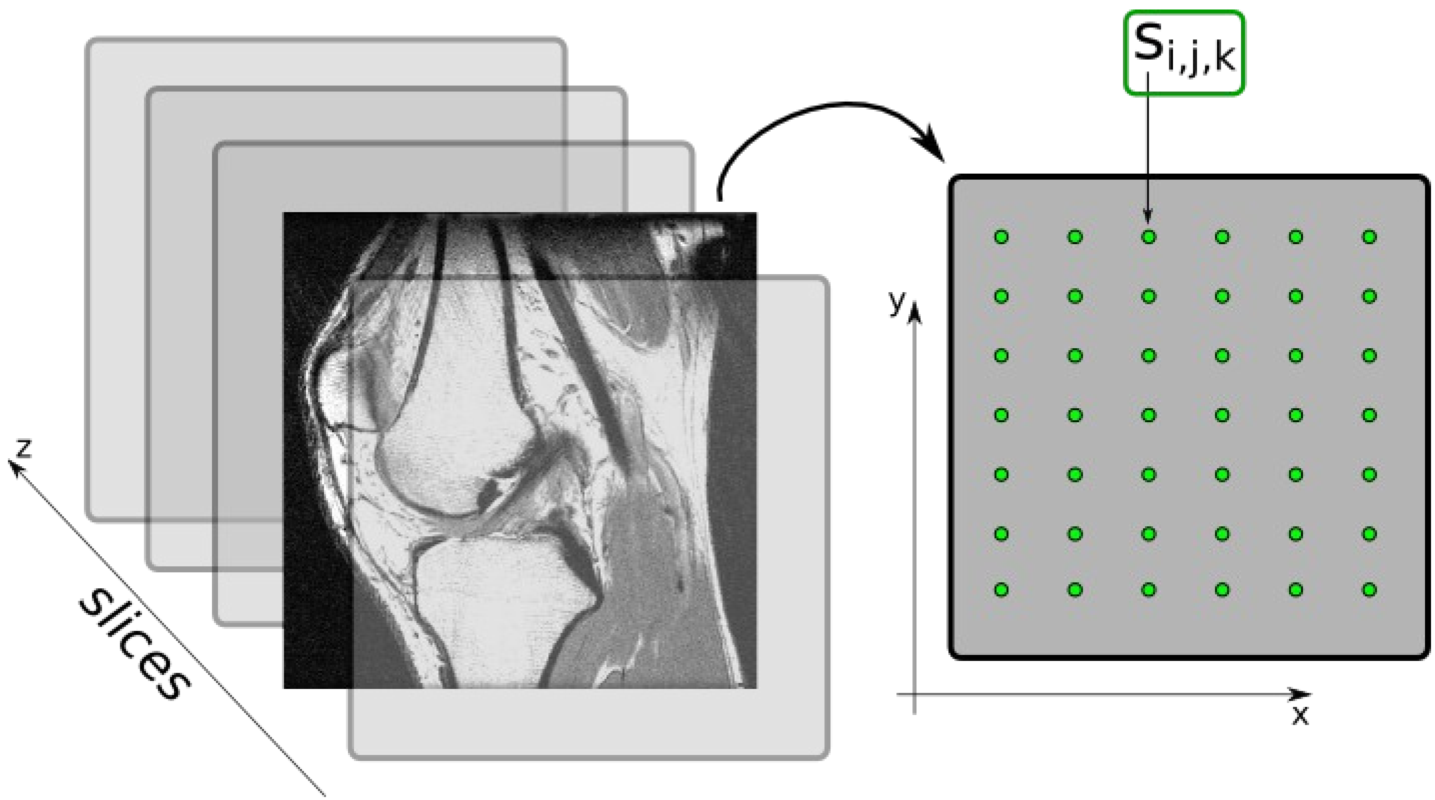
GPU-Based Cell Projection for Interactive Volume Rendering

Ricardo Marroquim
André Maximo
Ricardo Farias
Claudio Esperança

Volume Rendering : Acquisition

- 3D scalar fields:
  - Density, heat, velocity, etc...

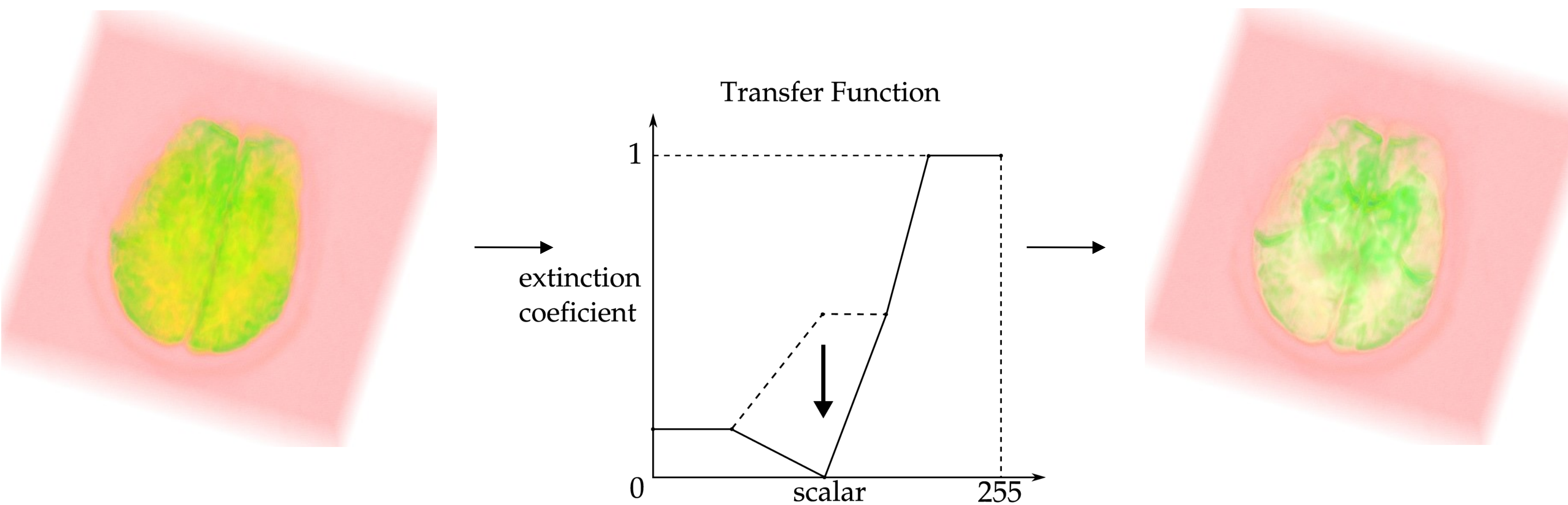


Volume Rendering : Mesh

- Scalar field -> Tetrahedral mesh
- Compose slices in hexagons (4 points of front slice and 4 of back slice)
- Each hexagon can be subdivided in 6 or 5 tets without inserting new points
- Unstructured grids

Transfer Function

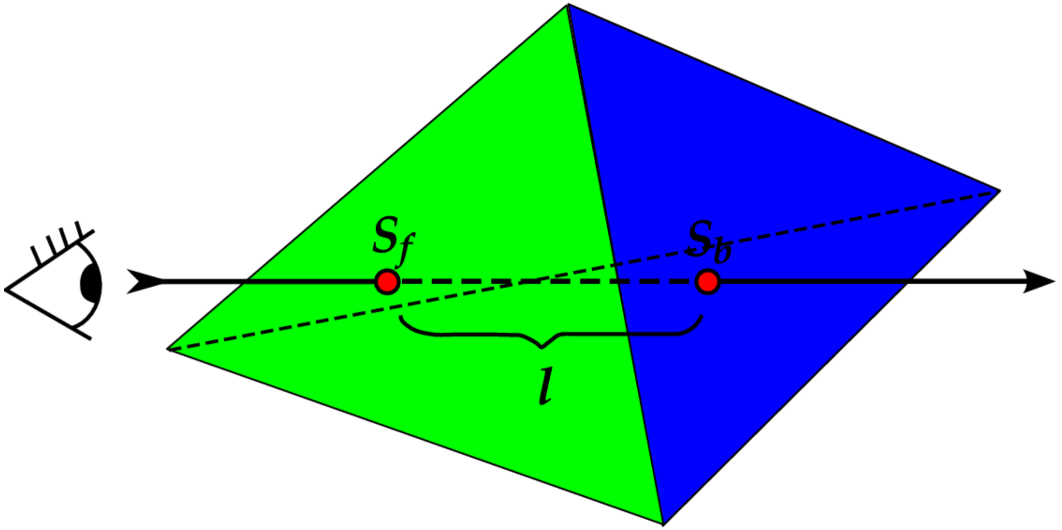
- Maps scalar value to chromacity and opacity values
- Each scalar ranges contains different features



Ray Integration

- Volume Rendering Integration:

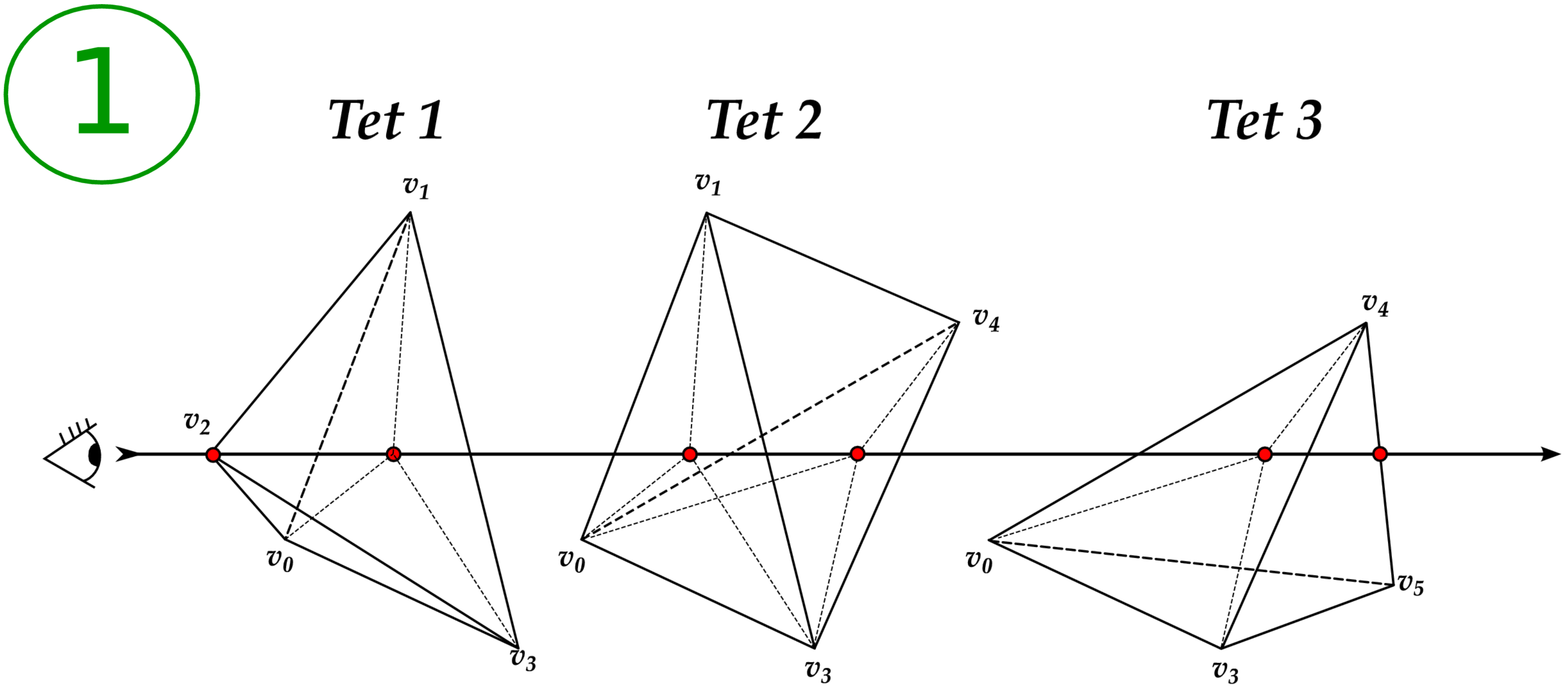
$$I_D = I_0 e^{-\int_0^l \tau(t)\,dt} + \int_0^l k(s)\,\tau(s)\,e^{-\int_s^l \tau(t)\,dt}\,ds$$



Projected Tetrahedra (PT) : Overview

- Introduced by Shirley & Tuchman (1990)
  - Projects tetrahedra to screen space
  - Decompose tetrahedra into triangles
  - Find color and opacity values for the triangles vertices
  - Render in visibility order

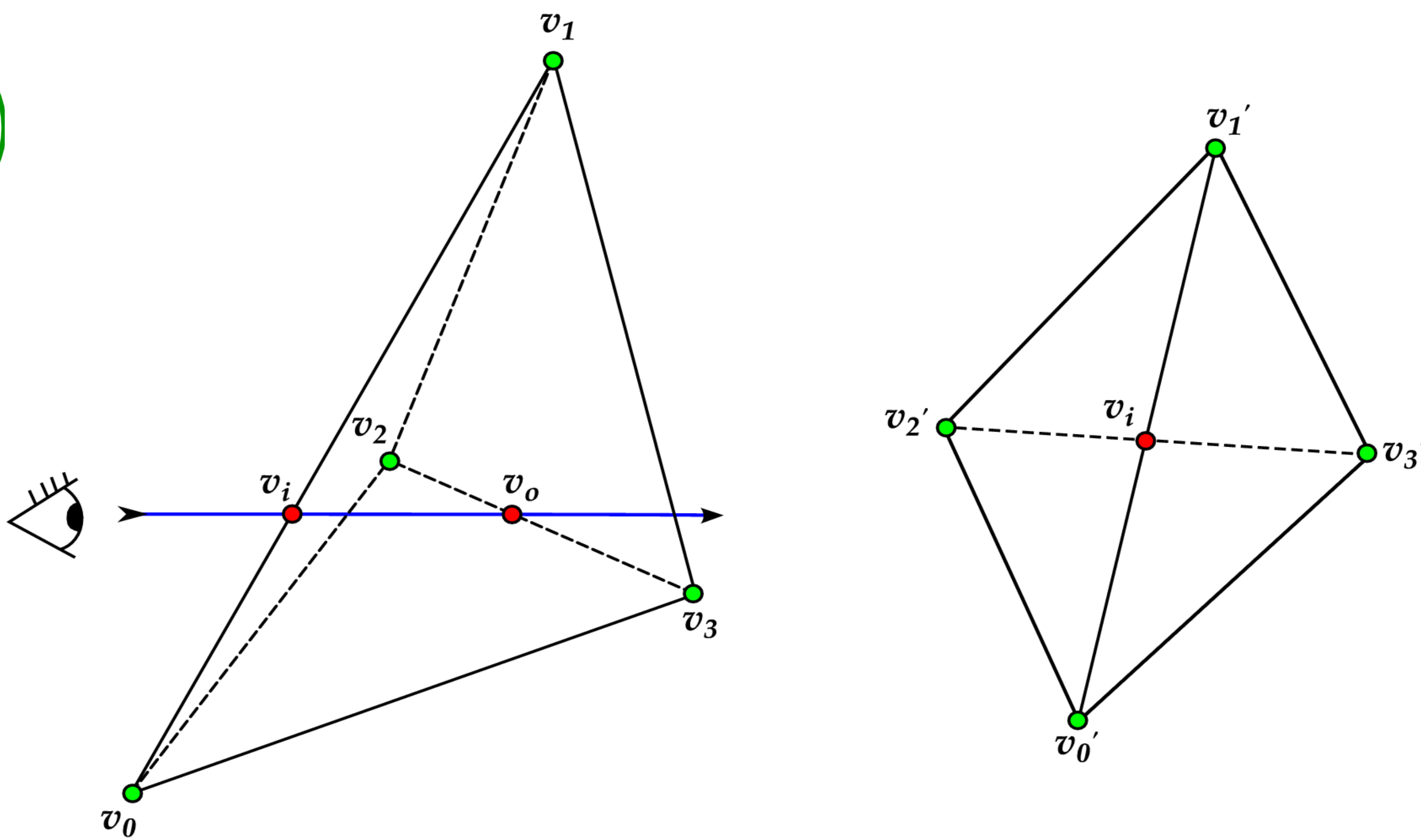
Projected Tetrahedra (PT)



- Sort elements in visibility order

Projected Tetrahedra (PT)

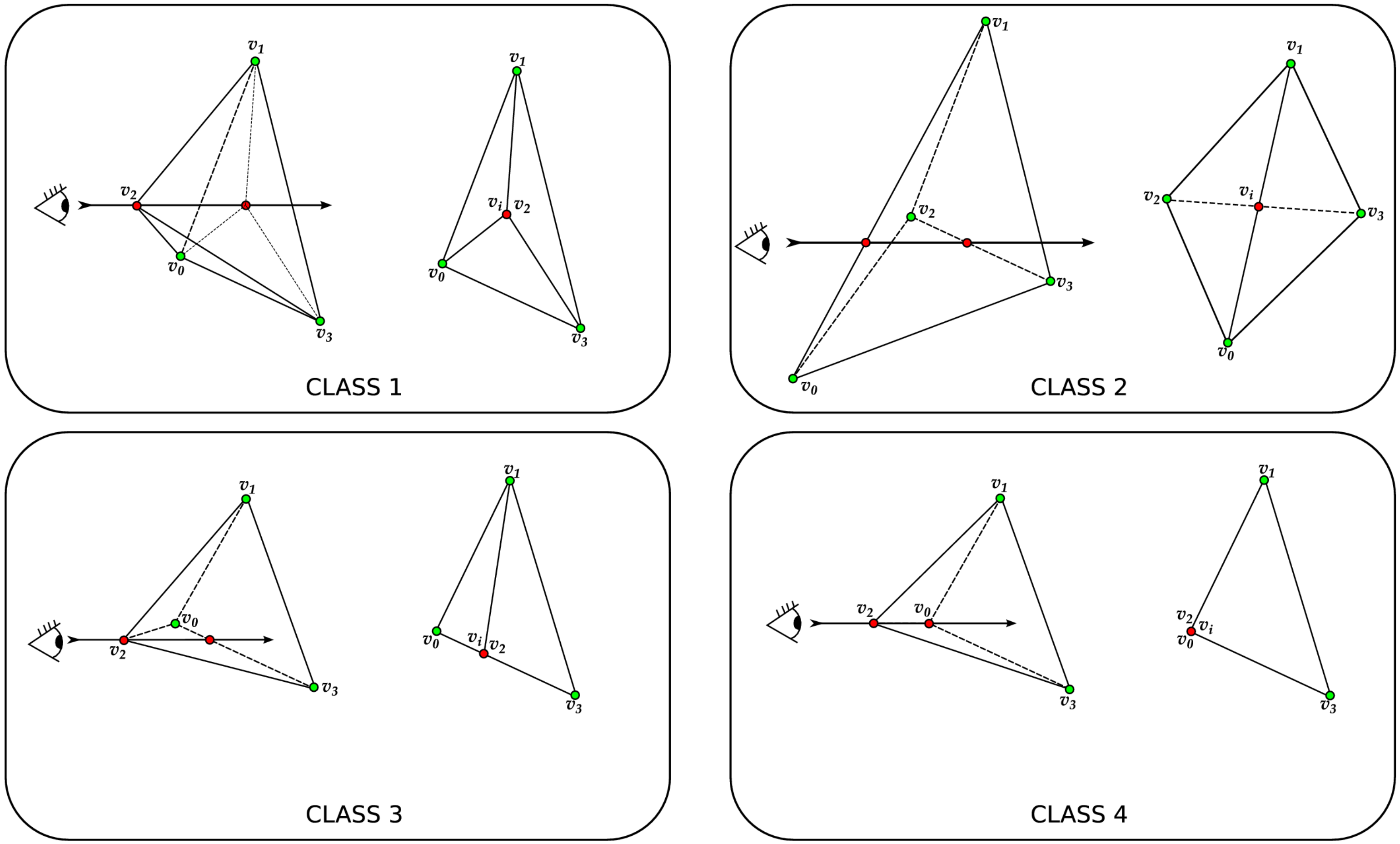
2



- Project tetrahedra to screen space

Projected Tetrahedra (PT)

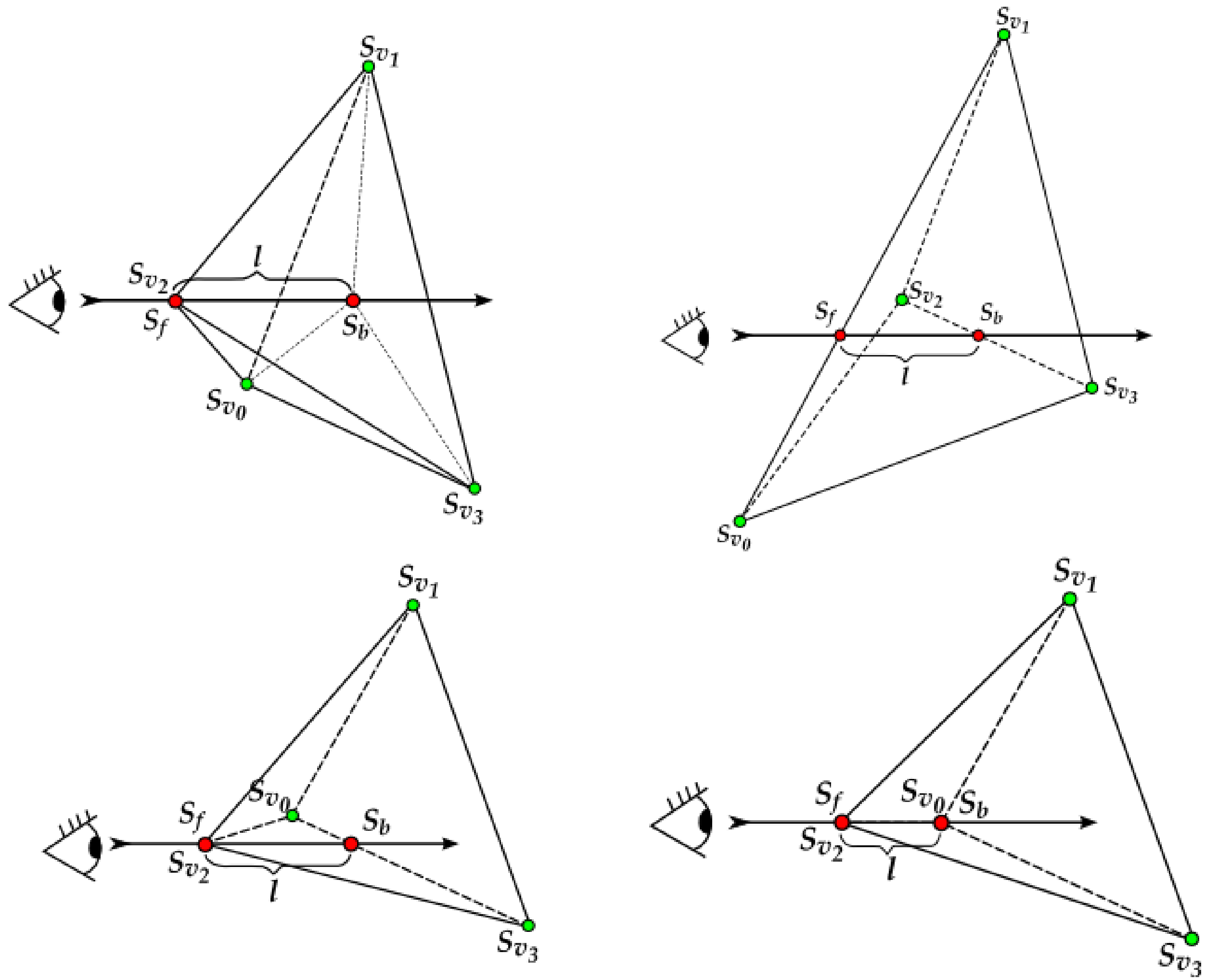
3



- Determine projection class of each tetrahedron

Projected Tetrahedra (PT)

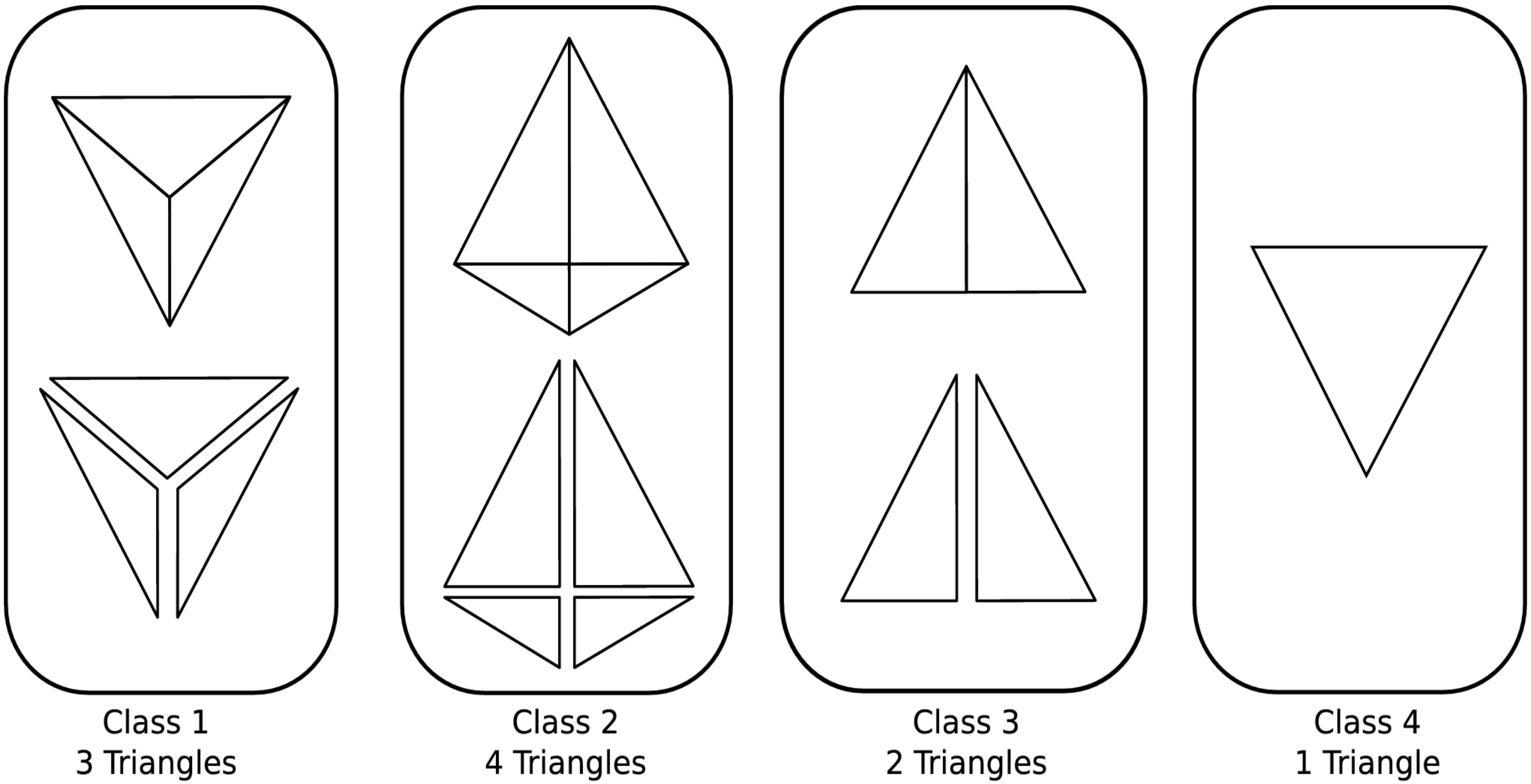
4



- Compute tetrahedron's thickness and scalar value at ray's entry and exit point

Projected Tetrahedra (PT)

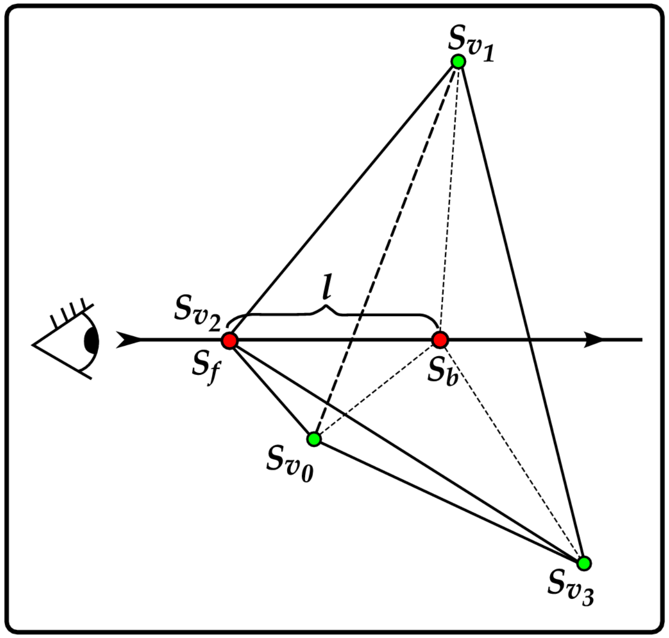
5

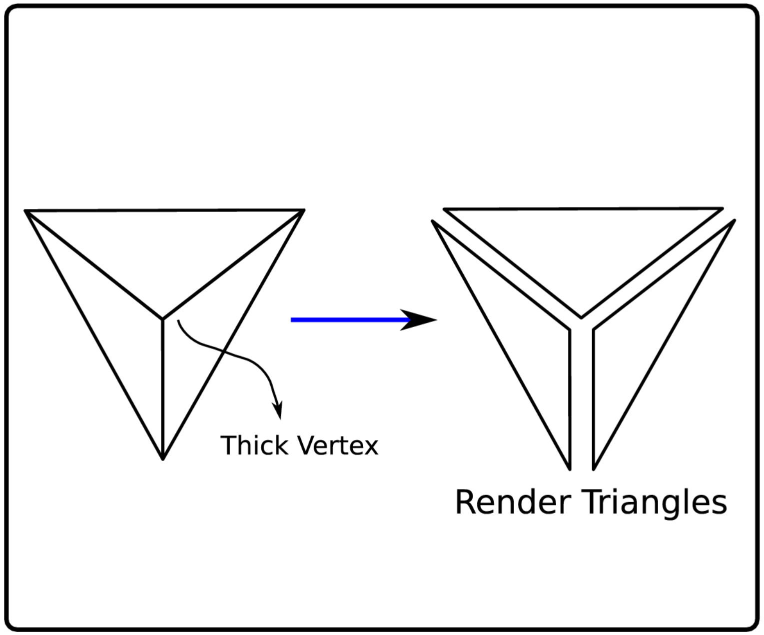


- Decompose projected tetrahedra in triangles

Projected Tetrahedra (PT)

6





- Color and opacity at thick vertex:

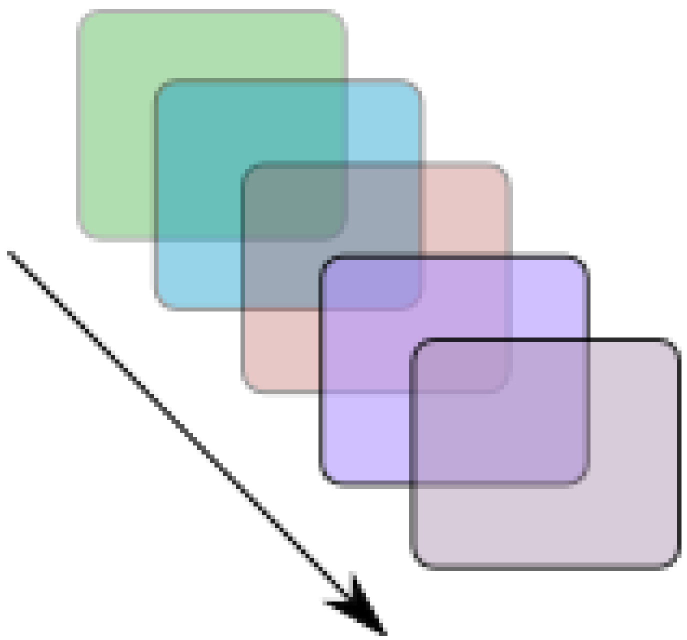
$$v2_{rgb} = \frac{(C_{rgb}(S_f) + C_{rgb}(S_b))}{2}$$

$$\alpha = 1 - e^{-\tau l} \approx \tau l$$

Projected Tetrahedra (PT)

7

$$I_{new} = \alpha * C_{rgb} + (1 - \alpha) * I_{old}$$

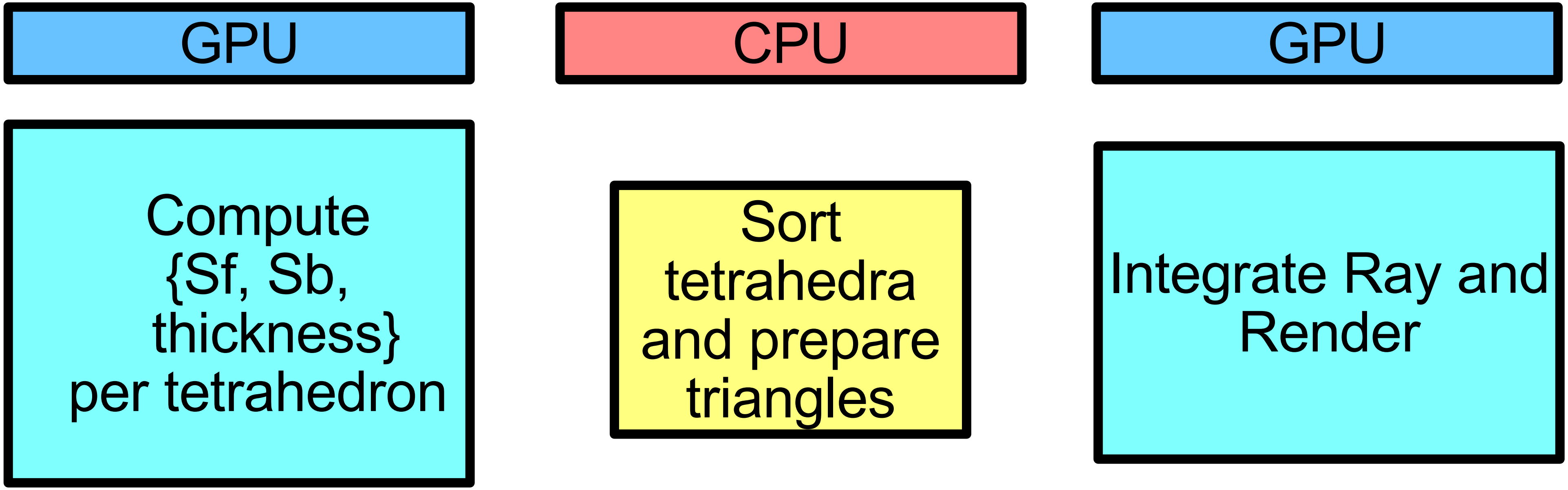


- Compose fragments

GPU Implementation

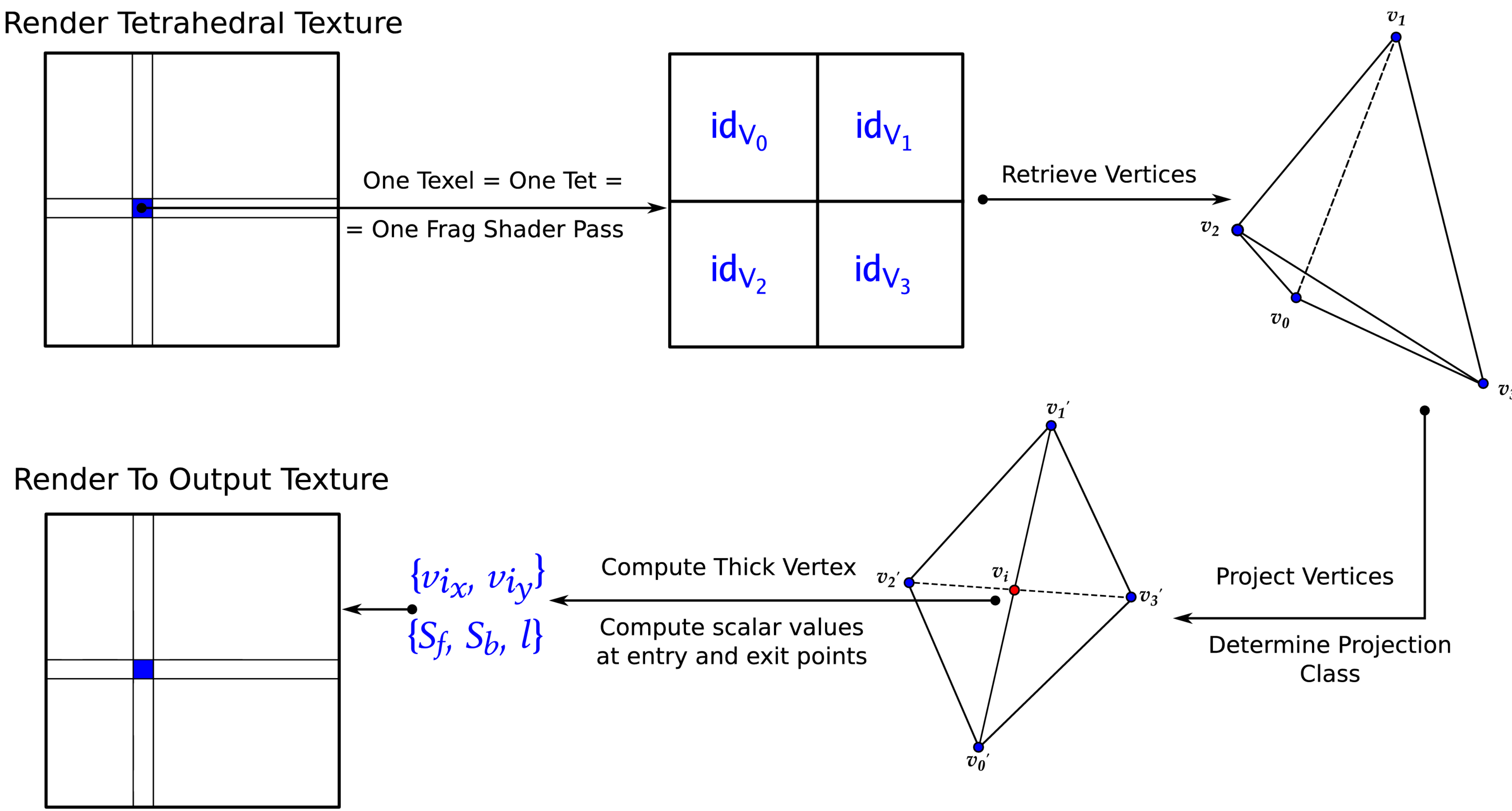
- Based on Brian Wylie (2002)
- Maps the triangles to GPU (Basis Graph)
- Better integration methods instead of average colors
- GPGPU (General Purpose Computation on GPU) techniques

Two steps approach



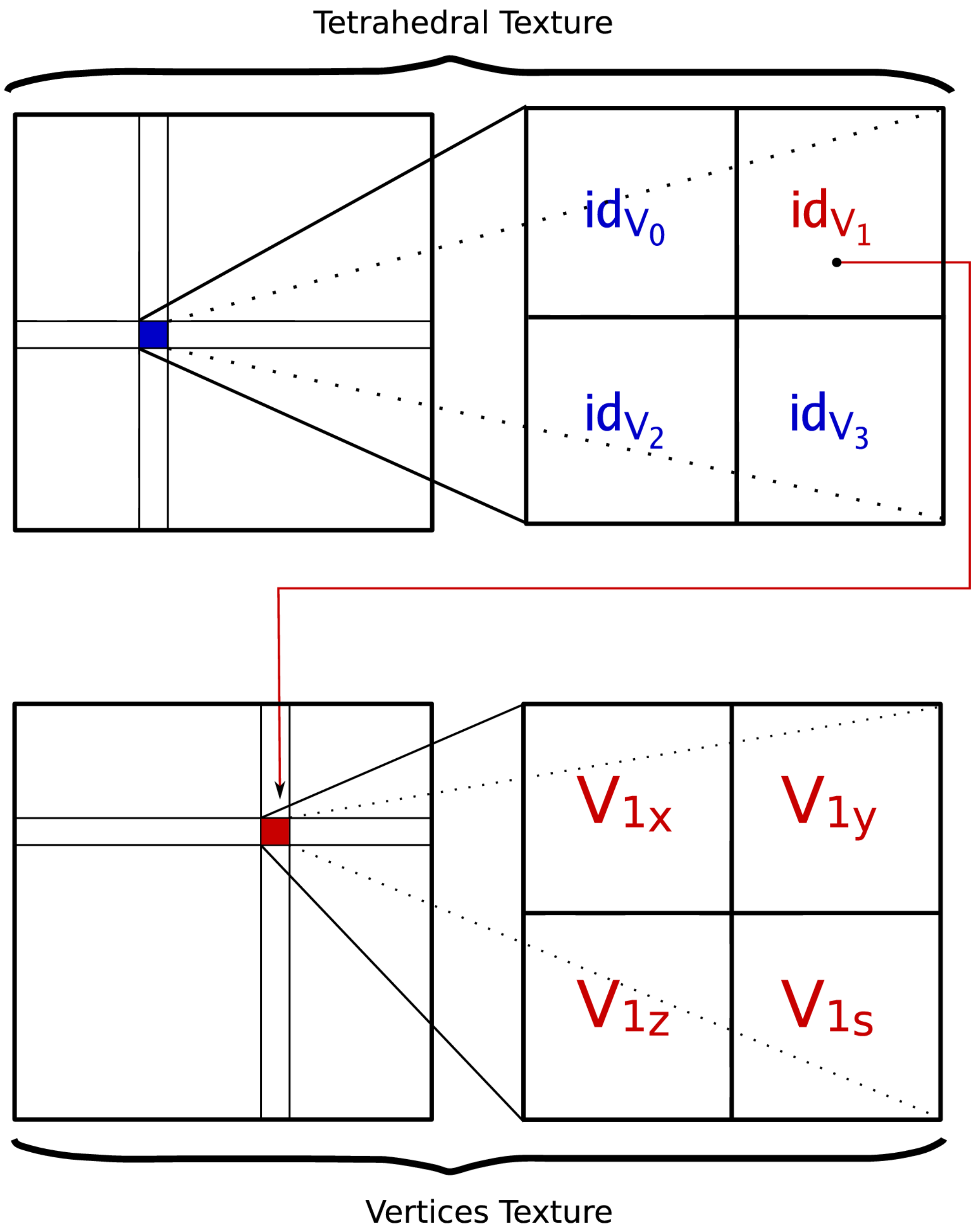
- Two GPU passes

First Shader Overview



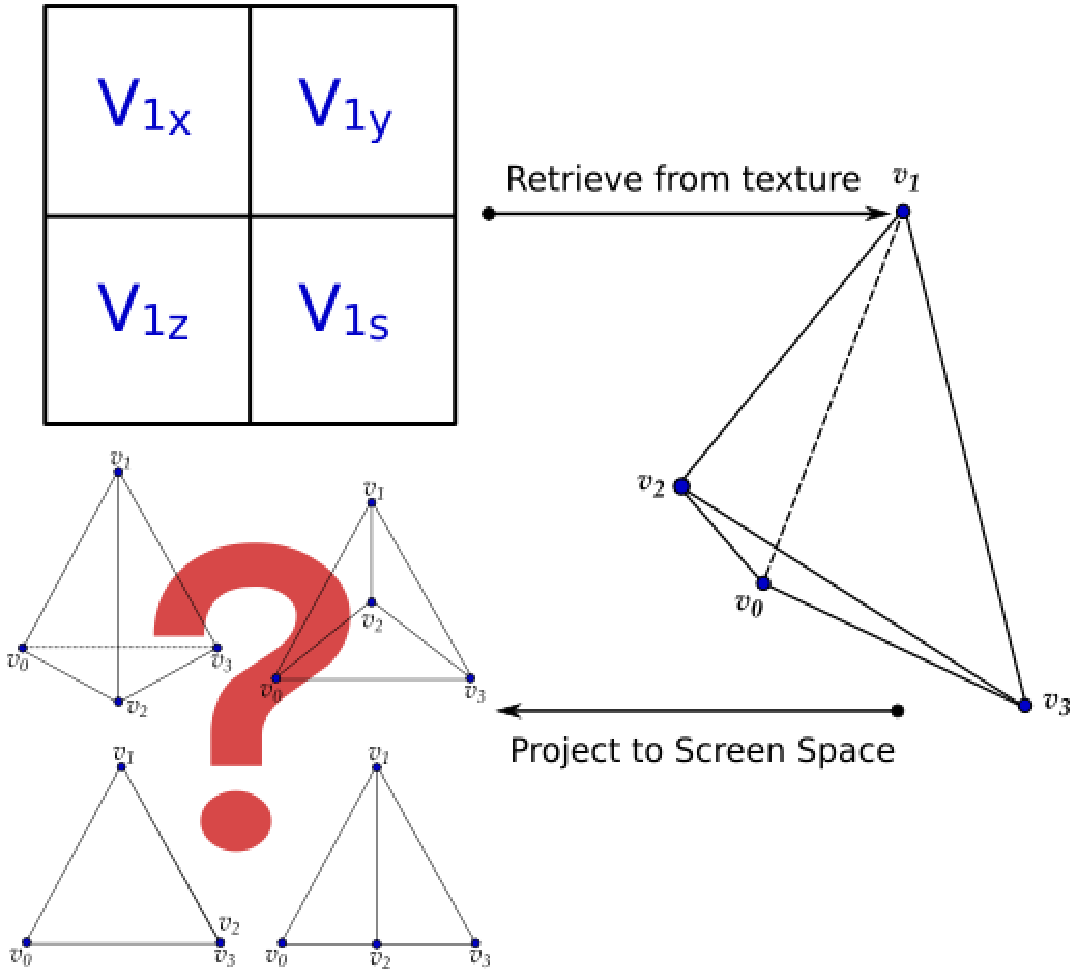
First Shader : Textures

- Render tetrahedral texture
  - Same size as viewport
  - Maps one texel to one fragment
  - One fragment shader pass for each tetrahedron
- 32 bits per component

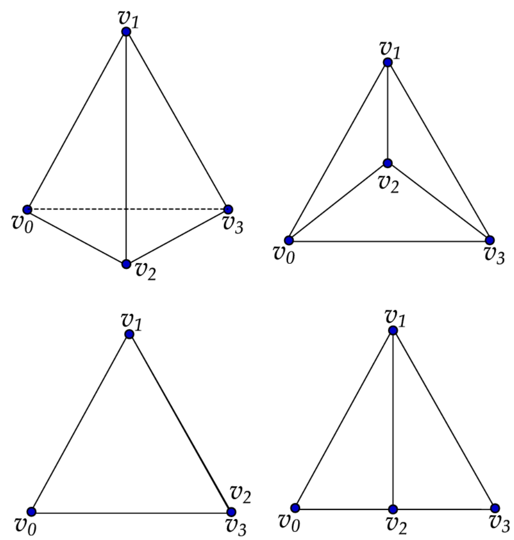


First Shader : Retrieve Vertices

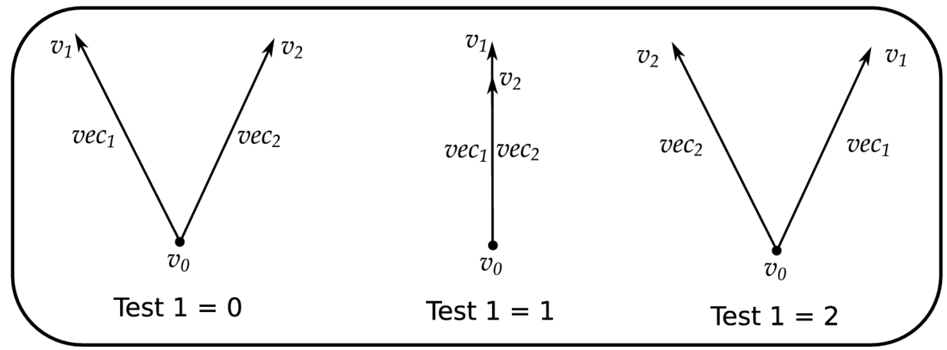
- Retrieve vertices from texture
- Project to screen space
  - Still no information about vertices arrangement



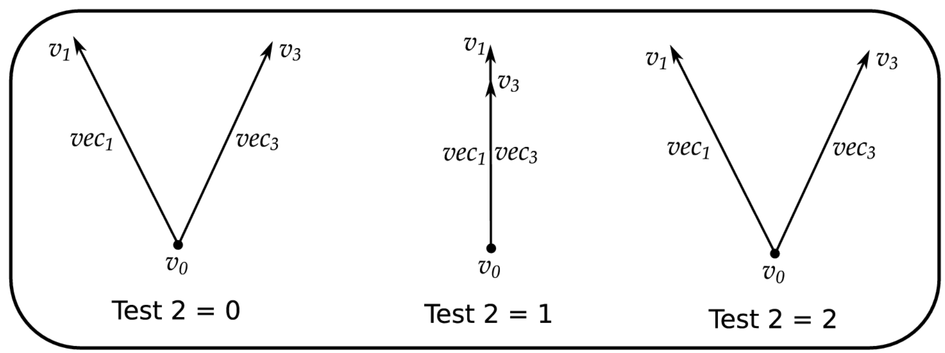
First Shader : Determine Projection Classes



Test 1 = sign((vec1 x vec2).z)+ 1

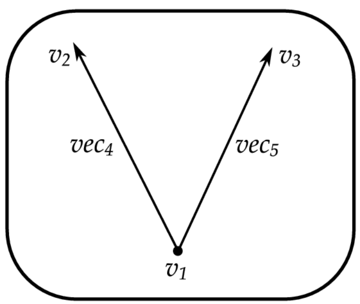


Test 2 = sign((vec1 x vec3).z)+ 1



Test 3 = sign((vec2 x vec3).z)+ 1

Test 4 = sign((vec4 x vec5).z)+ 1



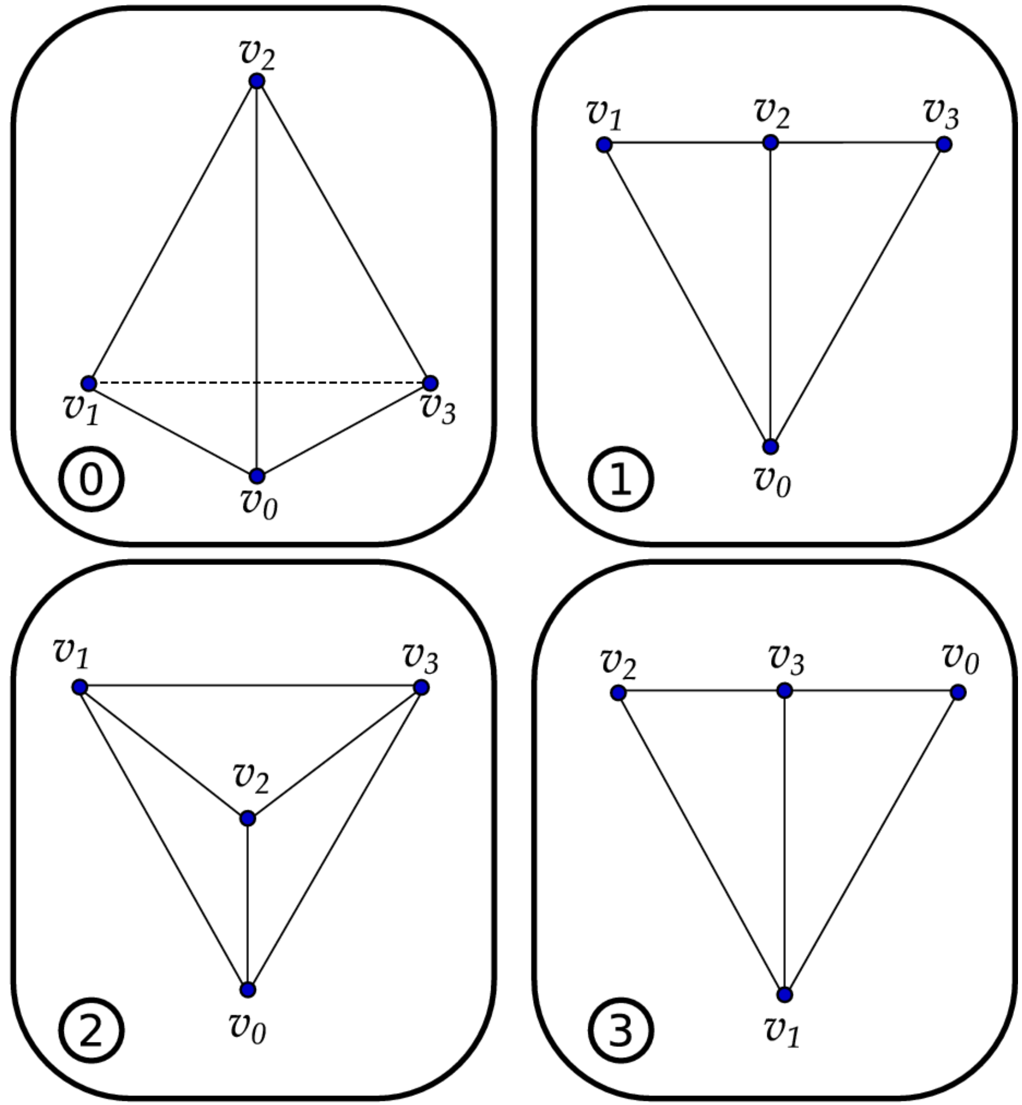
- 4 Cross Product Tests
  - Covers all possible projections

First Shader : Classification Table

- Ternary Truth Table
  - Three test results : 0, 1, 2
  - 4 Vertices in correct order (rows)
  - $3^4 = 81$ rows
  - Row id = (test1* 27) + (tests2 * 9) + (tests3 * 3) + (tests4* 1)
  - Actually, there are only 50 cases
    - Maximum two tests = 1 per tetrahedron
    - Else degenerated tet -> discard

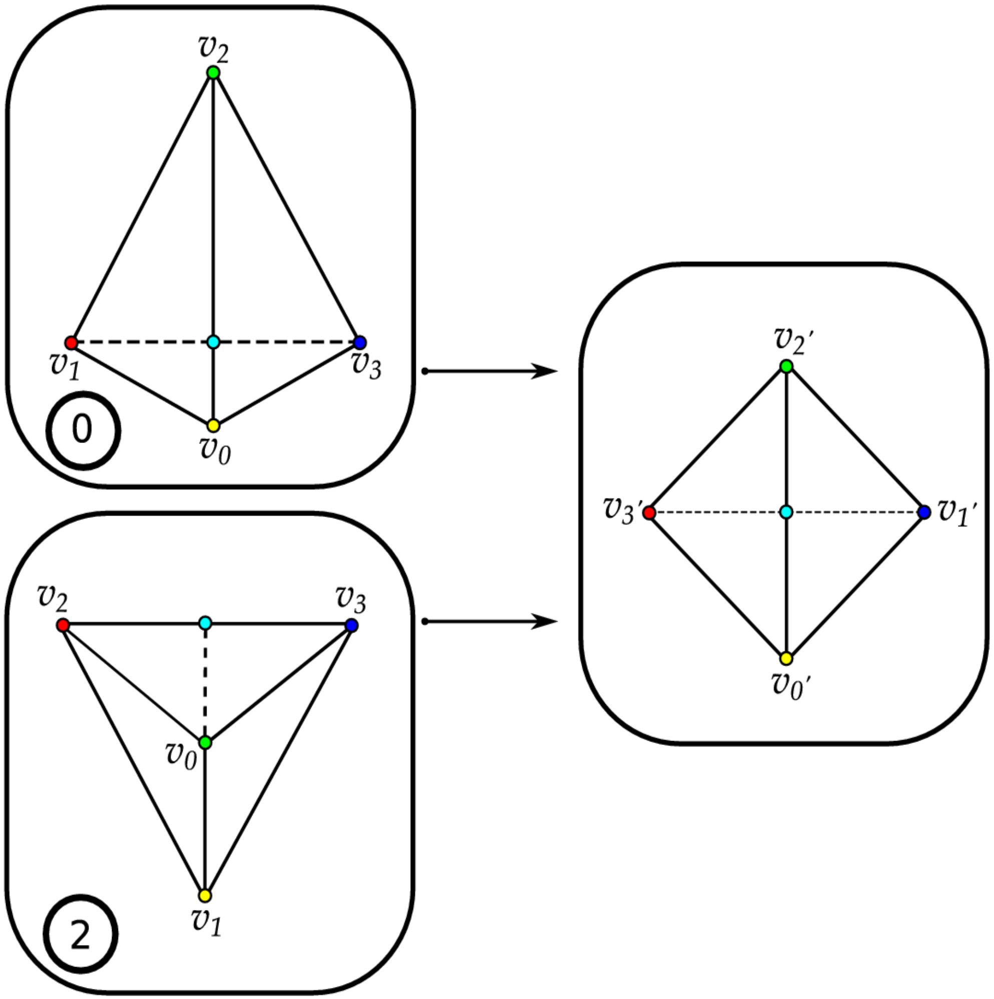
First Shader : Classification Table

Id	Test 1	Test 2	Test 3	Test 4	V_0	V_1	V_2	V_3
0	0	0	0	0	0	3	2	1
1	0	0	0	1	0	3	2	1
2	0	0	0	2	0	3	2	1
3	0	0	1	0	1	0	3	2



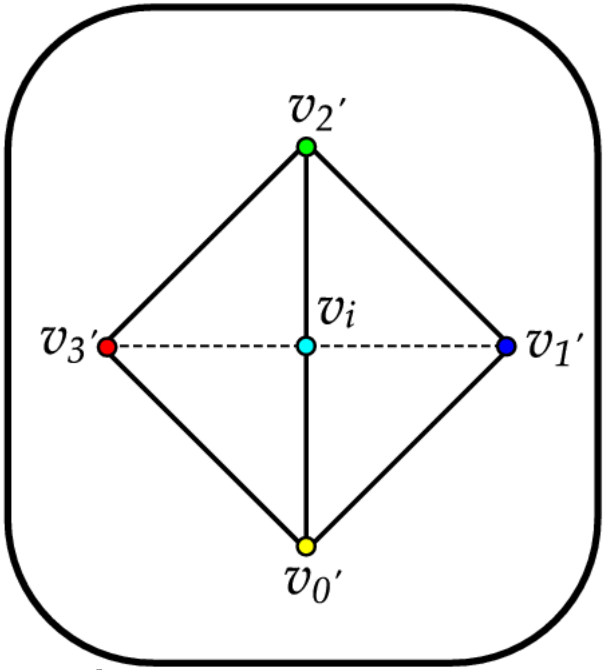
First Shader : Map to Basis Graph

- Map projected vertices to basis graph
  - Compute intersection vertex with same vectors for every class



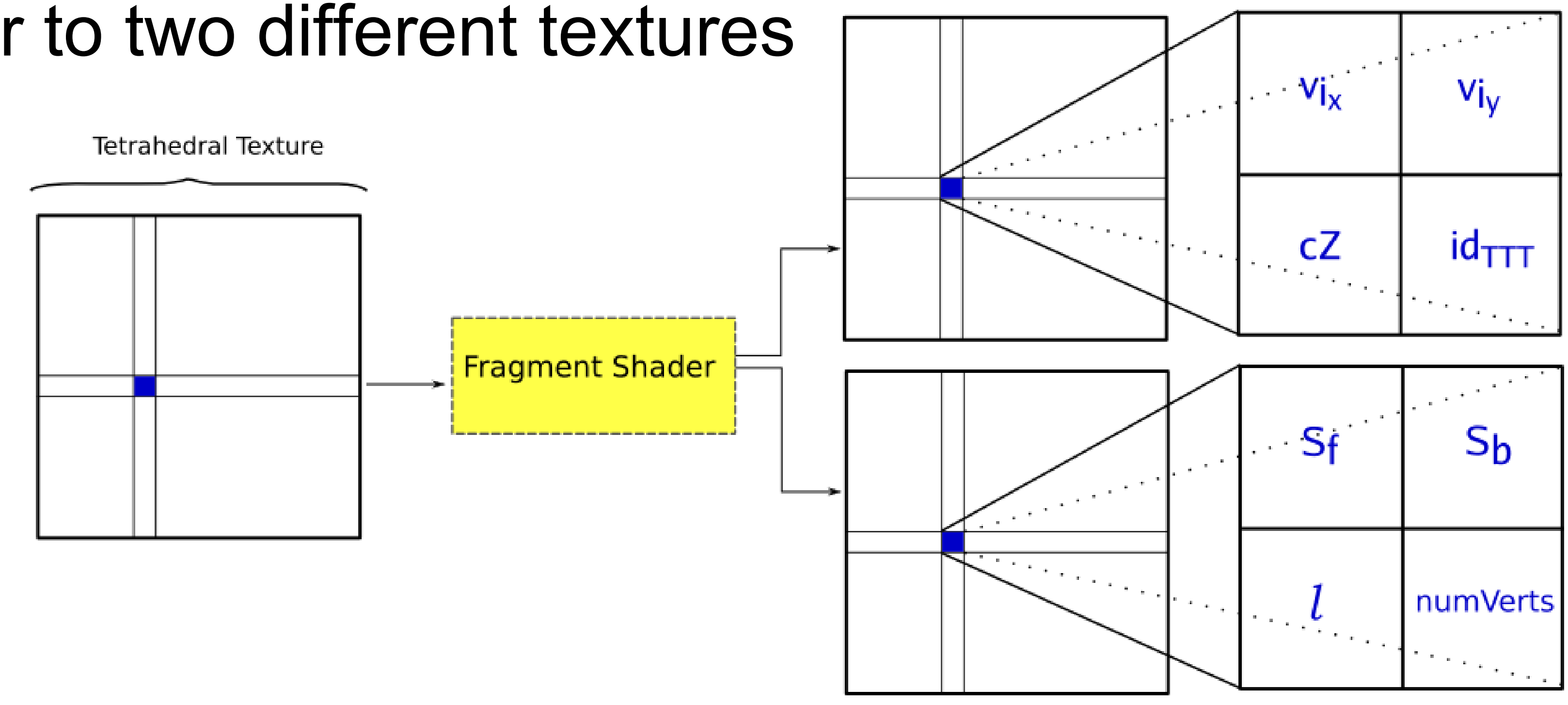
First Shader : Compute Intersection Vertex

- Compute thick vertex in basis graph
  - Intersection between segments $v_0'v_2'$ and $v_1'v_3'$
  - Line coefficients:
    - (front) $v_i = v_0' + \boldsymbol{u} * (v_2' - v_0')$
    - (back) $v_i = v_1' + \boldsymbol{t} * (v_3' - v_1')$
  - Compute v_i, Sf, Sb
  - Thickness l
    - Compute difference in z between front and back intersections vertices



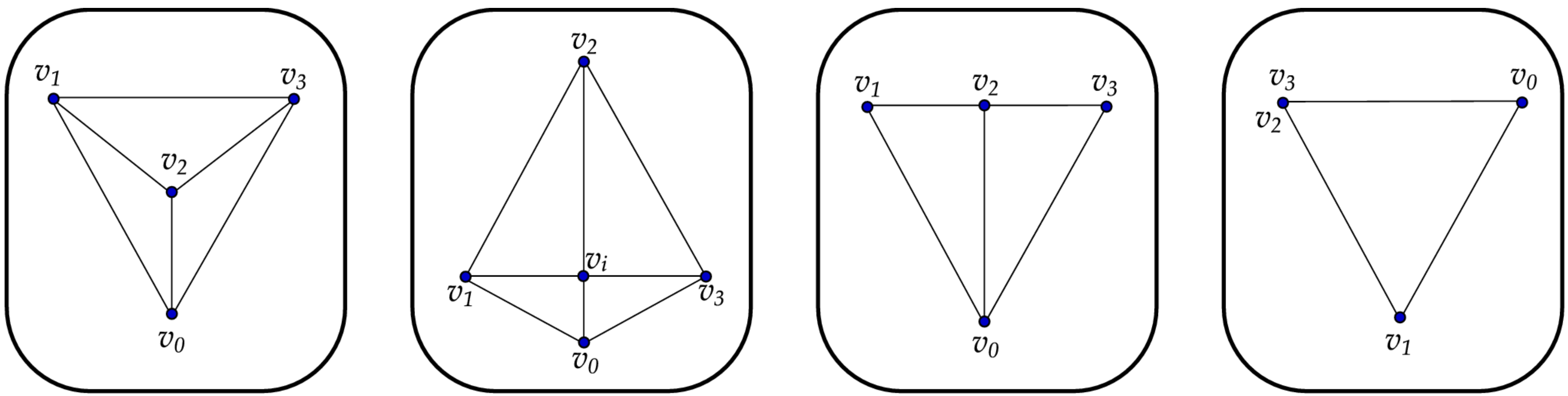
First Shader : Render to Texture

- Capture fragment shader using FBO (Frame Buffer Object)
  - Instead of rendering to screen, render to a texture
- MRT (Multiple Rendering Targets)
  - Render to two different textures



Preparing for Second Shader

- Before executing second shader:
  - Sort tetrahedra (merge sort using centroids)
  - Setup Vertex and Color Arrays
    - Primitives are passed to second shader as triangle fans
    - For each class the fan has a specific number of triangles



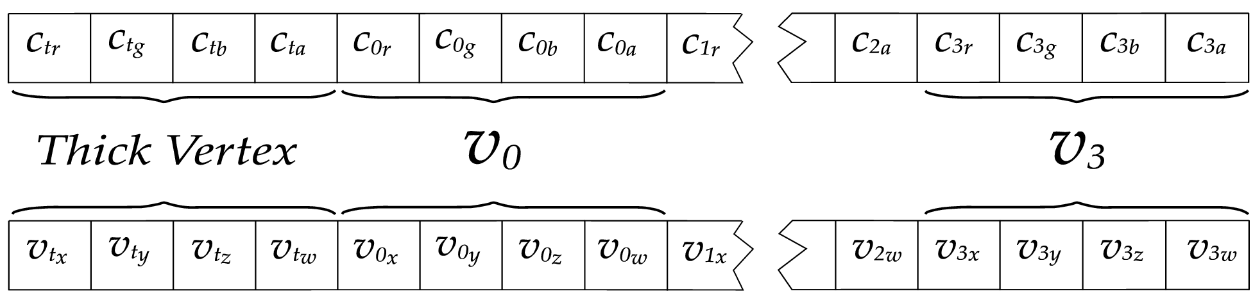
Class 1 3 Triangles | Class 2 4 Triangles | Class 3 2 Triangles | Class 4 1 Triangle

Preparing for Second Shader : Sorting

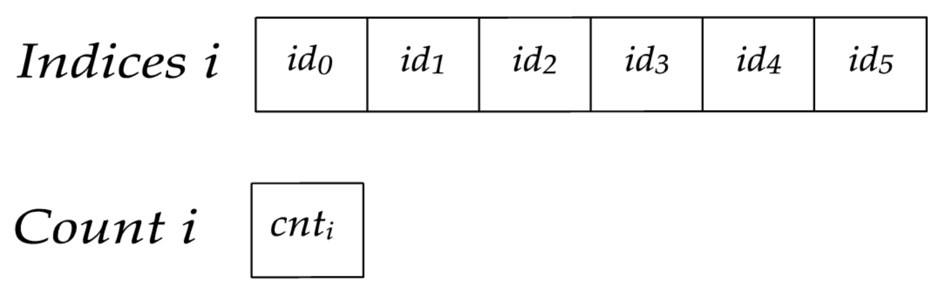
- Centroids are computed on first fragment shader
- Merge sort O(*n* log *n*) when model is still
- Simple layer sorting during rotations O (*n*)
  - Distribute tetrahedra in slabs perpendicular to the viewing vector
  - Render slabs back to front
  - Tetrahedra inside slabs remain unsorted

Preparing for Second Shader : Arrays Structure

- Vertex and Color Arrays
  - 5 vertices per Tet (four vertices + thick vertex)

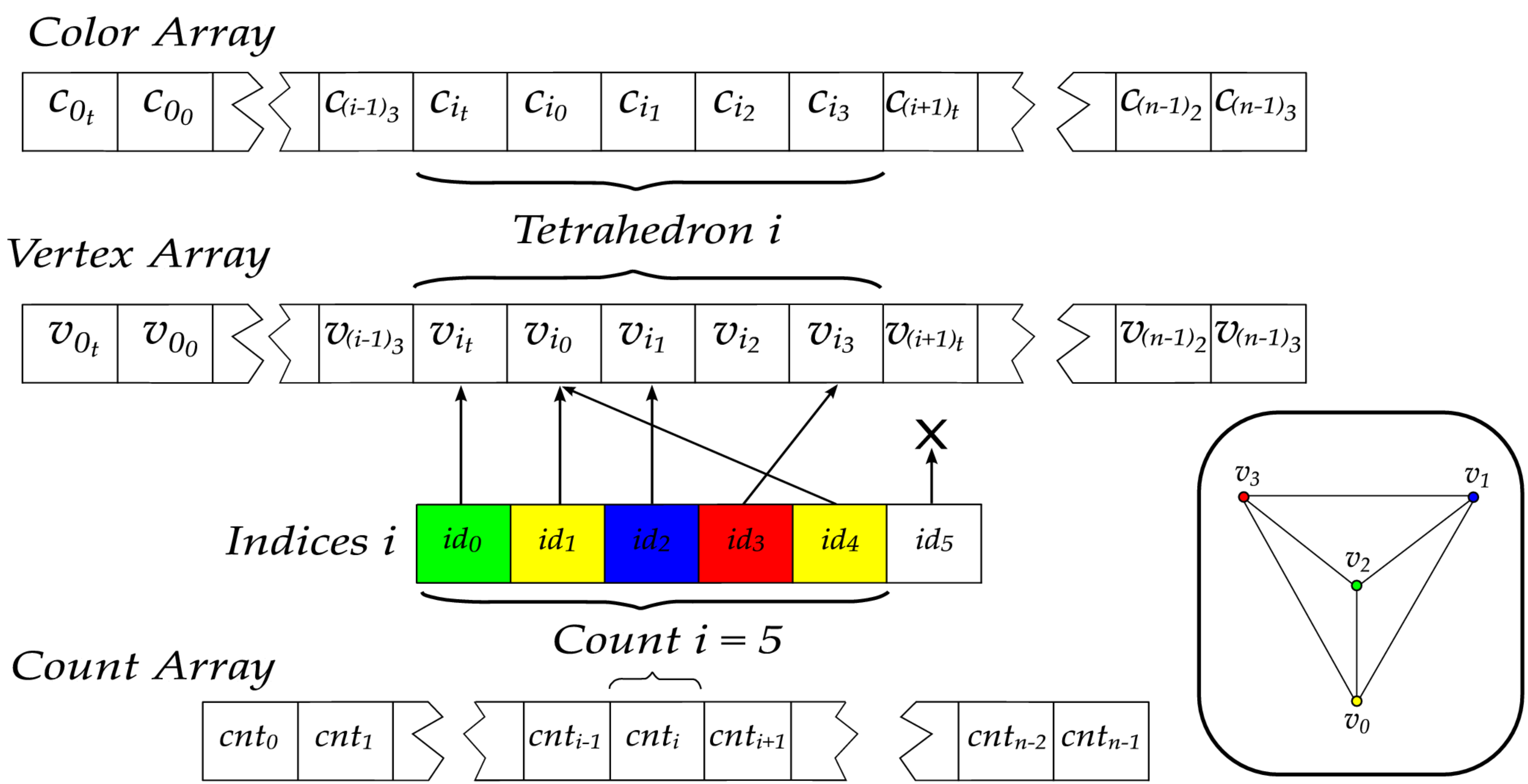


- Indices and Count Arrays
  - Order and number of vertices in triangle fan



Preparing for Second Shader : Arrays Structure

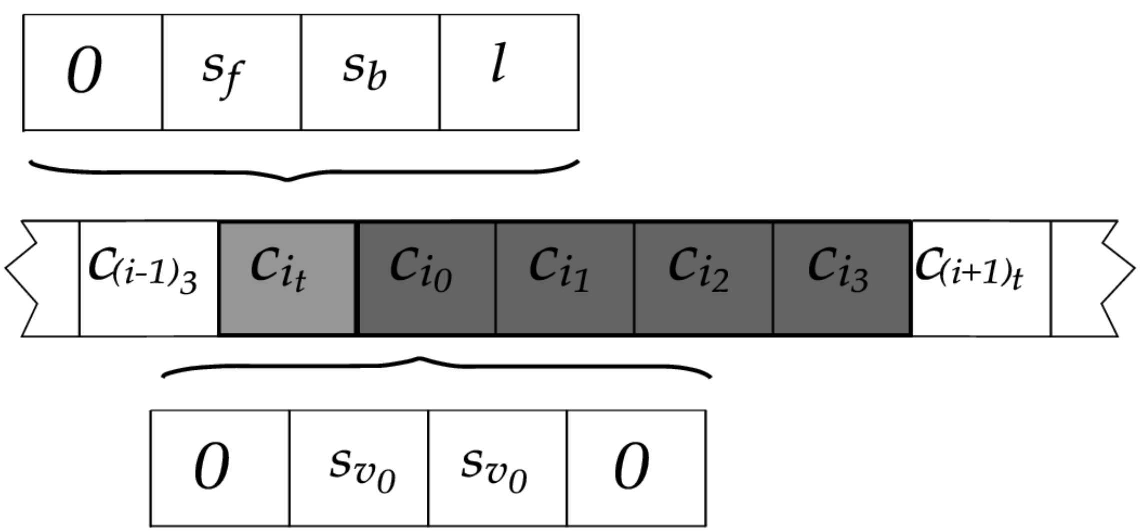
- Array structures for *glMultiDrawElements*:

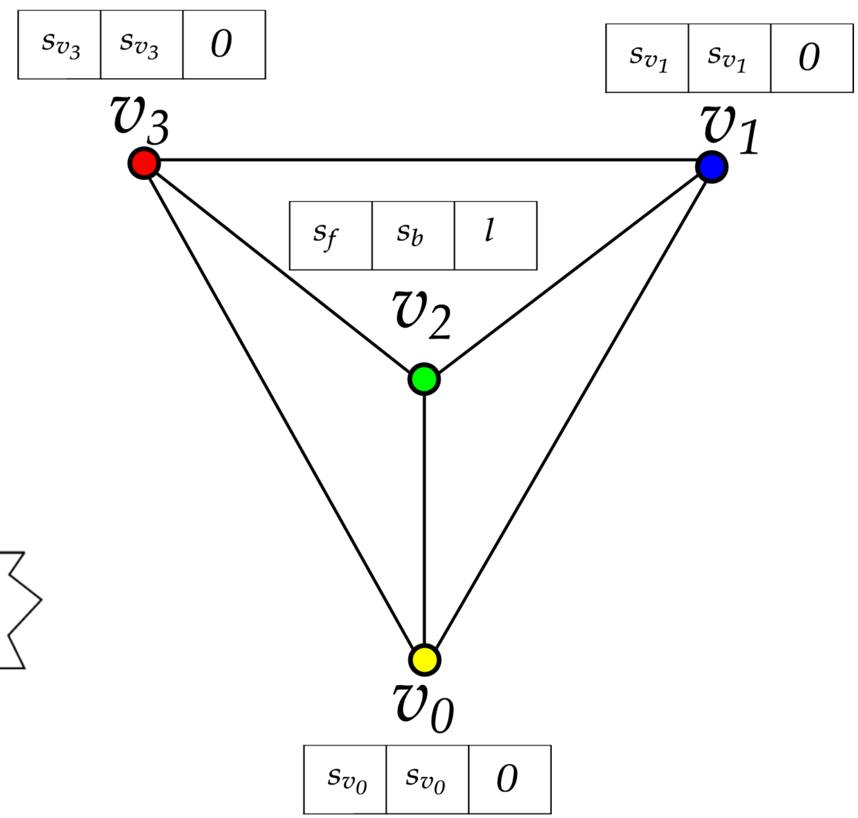


Second Shader : Vertices Interpolation

- Except for thick vertex, all others are rendered with original values of the color array

Color Array for Tetrahedron i :



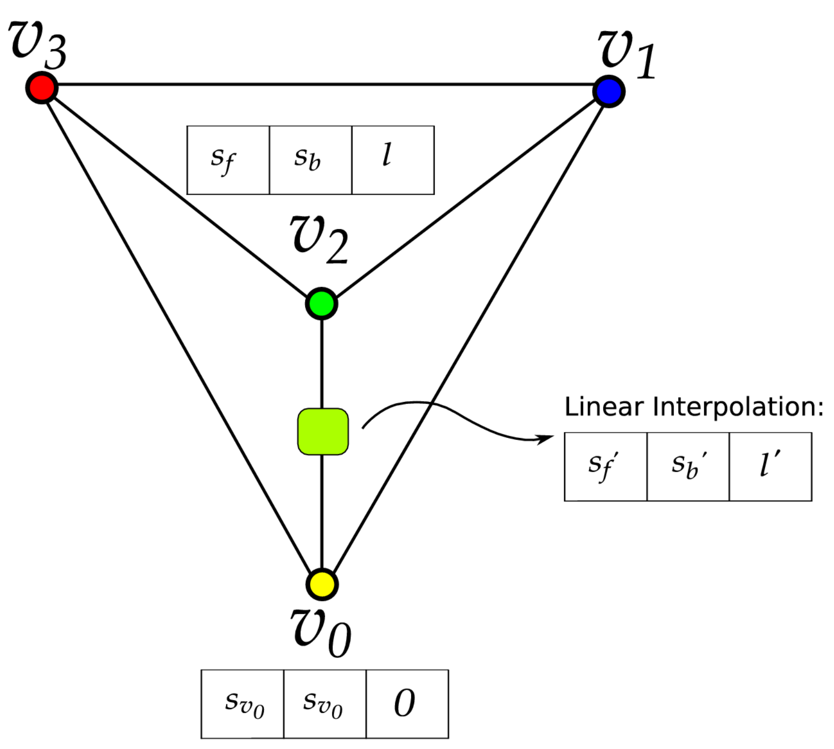


Second Shader : Fragment Color

- Linear interpolation of vertices scalar and thickness values

- Average Scalar:

$$S_{avg} = \frac{(S_f' + S_b')}{2}$$



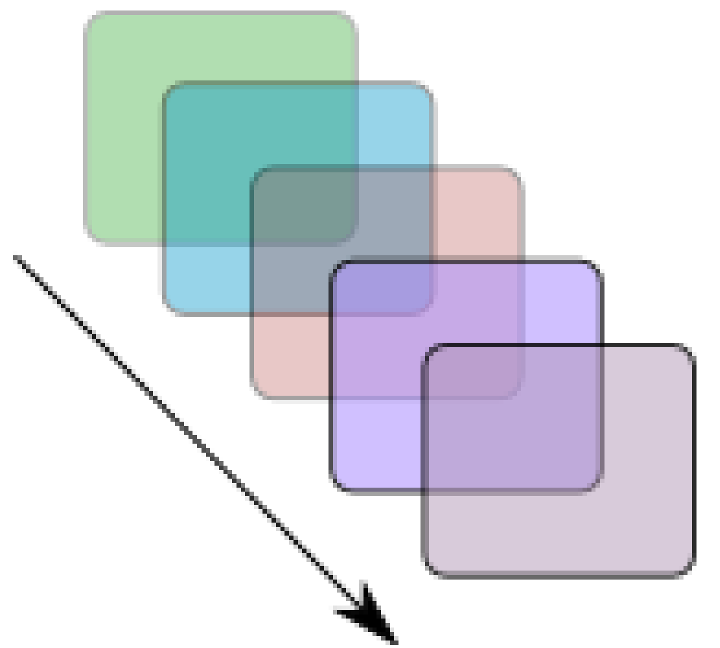
Second Shader : Fragment Color and Composition

- Transfer Function Texture Look up
  - 1D Texture (255 values)
  - Each texel contains RGBA values

$$RGB\tau_{Savg} = tf\,(S_{avg})$$

$$\alpha_{frag} = 1 - e^{-\tau\, l}$$

$$RGB_{frag} = RGB_{Savg} * \alpha_{frag}$$



$$I_{new} = I_{old}\,(1 - \alpha_{frag}) + RGB_{frag}$$

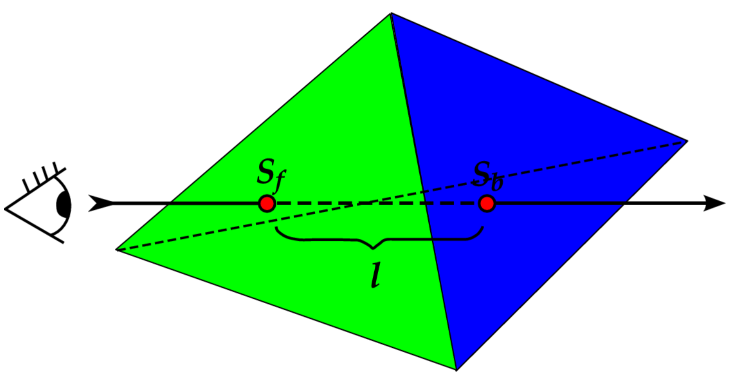
Ray Integration

- Volume Rendering Integration:

$$I_D = I_0 \underbrace{e^{-\int_0^l \tau(t)\,dt}}_{1-\alpha} + \underbrace{\int_0^l k(s)\tau(s) e^{-\int_s^l \tau(t)\,dt} ds}_{\alpha * C_{rgb}}$$

- Approximate using average scalar:

$$I_D = I_0 \underbrace{e^{-l\frac{1}{2}(\tau_b + \tau f)}}_{1-\alpha} + \underbrace{\frac{1}{2}(k_f + k b)(1 - e^{-l\frac{1}{2}(\tau_b + \tau f)})}_{\alpha * C_{rgb}}$$



Pre-Integration

- Introduced by Engel et al. (2001)
  - More accurate : Less artifacts
  - Compute integration for different Sf, Sb and thickness values
  - Use a 3D texture to store the values : slow access!
  - Access the texture during second fragment shader computation to obtain final color and opacity value
  - Problem : creation of the 3D texture is slow!

$$I_D = I_0 \underbrace{e^{-\int_0^l \tau(t)\,dt}}_{1-\alpha_{t3D}} + \underbrace{\int_0^l k(s)\,\tau(s)\,e^{-\int_s^l \tau(t)\,dt}\,ds}_{RGB_{t3D}}$$

Partial Pre-Integration

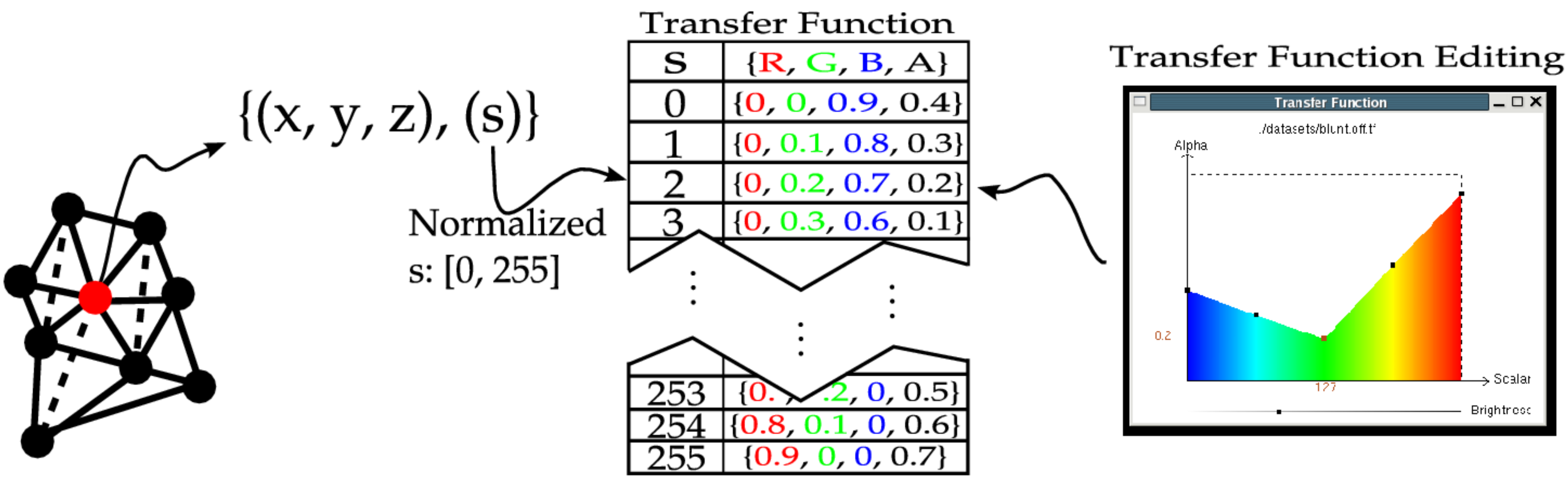
- Introduced by Moreland e Angel (2004):
  - Compute integration for different Sf, Sb and thickness values without transfer function dependency
  - Use a 2D texture to store the values (different resolutions : 512x512 , 1024x1024)
  - Access the texture during second fragment shader to obtain weight of color
  - Slower than pre-integration (some computation is left to the shader)
  - Not a problem : creation of the 2D texture is slow, but is computed once (not pre-computation)

Transfer Function

- Interactive editing:
  - Update the transfer function texture each time it changes
  - Only possible using partial pre-integration or average scalar method
  - Pre-integration : time to recompute too high
  - Colors use logarithmic scale : smooth transition, attenuates artifacts

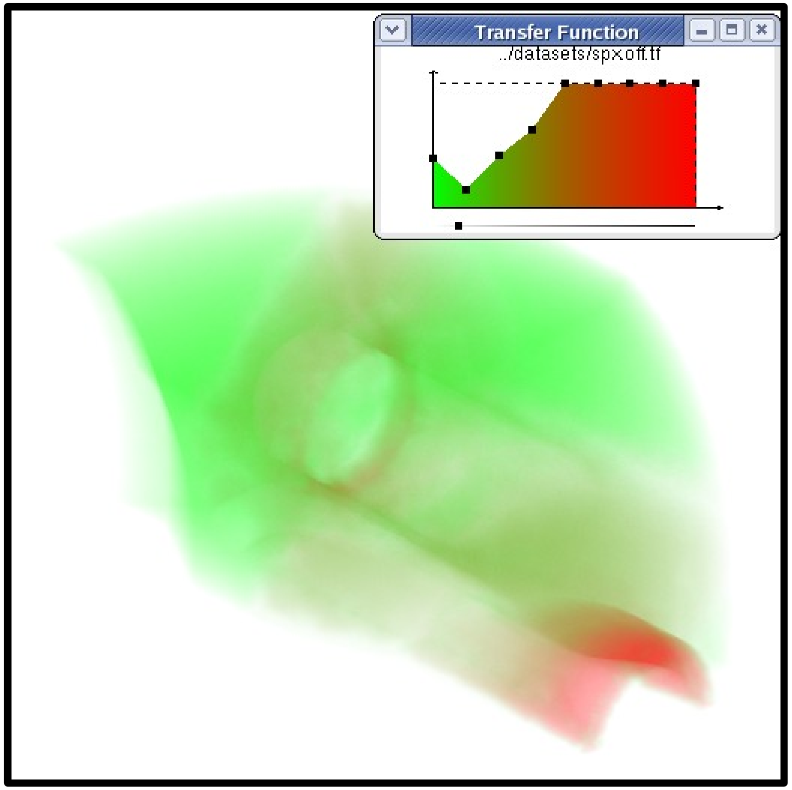
Transfer Function

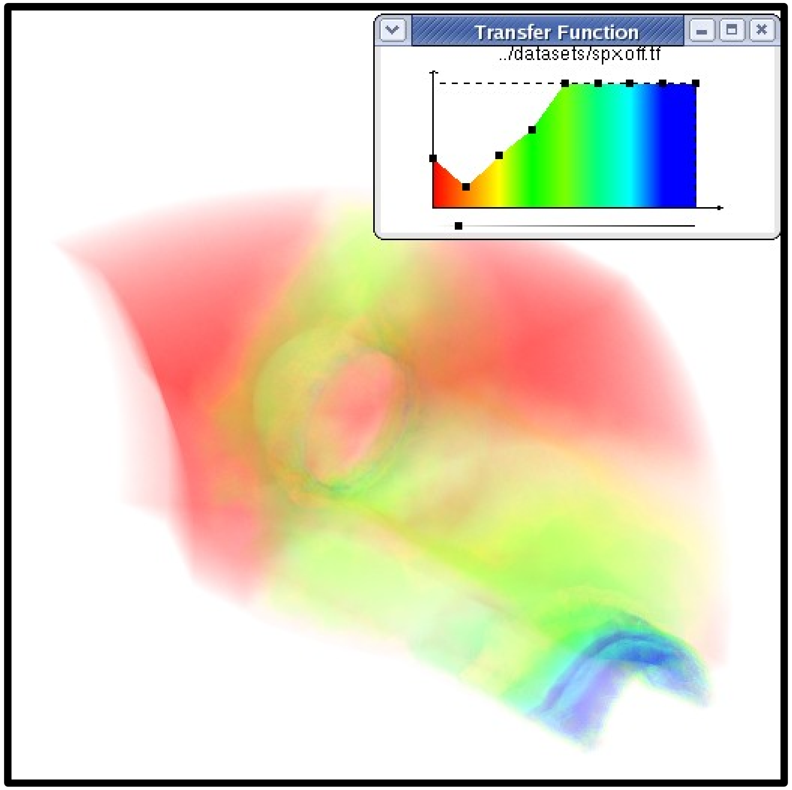
- Interactive editing:



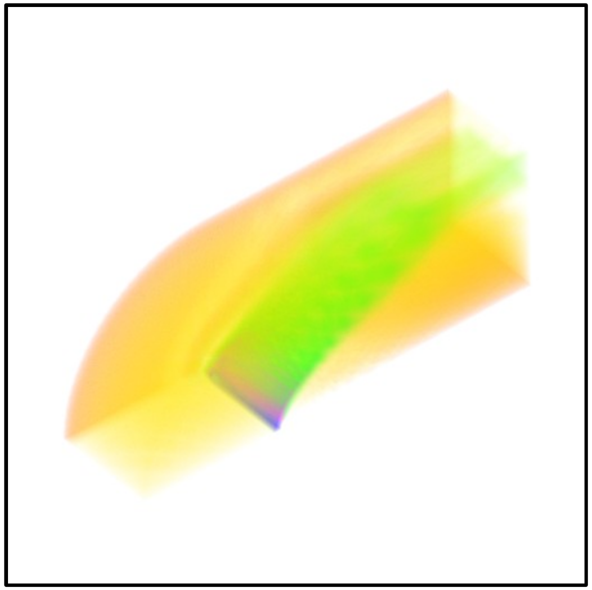
Transfer Function

- Interactive editing and different color maps:

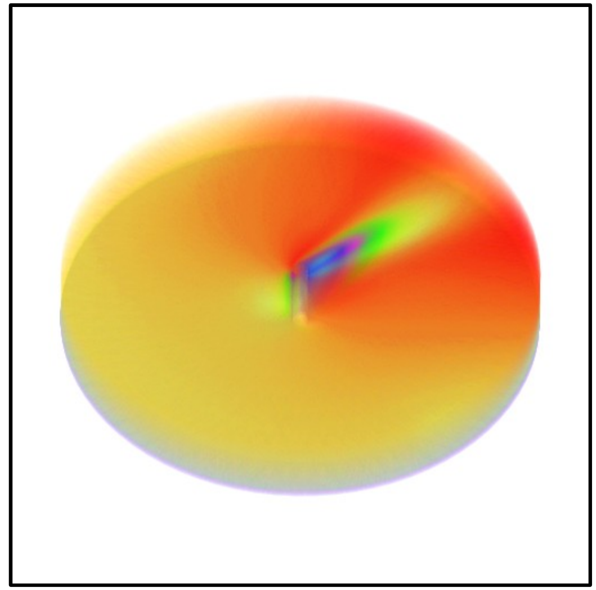




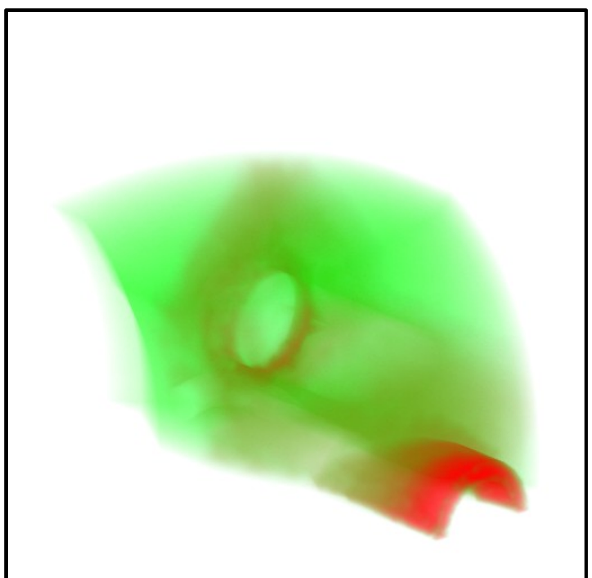
Results



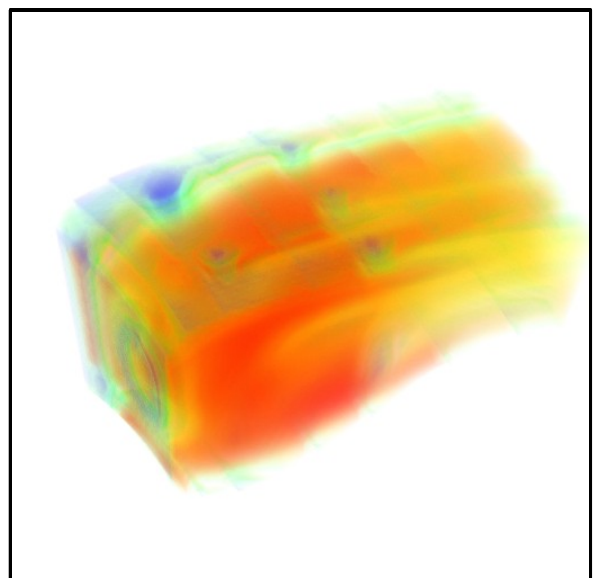
Blunt Fin



Oxygen Post



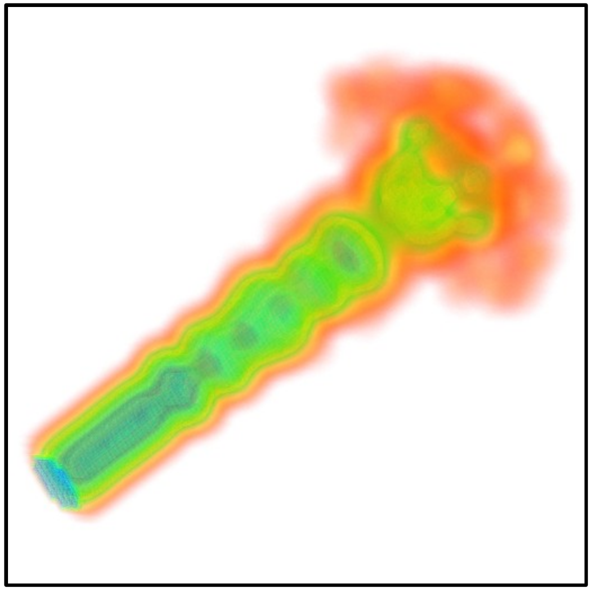
Spx

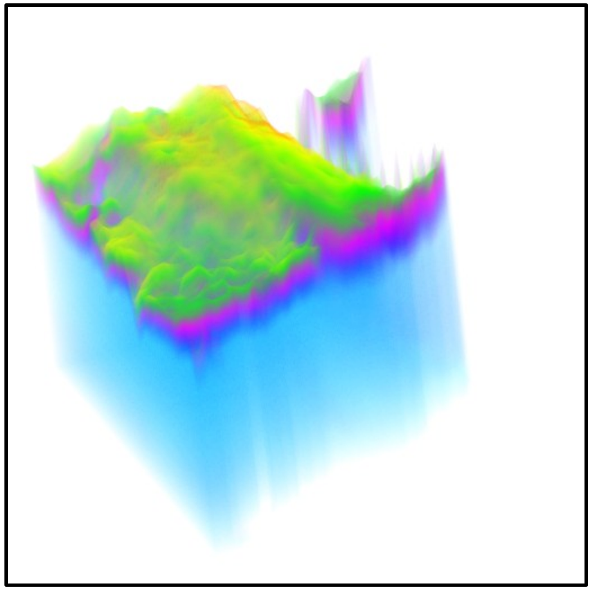


Combustion Chamber

Results

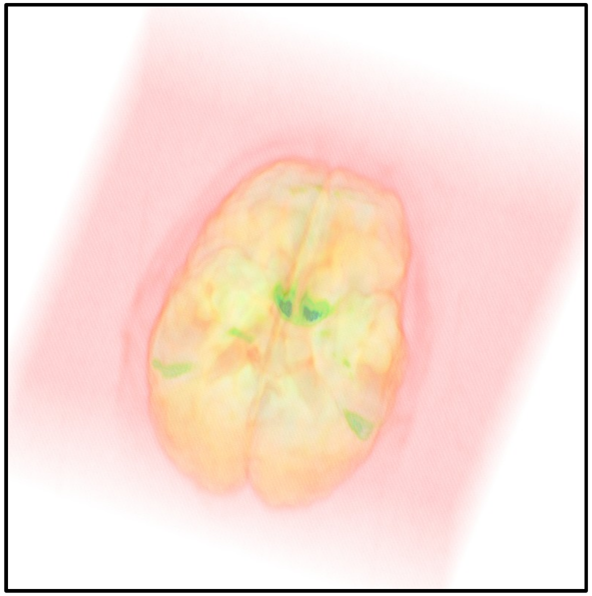
Fuel Injection





Ocean

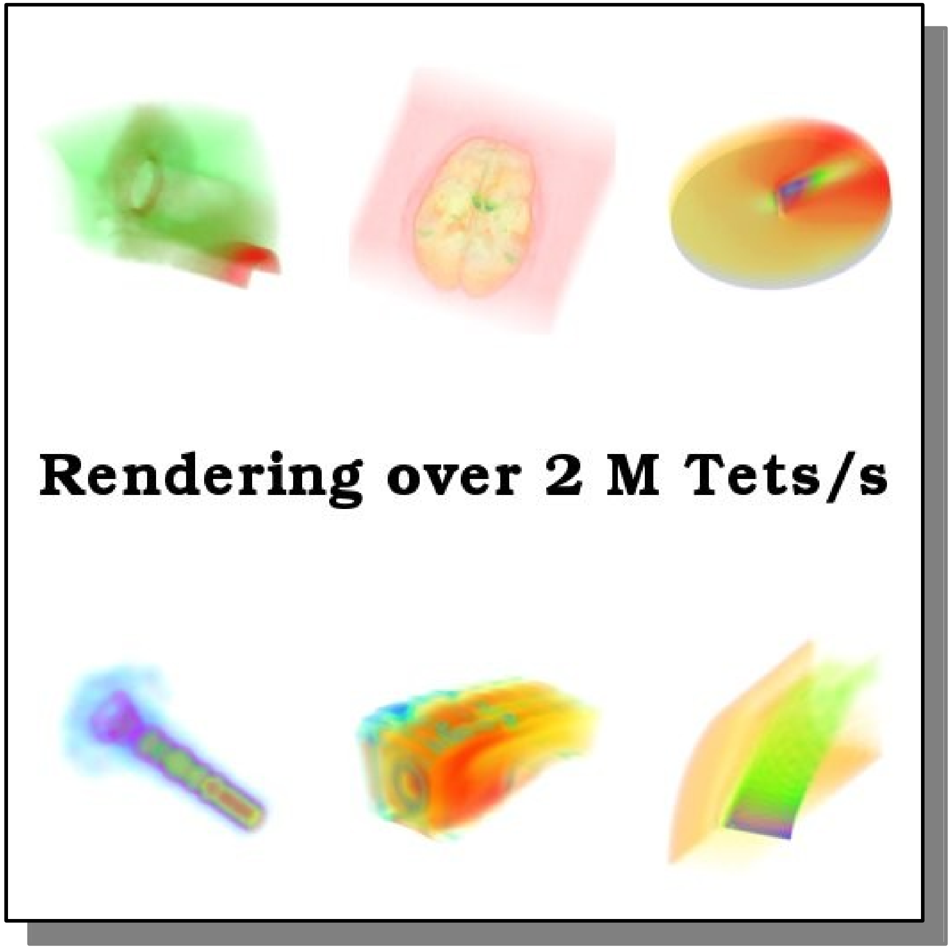
Brain DTI



Data Set	# Vertices	# Tets	FPS	Tets / sec
Spx 1	2.9 K	13 K	95.32	1233.2 K
Blunt	40 K	187 K	11.3	2119.7 K
Comb	47 K	215 K	9.32	2005.4 K
Post	110 K	513 K	4.49	2384.4 K
Spx 2	150 K	828 K	3.04	2526.9 K
Fuel	262 K	1.5 M	1.49	2246.0 K
Brain	950 K	5.5 M	0.46	2560.8 K

Results

- Video



Performance Gain

- What makes the difference?
  - Use of Vertex Array (OpenGL optimization)
  - Use fragment shader for heavy computations (vertex shader is slower)
  - No vertex attributes (reduces CPU—BUS transfers)
  - Keep model in GPU texture memory
  - Choosing texture formats and types:
    - GL_TEXTURE_2D faster than GL_TEXTURE_RECTANGLE_2D
    - GL_TEXTURE_3D : slow access!
  - Eliminate heavy computations from shaders (look up classification table and exponential table for example)

Quality Gain

- Rendering better images:
  - Use 32 bits texture (reducing precision loss)
    - Tetrahedral texture
    - Vertices texture
  - For some textures 8 bits is appropriate and faster, classification table for example
  - Access textures using linear interpolation instead of nearest value
  - Use partial pre-integration instead of average colors

Conclusion

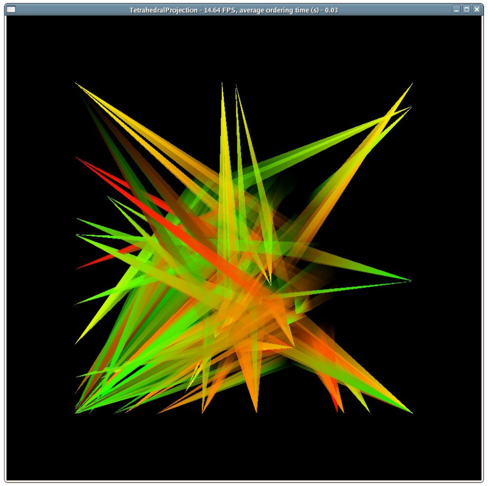
- Implementation of PT algorithm with GPU:
  - no major bus transfer overhead : whole model is stored in texture memory
  - low memory usage (no auxiliary data structures) : 20 bytes / tet
  - Up to 7 million tetrahedra
  - Over 2.0 M Tets/sec
  - Interactive transfer function

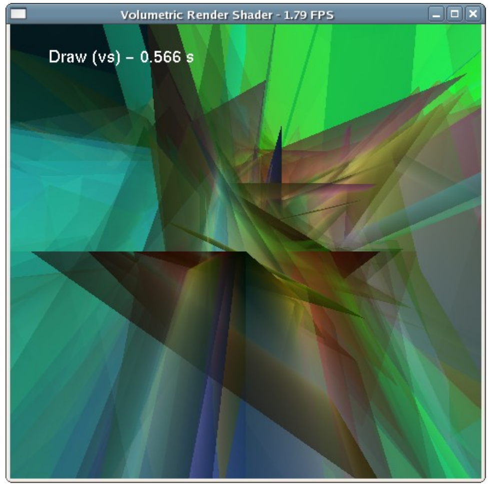
Future Works I

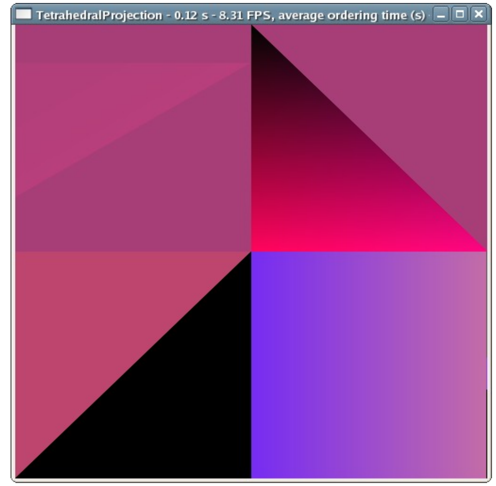
- Use Vertex Buffer Objects (performace)
  - render directly to vertex array, try to eliminate CPU passage (thick vertex update)
- Illumination model (quality)
  - gradients and isocontours (boundary estimation)
- Implement better sorting algorithm (quality & performance)

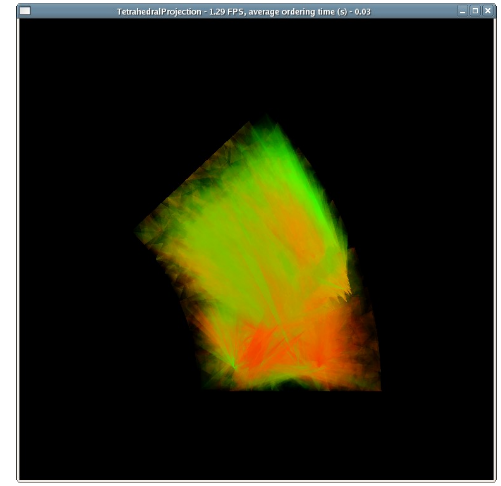
Future Works II

- Treat large meshes:
  - Remove tetrahedra without information (areas outside model)
  - Merge tetrahedra with same scalar
- Implement ray-casting for comparison:
  - No visibility sorting
  - More precise computation of thickness and scalar values
  - Auxiliary adjacency data structures (more bytes/tet)









"Making of" Questions? Thank you!