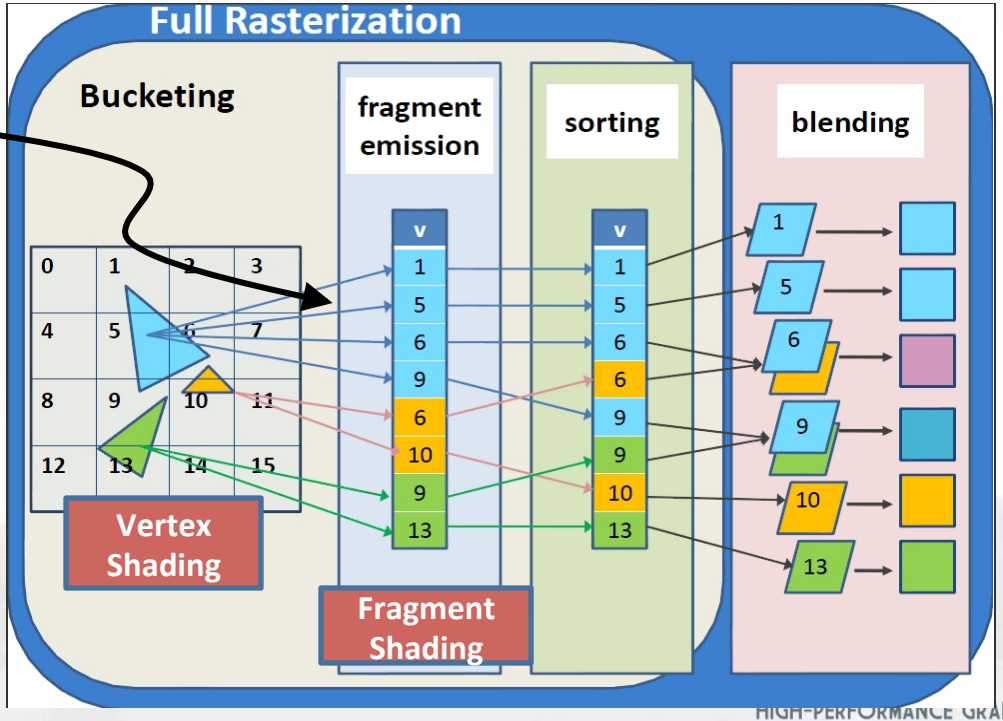
Conceptual View



Conceptual View

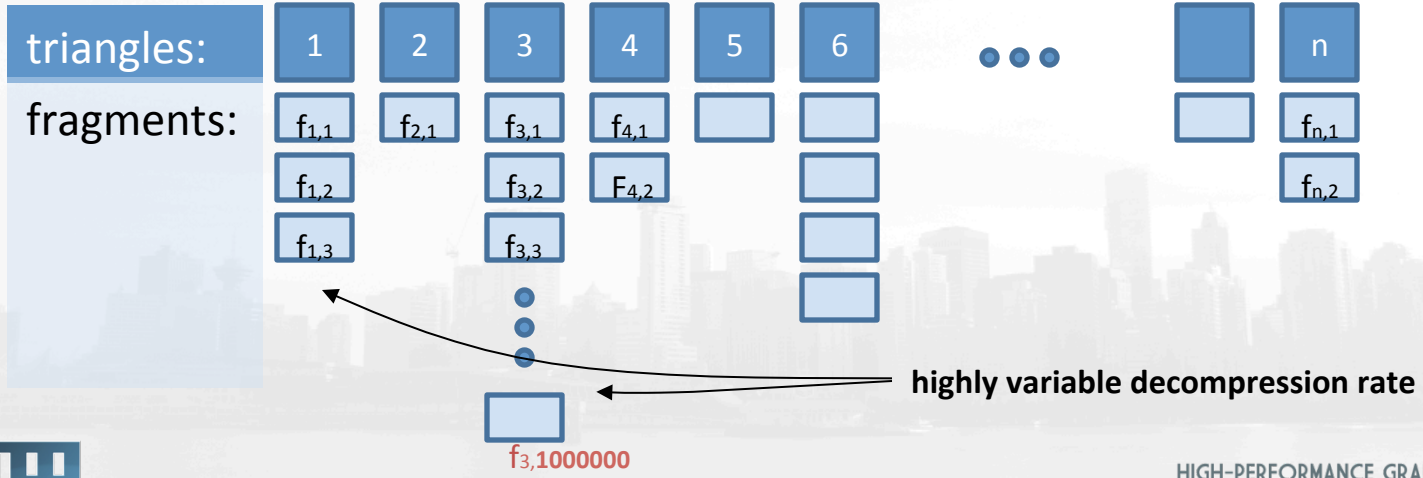
Highly Variable Expansion Rate

source of most load balancing problems



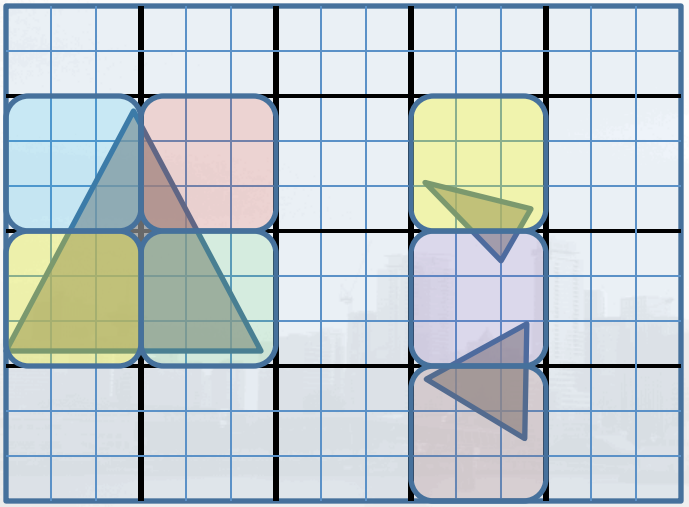
Observations (1)

1. Rasterization =
   Sorting of **Compressed** Batches of Elements



Observations (2)

2. Decompression and Sorting can be done **Hierarchically**



emit per-tile fragments

↓

sort by tile

↓

emit per-voxel fragments

↓

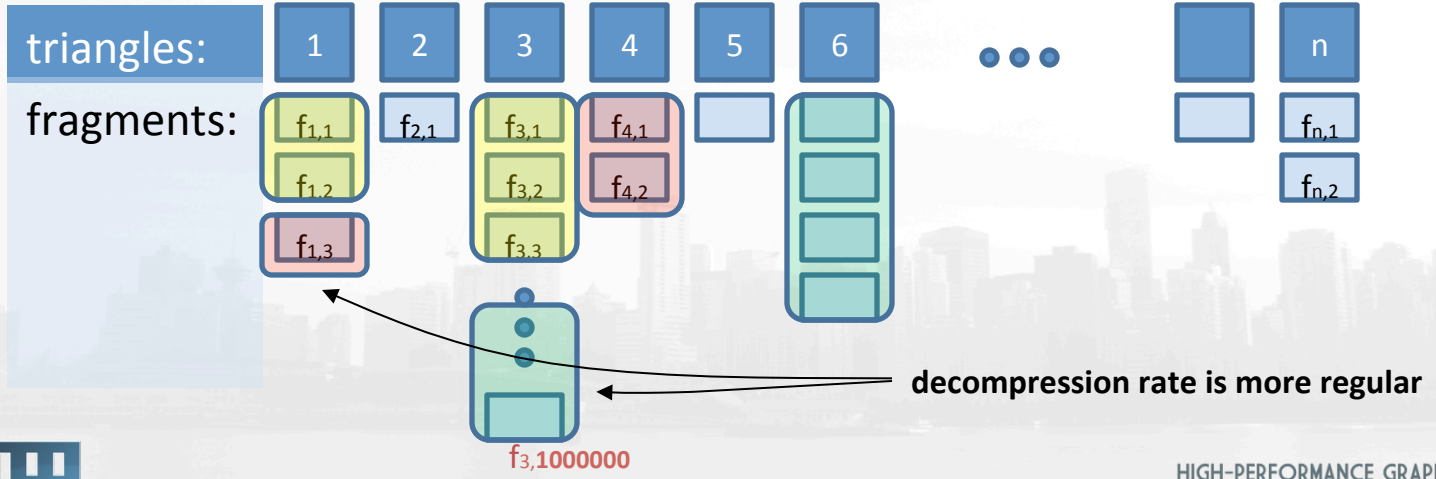
sort by voxel

↓

blend

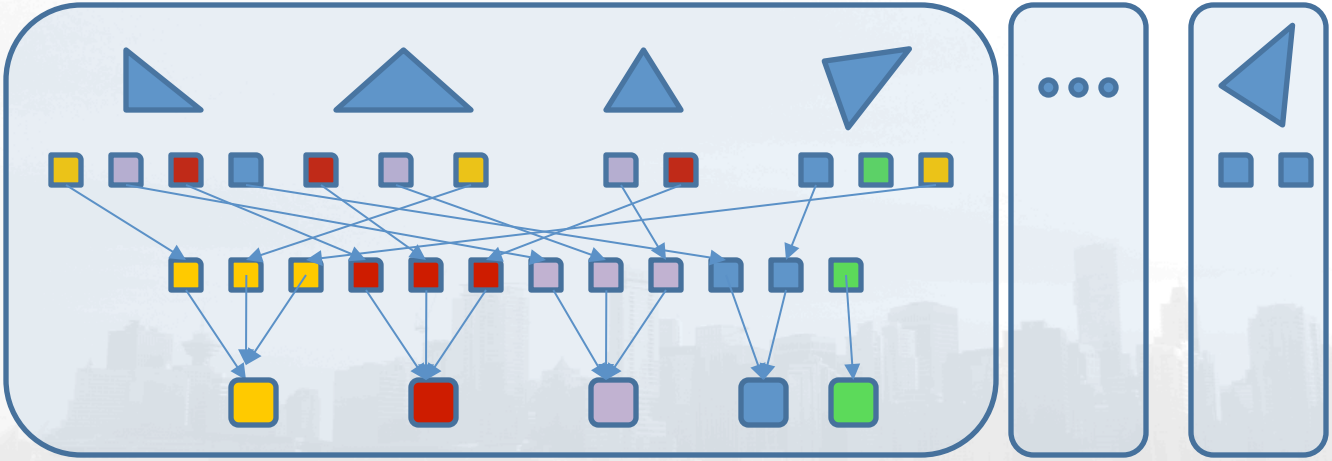
Observations (2)

2. Decompression and Sorting can be done **Hierarchically**



Observations (3)

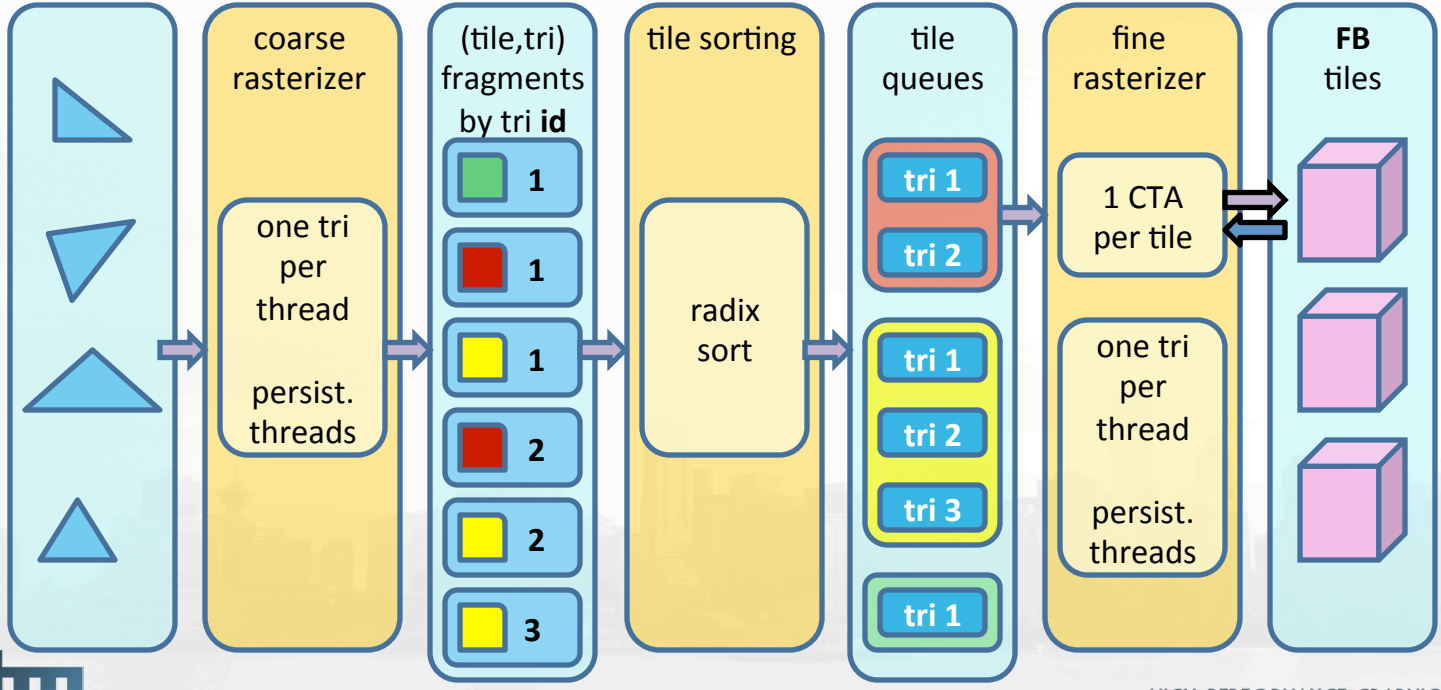
3. Blending allows to perform per-voxel sorting
   in Batches



=> 1. no need to sort every fragment relative to each other

=> 2. data can be consumed right away (e.g. in L1)

Pipeline Overview



Coarse Rasterizer

- Tiles as big as possible, to fit in smem
  (depends on FB format)
- 1 triangle / thread
- emit (tri,descriptor) pairs for all tiles covered by the triangle bbox
- descriptor = (tile_id << 2) + axis(tri)

Tile Sorting

- sort (tri, descriptor) pairs by descriptor => triangles sorted by tile and axis

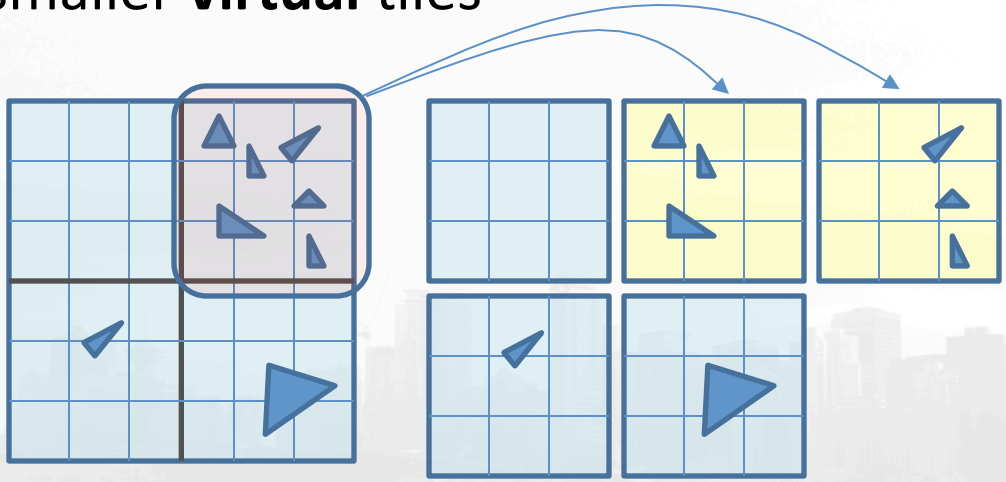
- efficient radix sorting with **log(# tiles)+2** bits

Fine Raster

- smem tiles => blending = smem atomics
- 1 tile / CTA, persistent CTAs
- 1 triangle / thread, persistent warps
- CTA size = F(tile size)

Fine Raster (2)

we break up overloaded **physical** tiles in smaller **virtual** tiles



Vertex & Fragment Programs

simple C++ classes:

```
struct MyShader
{
    T eval(const Fragment frag) const;

private:
    ...
};
```

can be any of the supported types!

Software Architecture

C++ high level **API**

CUDA C++ low level *driver*

Both are *template libraries* (STL-like)

Software Architecture

Deeply based on **Compile-Time Specialization**

i.e.

critical code path = **F<** FB type,
FB size,
BlendingMode,
Vertex Program,
Fragment Program**> ()**

Software Architecture

Deeply based on **Compile-Time Specialization**

⇒ needed **CUDA**,

impossible* to do with **OpenCL**

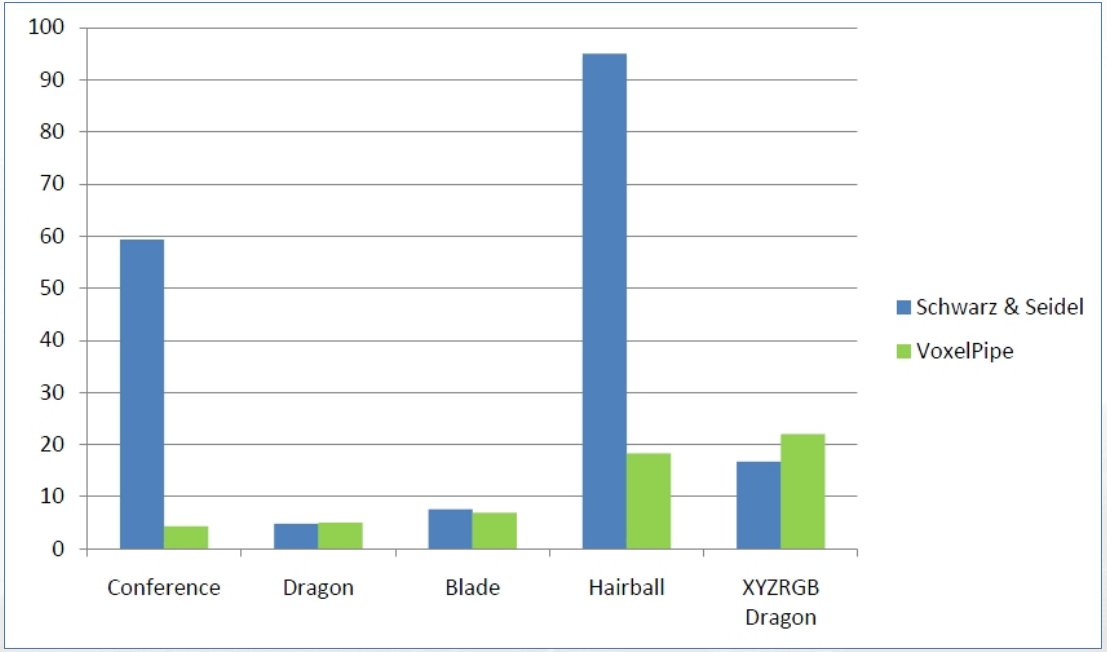
(*) = within bounds of human effort

Some Numbers

Scene	# of Triangles	Grid Res	Schwartz & Seidel	VP Binary	VP Float	VP A-buffer
Conference	282k	128^3	3.9 ms	3.3 ms	3.4 ms	3.8 ms
		512^3	59.3 ms	4.3 ms	8.3 ms	5.3 ms
		1024^3	237.6 ms	8.5 ms	24.0 ms	10.1 ms
Dragon	871k	128^3	3.5 ms	4.8 ms	5.0 ms	6.7 ms
		512^3	4.8 ms	5.0 ms	7.5 ms	8.7 ms
		1024^3	13.6 ms	5.9 ms	13.2 ms	11.6 ms
Turbine Blade	1.76M	128^3	3.6 ms	7.3 ms	7.9 ms	10.3 ms
		512^3	7.6 ms	6.9 ms	10.1 ms	11.6 ms
		1024^3	16.6 ms	8.4 ms	14.9 ms	12.7 ms
Hairball	2.88M	128^3	22.8 ms	12.8 ms	15.3 ms	23.8 ms
		512^3	95.0 ms	18.3 ms	38.9 ms	50.0 ms
		1024^3	266.8 ms	33.7 ms	192.8 ms	102.0 ms
XYZ RGB Asian Dragon	7.21M	128^3	11.4 ms	21.2 ms	26.0 ms	34.8 ms
		512^3	16.7 ms	22.0 ms	29.4 ms	39.9 ms
		1024^3	18.2 ms	23.6 ms	31.4 ms	43.0 ms

Table 1: *Voxelization timings for various scenes and different voxelization schemes. VP stands for VoxelPipe.*

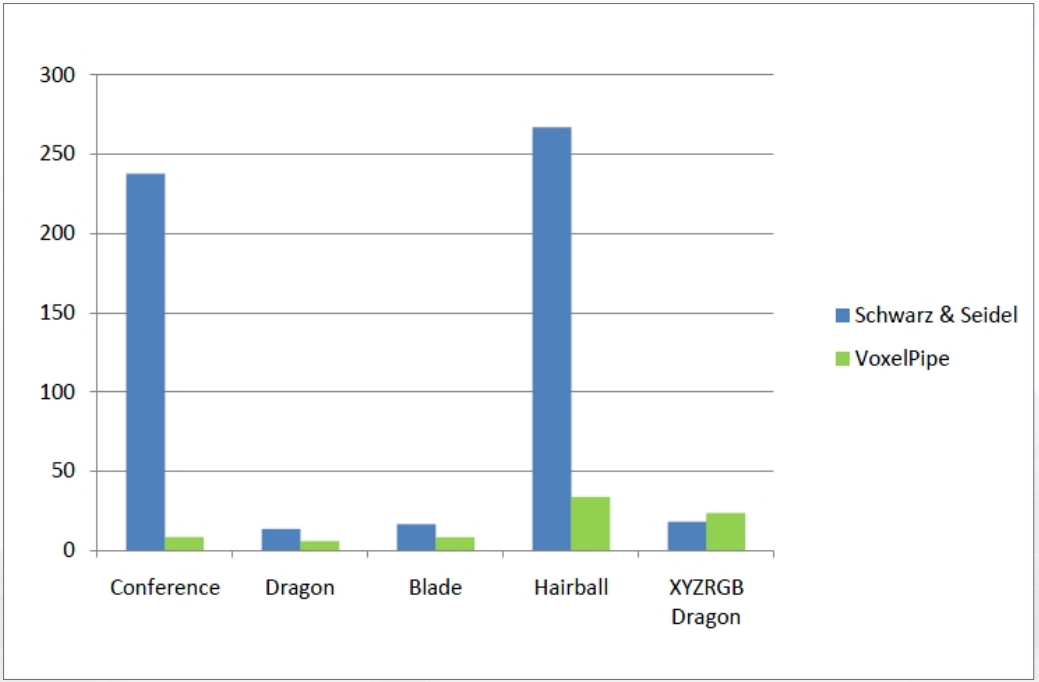
Some Numbers (ms)



binary output

512^3

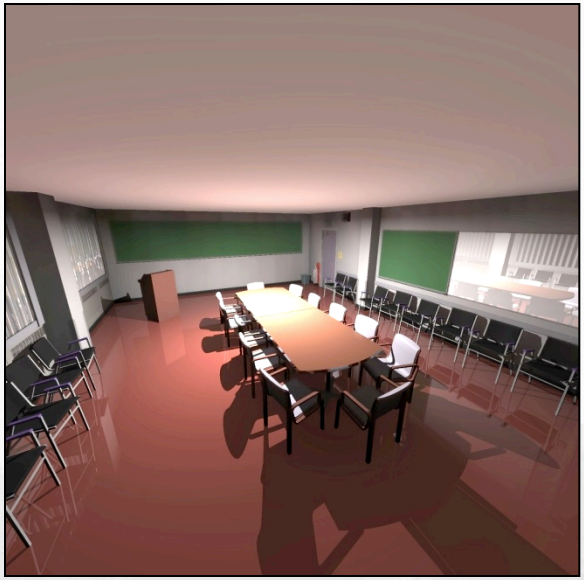
Some Numbers (ms)



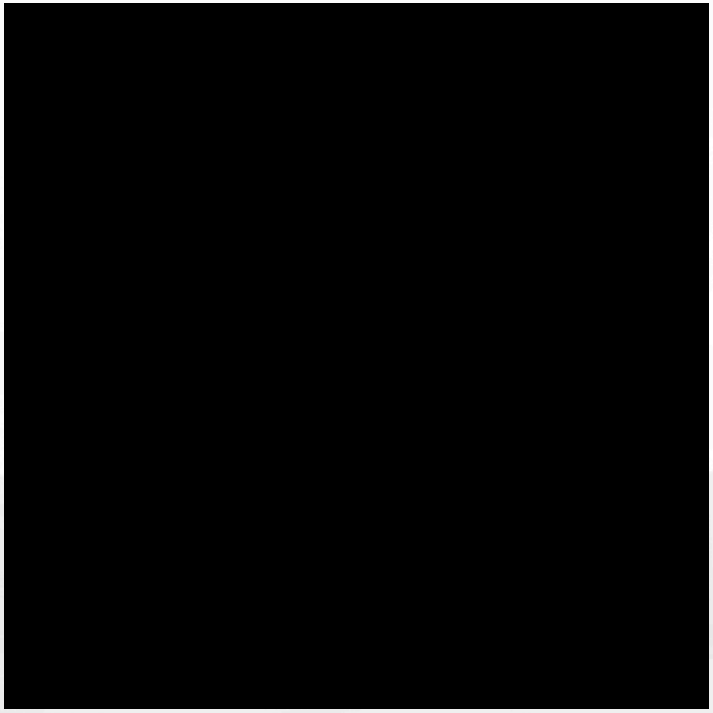
binary output

1024^3

A small app...



A small app...



Future Work

- Sparse Octrees
- Tessellation / Geometry Shaders
- Programmable ROP

VoxelPipe