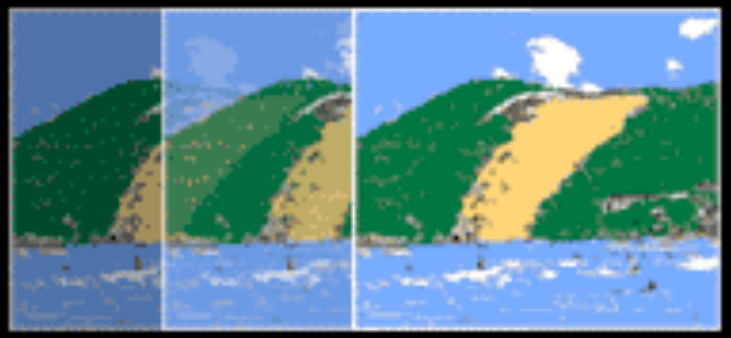
GPU-Based Volume Rendering of Unstructured Grids

Module 2:
Projected Tetrahedra + Polyhedral Cell Sorting

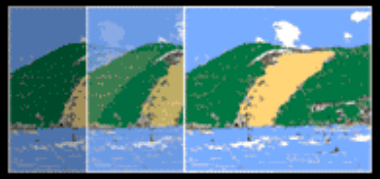
Cláudio T. Silva University of Utah



SIBGRAPI 2005

Natal - RN - Brazil

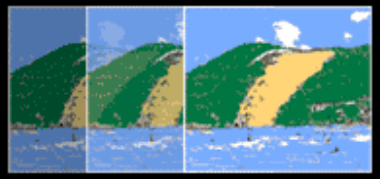
XVIII Brazilian Symposium on Computer Graphics and Image Processing



SIBGRAPI 2005

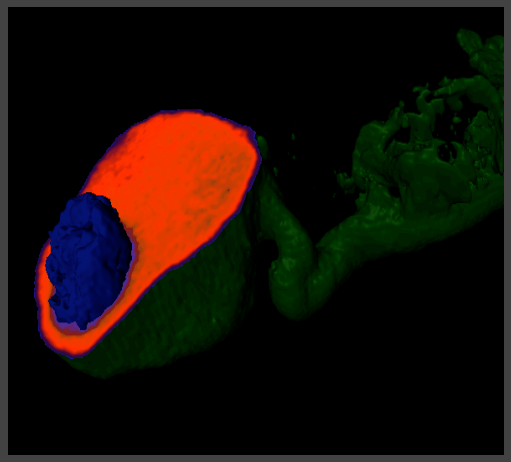
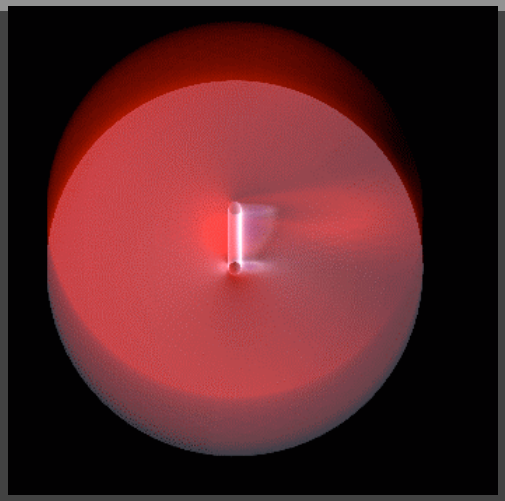
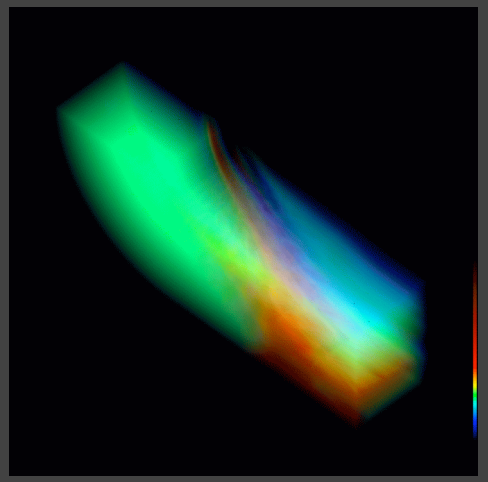
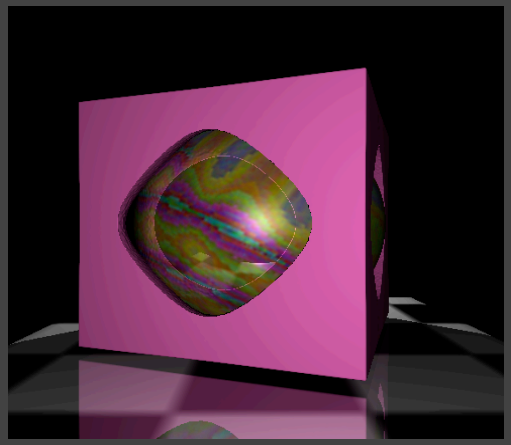
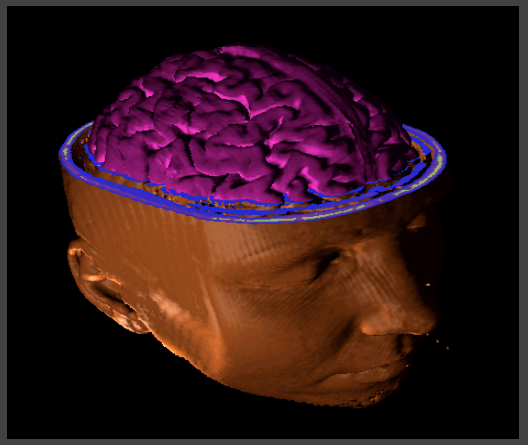
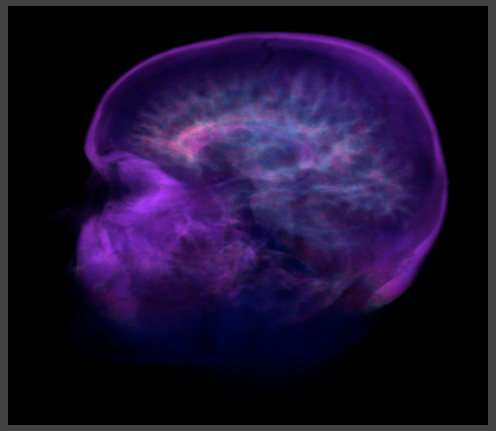
Outline

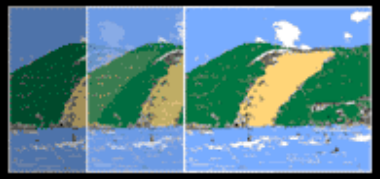
- Introduction to Volume Rendering
- Polyhedral Cell Sorting
- Hardware-Assisted Techniques



SIBGRAPI 2005

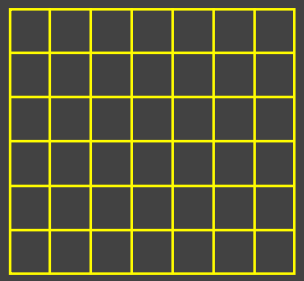
Introduction to Volume Rendering



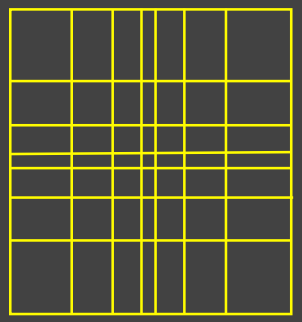
SIBGRAPI 2005

Grid Types

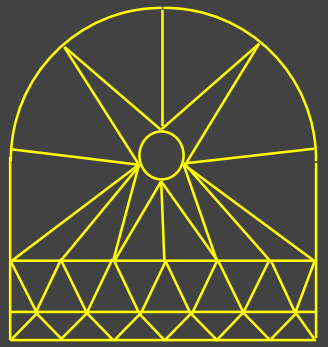


Regular

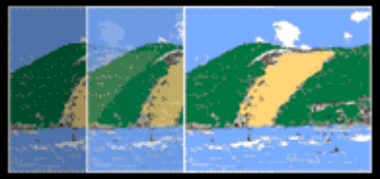
Curvilinear



Rectilinear



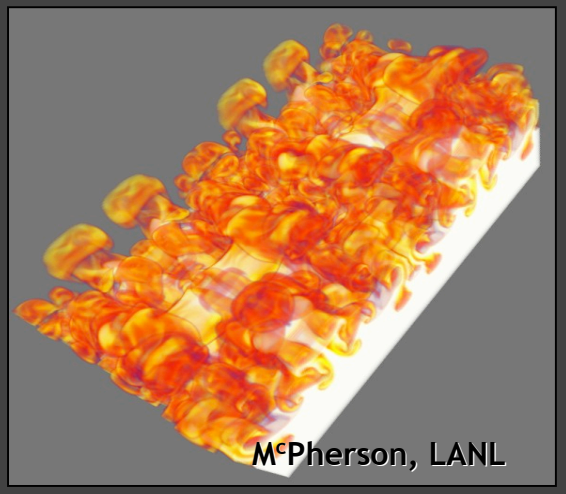
Irregular

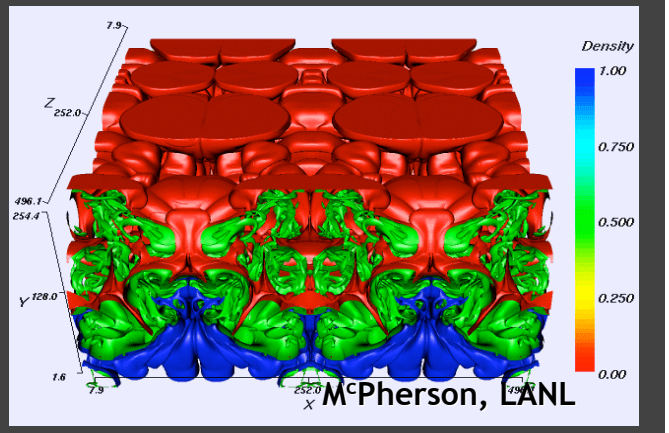


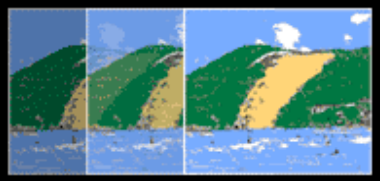
SIBGRAPI 2005

Volume Rendering vs Isosurfaces

- Direct Volume Rendering
  - Volume data → Image
  - Looks “inside” the data
- Isosurfaces
  - Volume data → Polygon model
  - Slice of the data



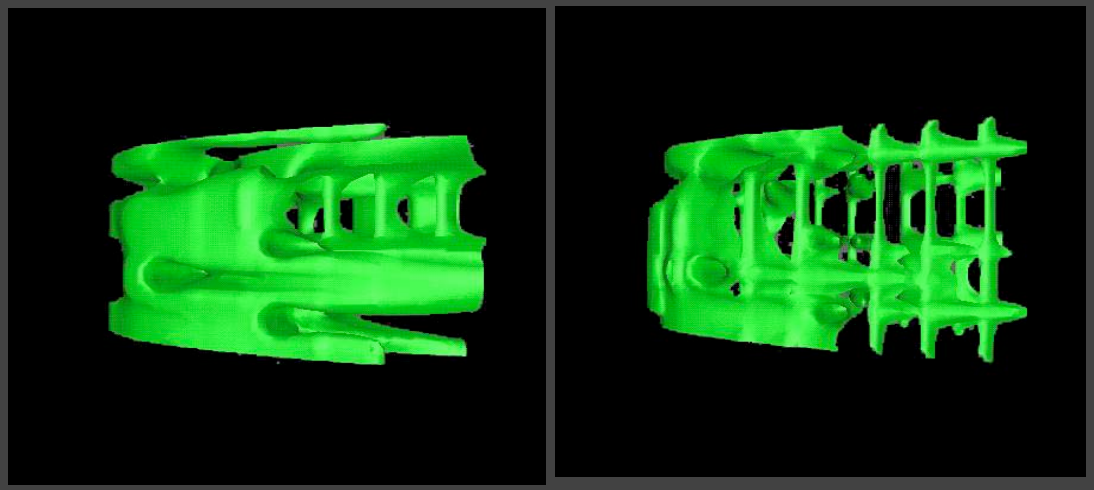


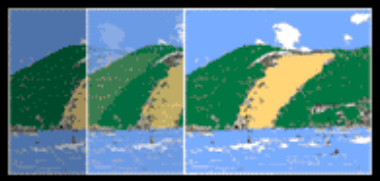


SIBGRAPI 2005

Isosurfaces

- For a query value q, find and display the isosurface of q: C(q) = {p | F(p) = q}



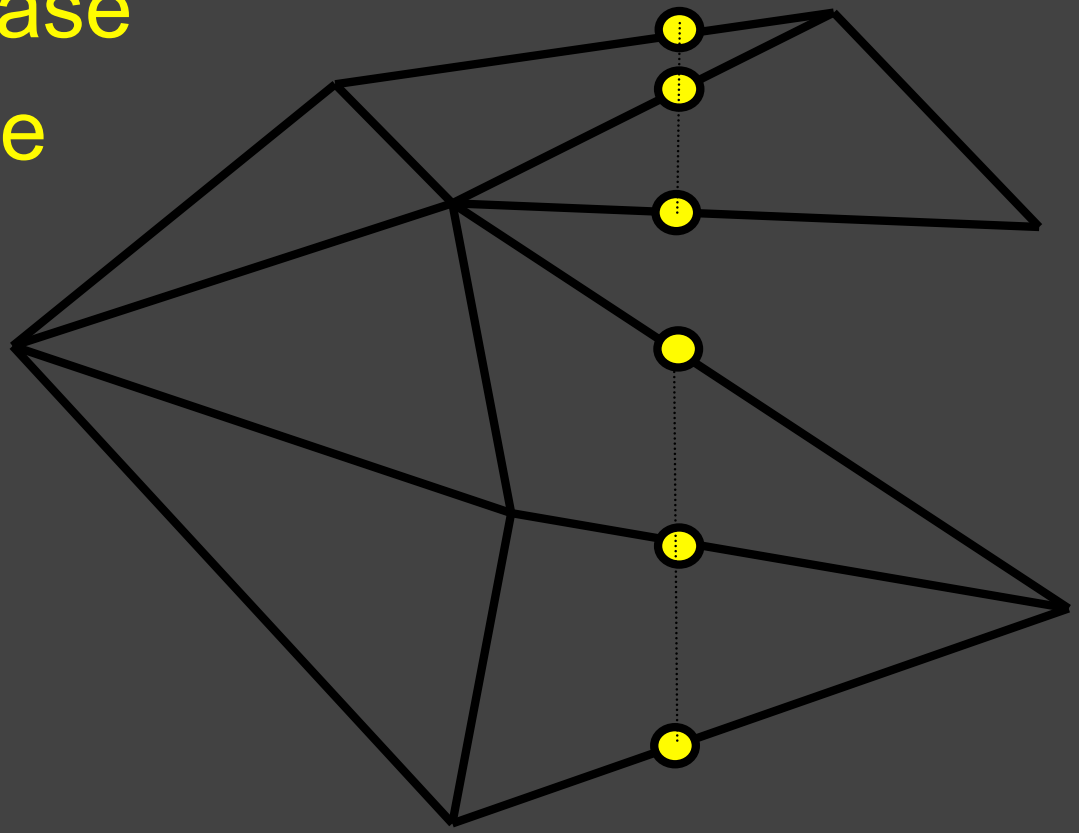


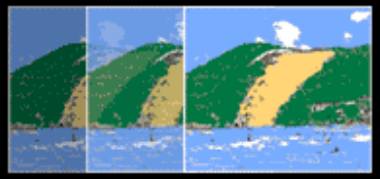
SIBGRAPI 2005

Volume Rendering at a High Level

(a) Sampling Phase

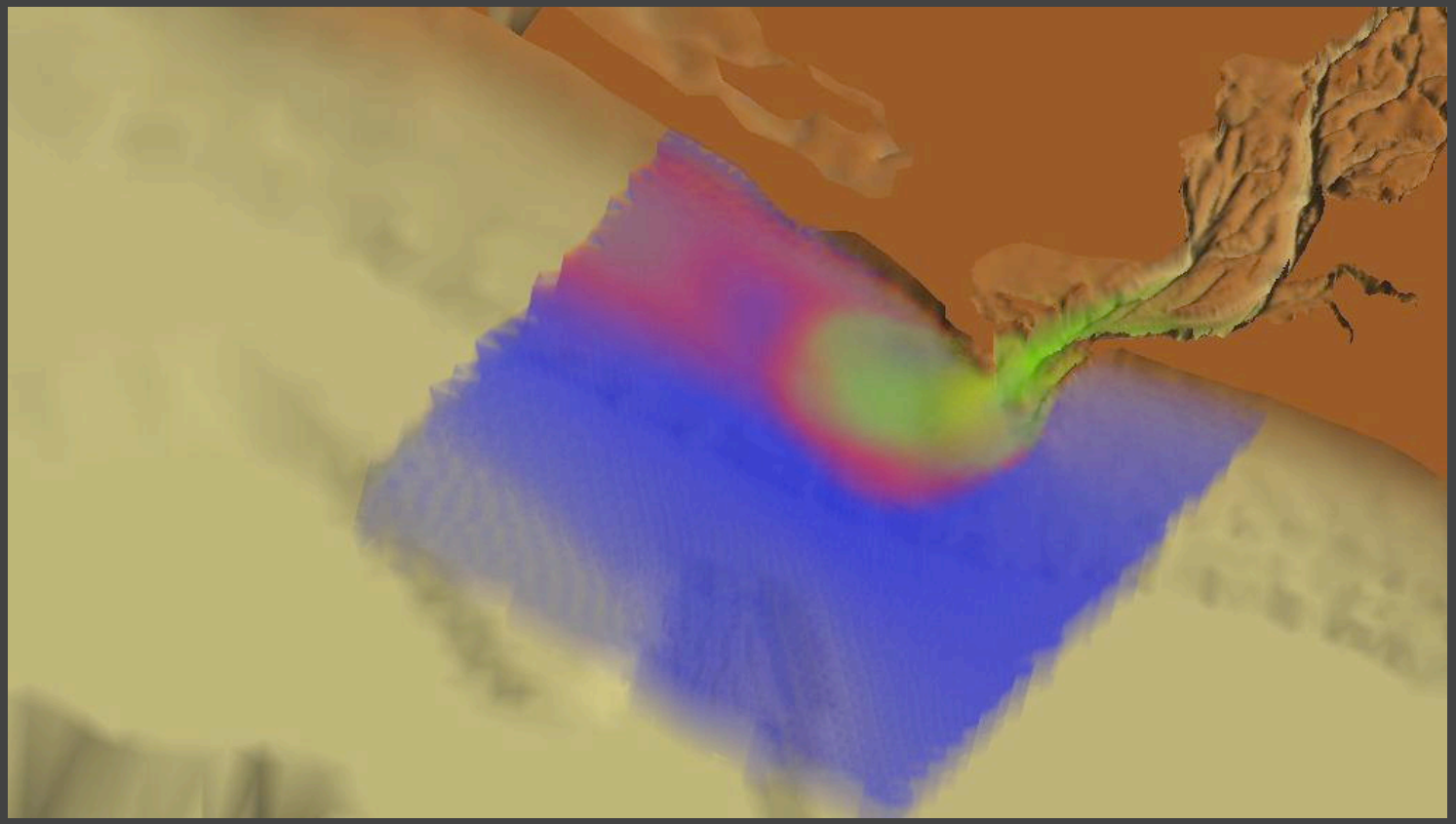
(b) Sorting Phase

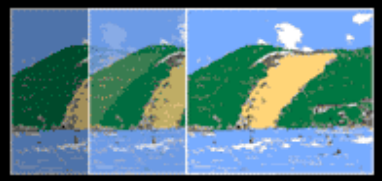




SIBGRAPI 2005

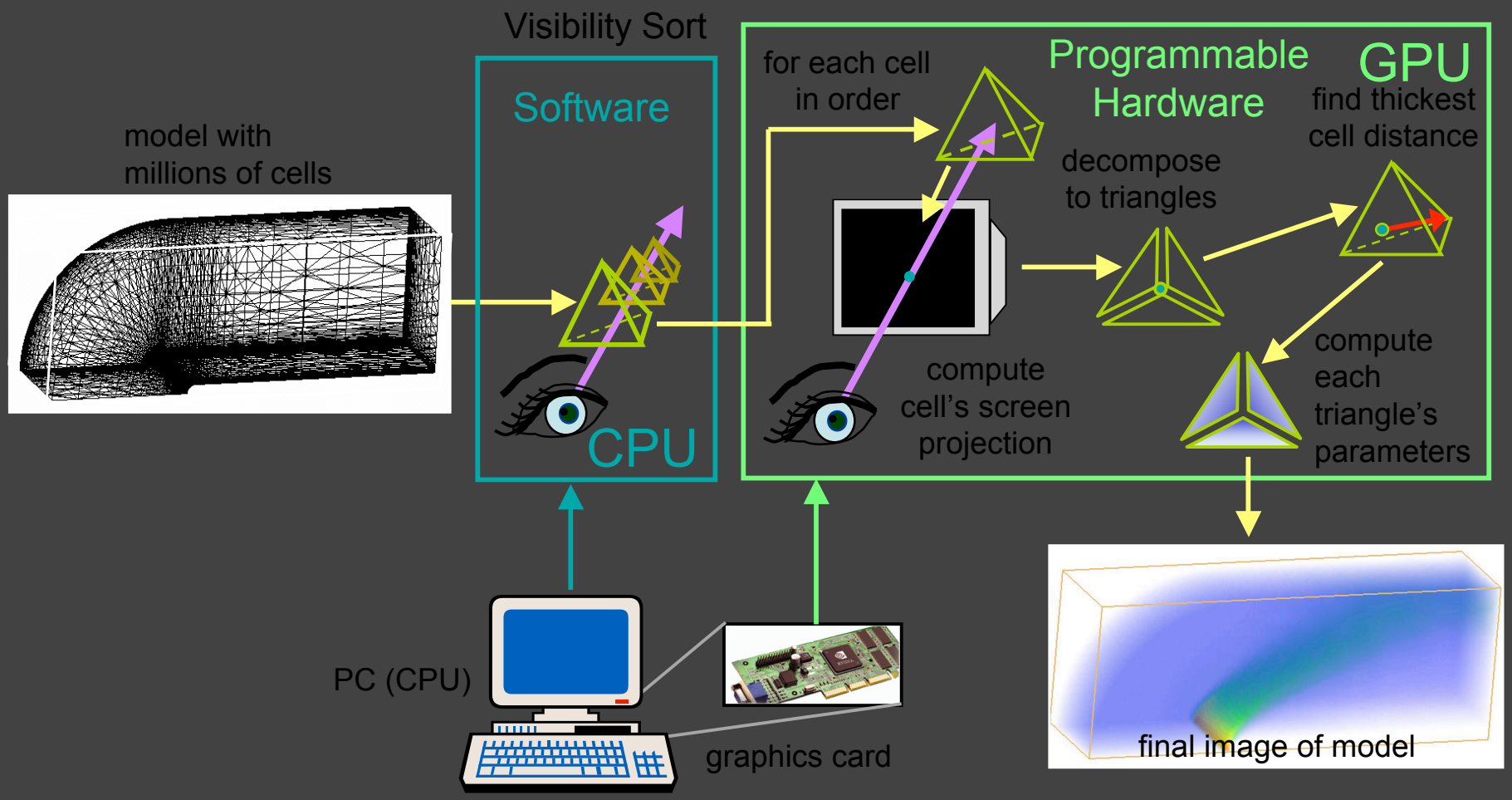
Video

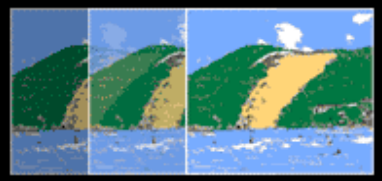




SIBGRAPI 2005

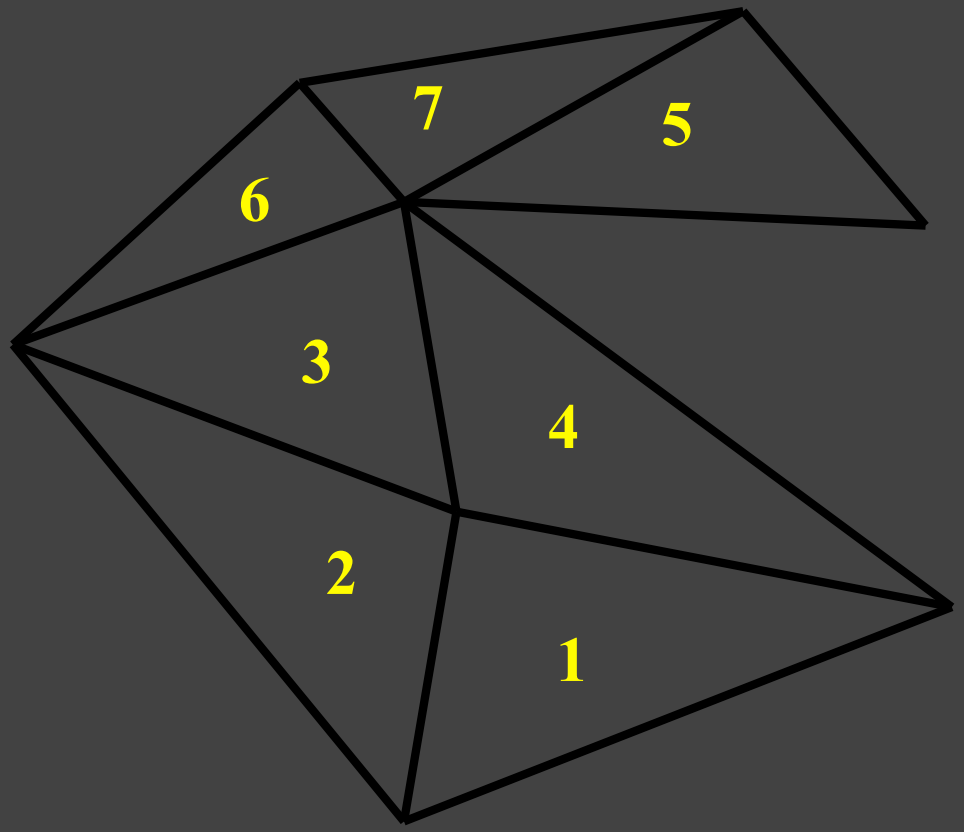
Rendering Unstructured Grids

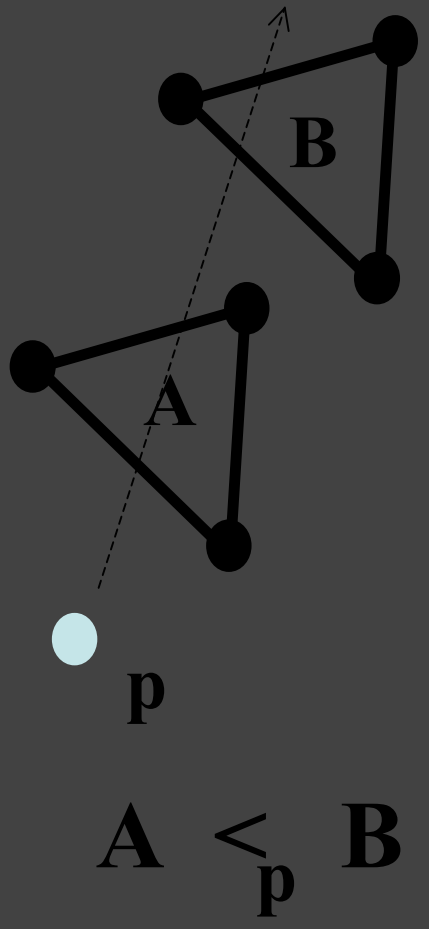


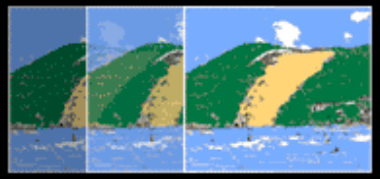


SIBGRAPI 2005

Visibility Sorting

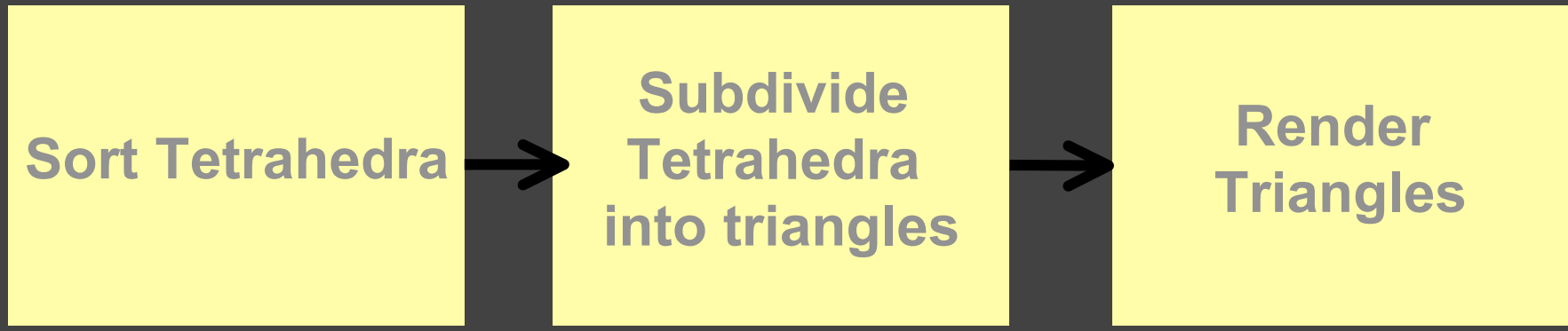


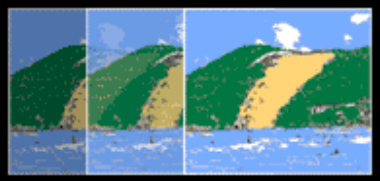




SIBGRAPI 2005

Typical Rendering Pipeline





SIBGRAPI 2005

Existing Techniques

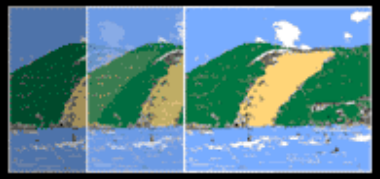
CPU | GPU

Sort Tetrahedra → Subdivide Tetrahedra into triangles → Render Triangles

XMPVO	$O(n+b^2)$
BSP-XMPVO	$O(n+bp)$
SXMPVO	$\sim O(n)$

Shirley-Tuchman 90
Generates ~3.4n triangles

- Given mesh with n cells, b of them on the boundary.



SIBGRAPI 2005

Existing Techniques

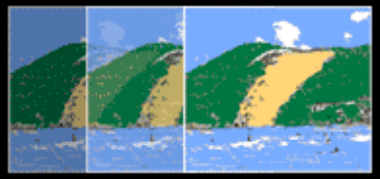
CPU | GPU

Sort Tetrahedra → Subdivide Tetrahedra into triangles → Render Triangles

XMPVO	$O(n+b^2)$
BSP-XMPVO	$O(n+bp)$
SXMPVO	$\sim O(n)$

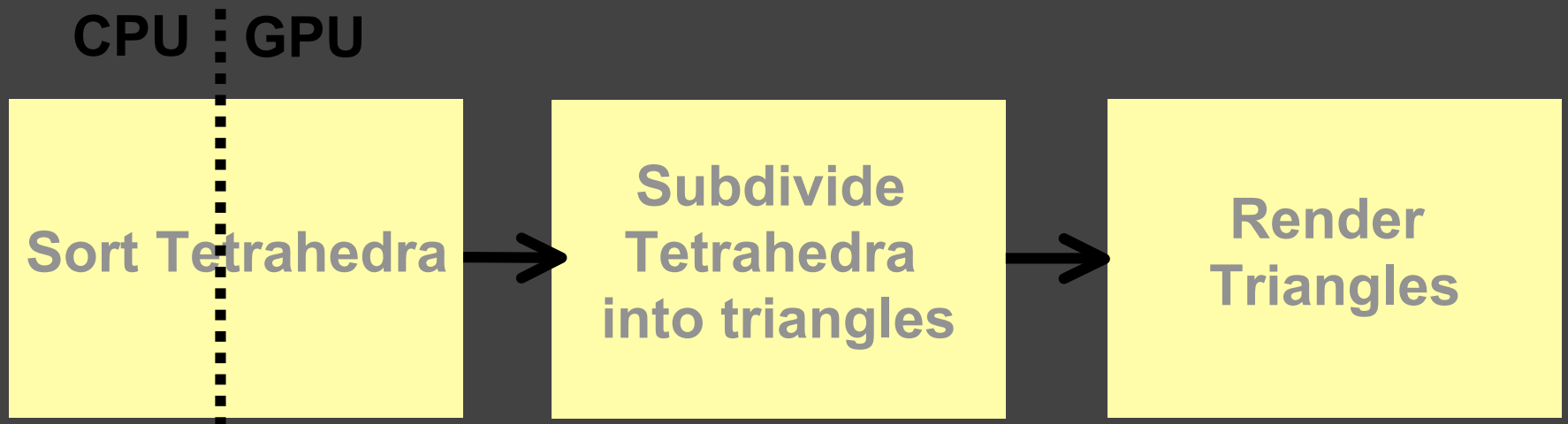
Wylie et al '02
Generates ~3.4n triangles

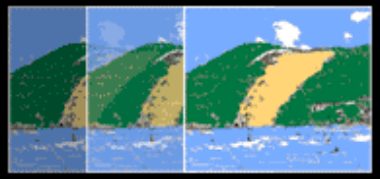
Unfortunately, it is slower in practice because of some implementation overheads!



SIBGRAPI 2005

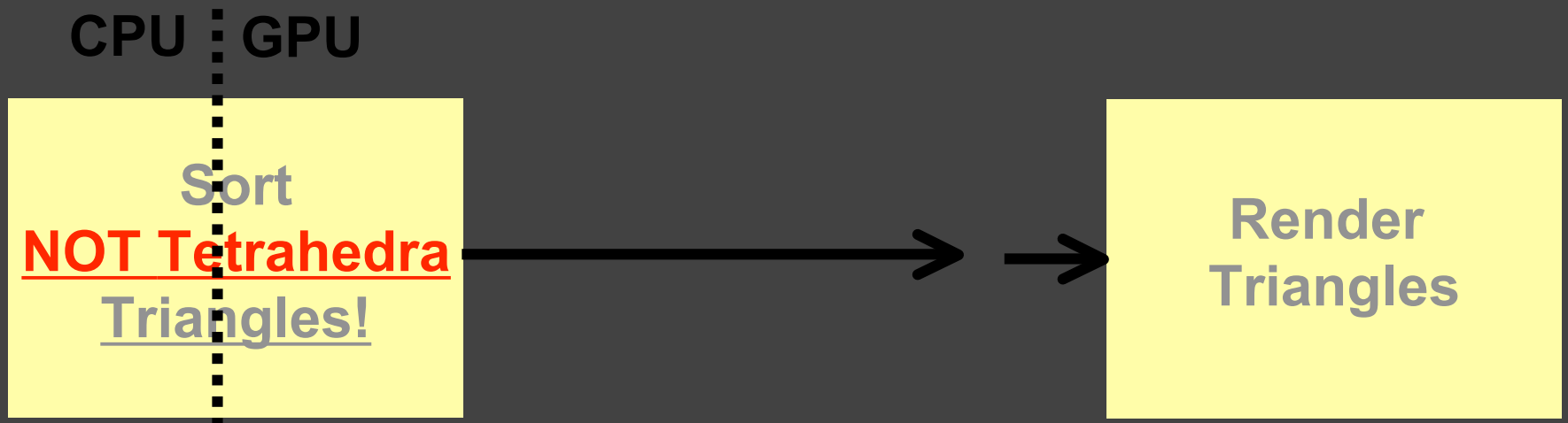
HAVS

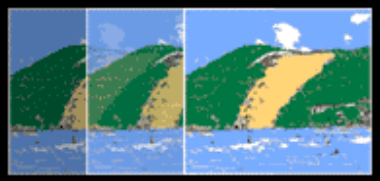




SIBGRAPI 2005

HAVS

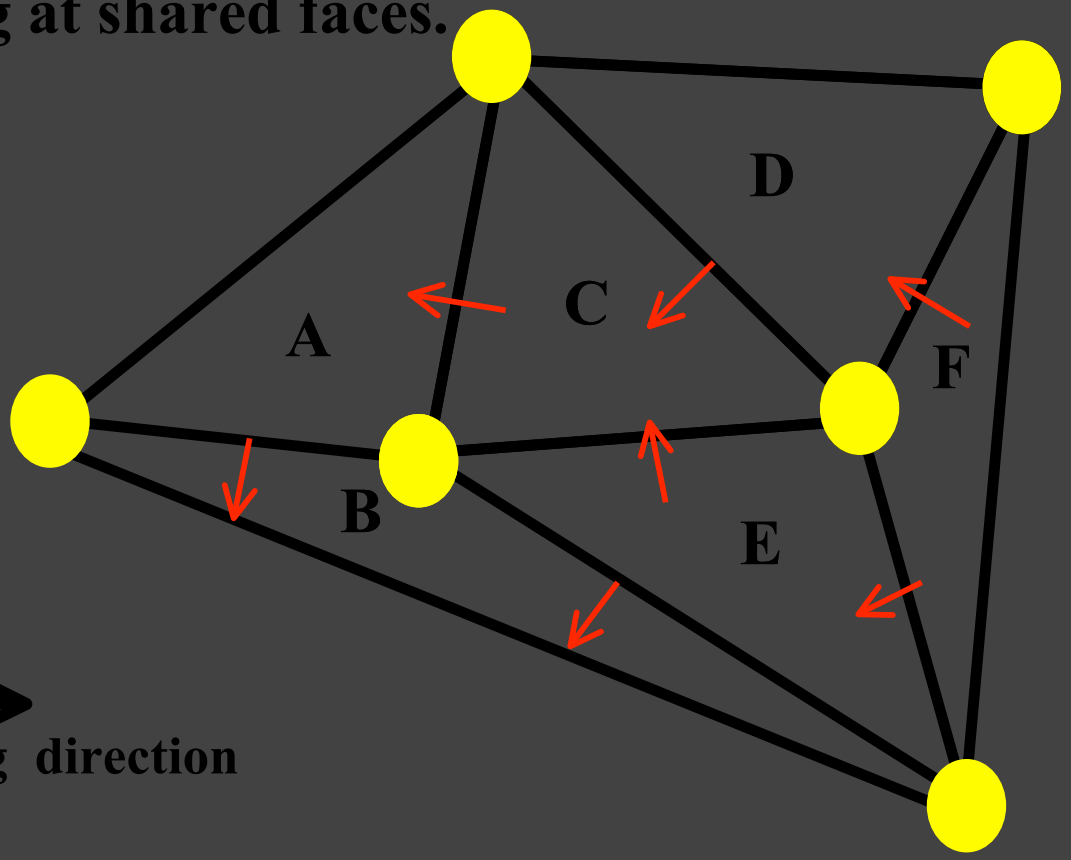




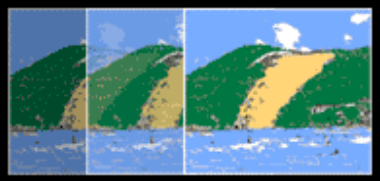
SIBGRAPI 2005

Williams' MPVO

Idea: Define ordering relations by looking at shared faces.



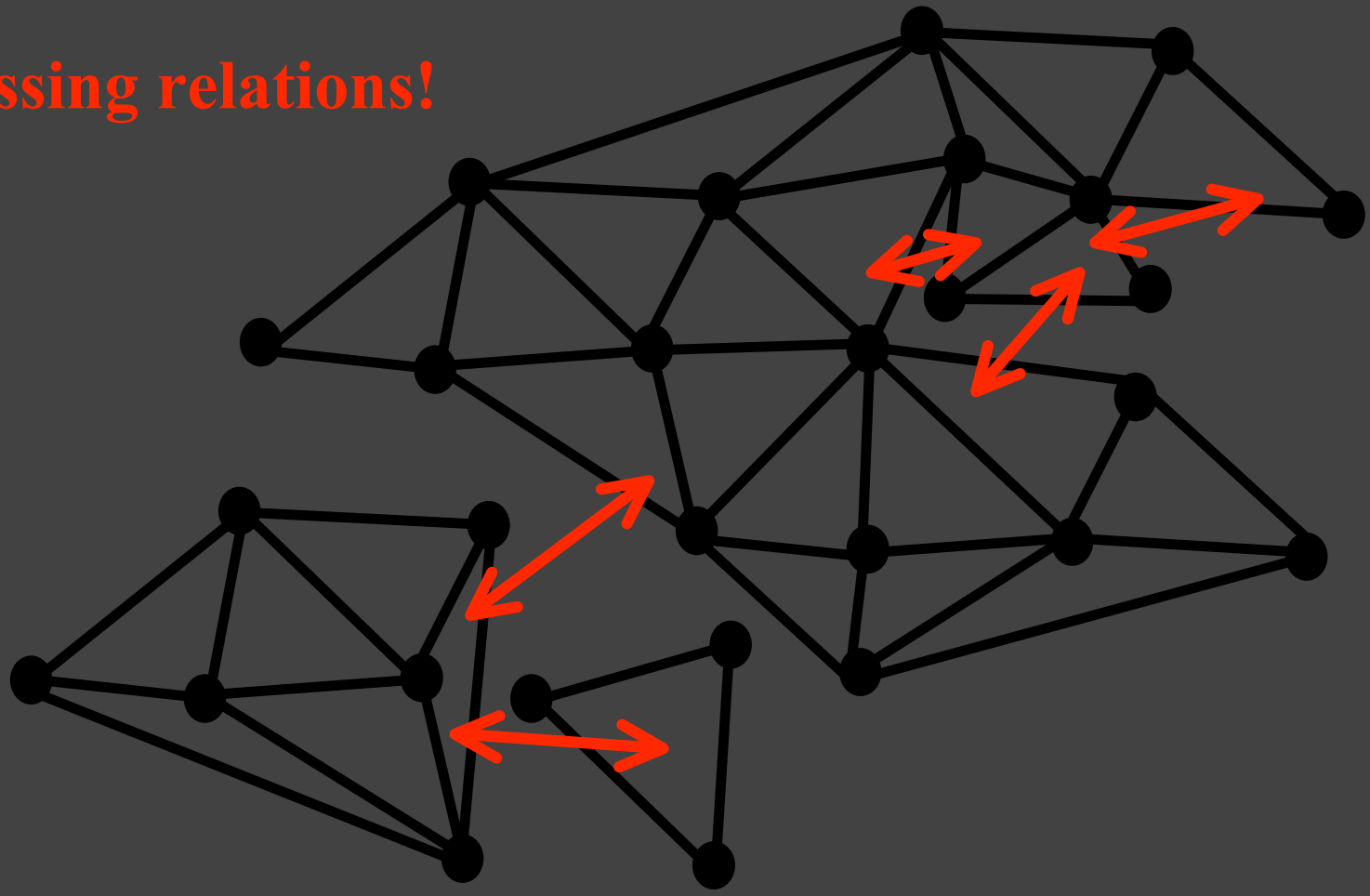
B < A
A < C
B < E
C < E
C < D
E < F
D < F

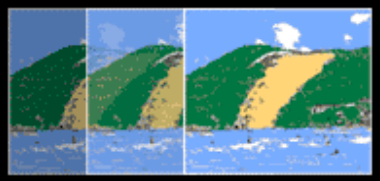


SIBGRAPI 2005

MPVO Limitations

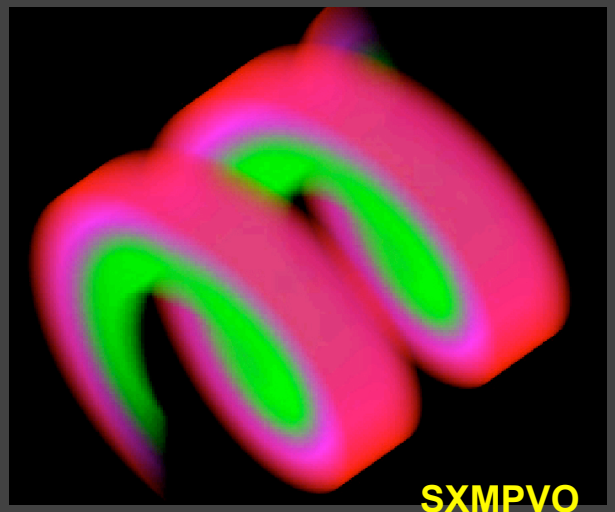
Missing relations!

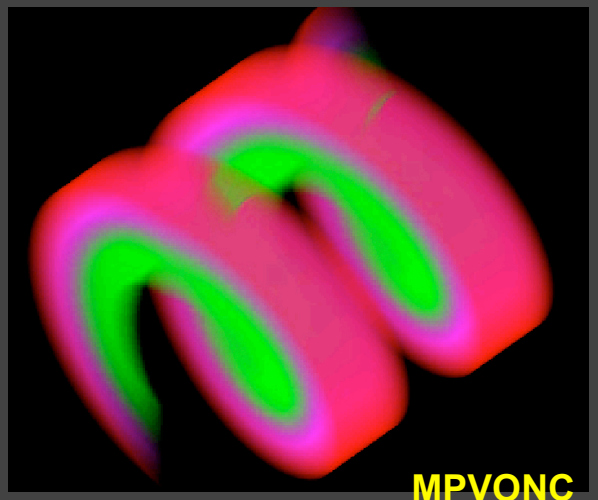


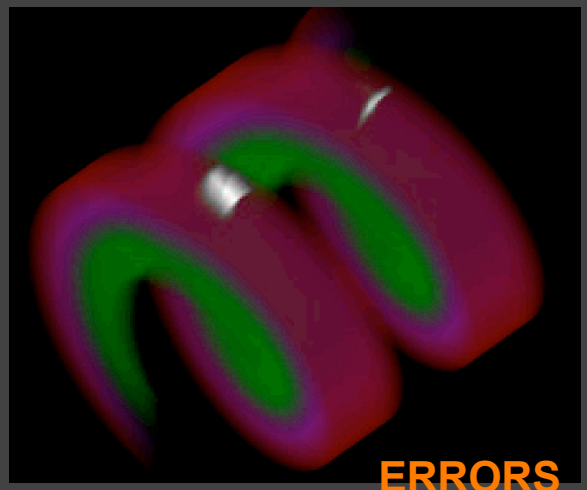


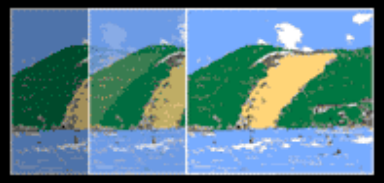
SIBGRAPI 2005

MPVONC Rendering Errors



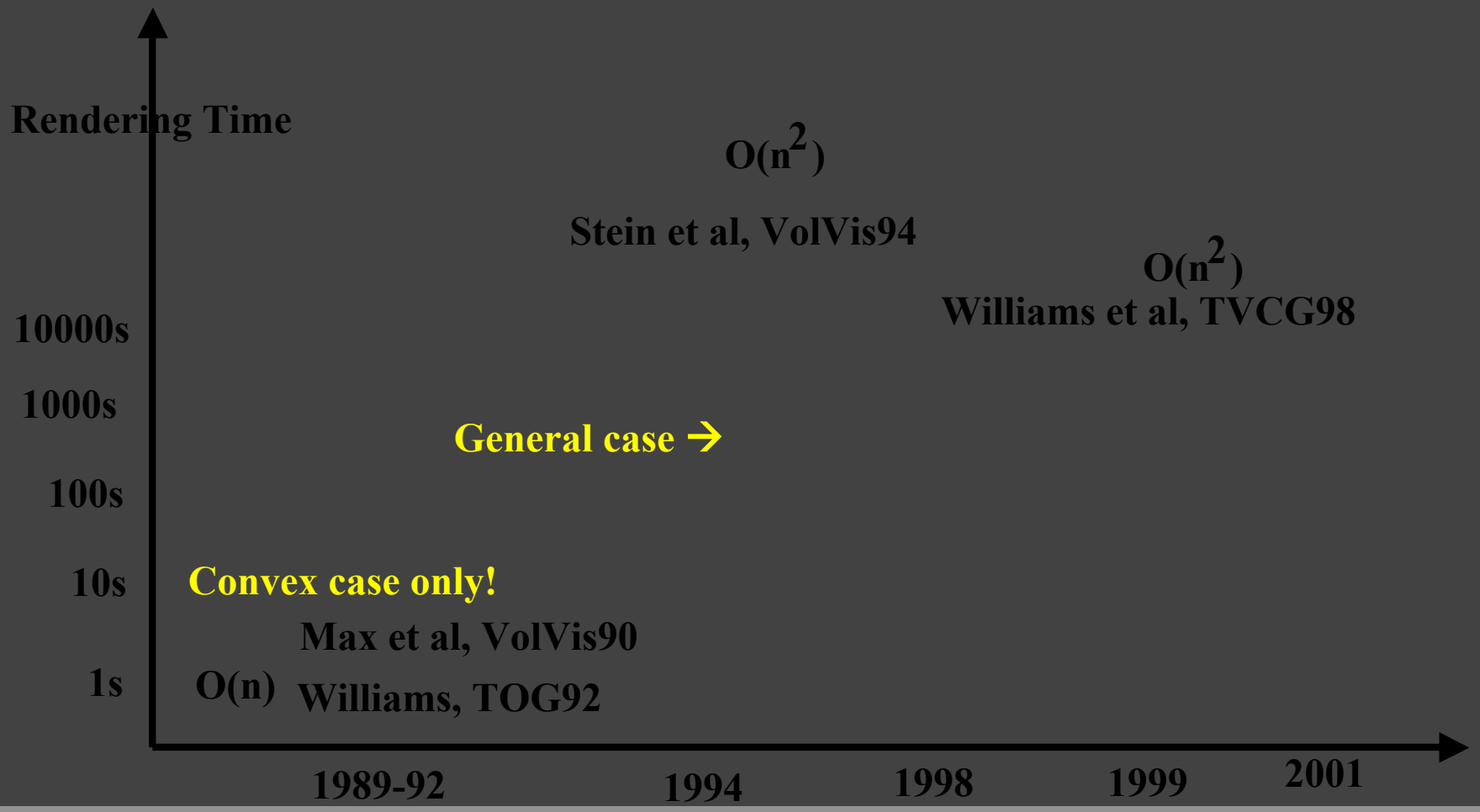


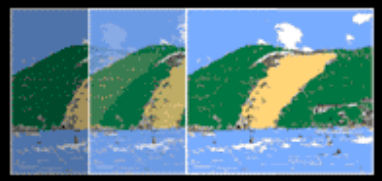




SIBGRAPI 2005

Depth-Sorting Algorithm History

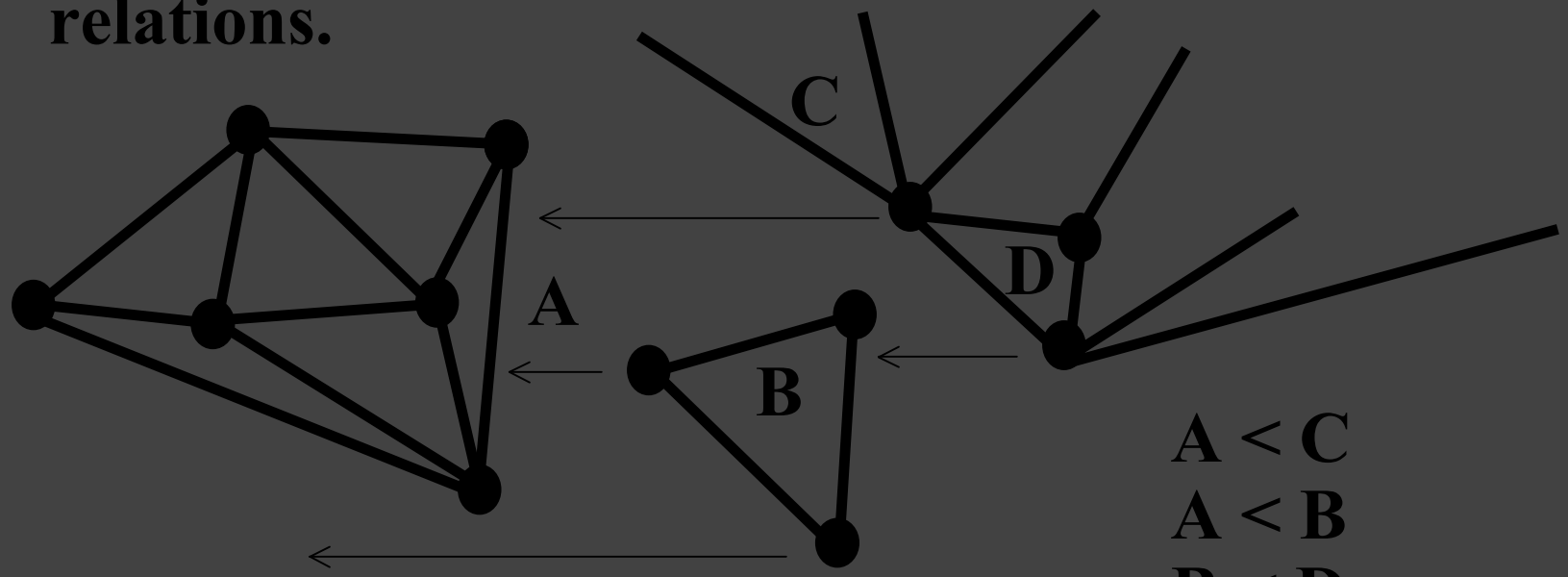


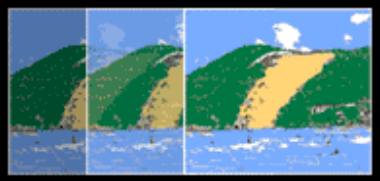


SIBGRAPI 2005

XMPVO

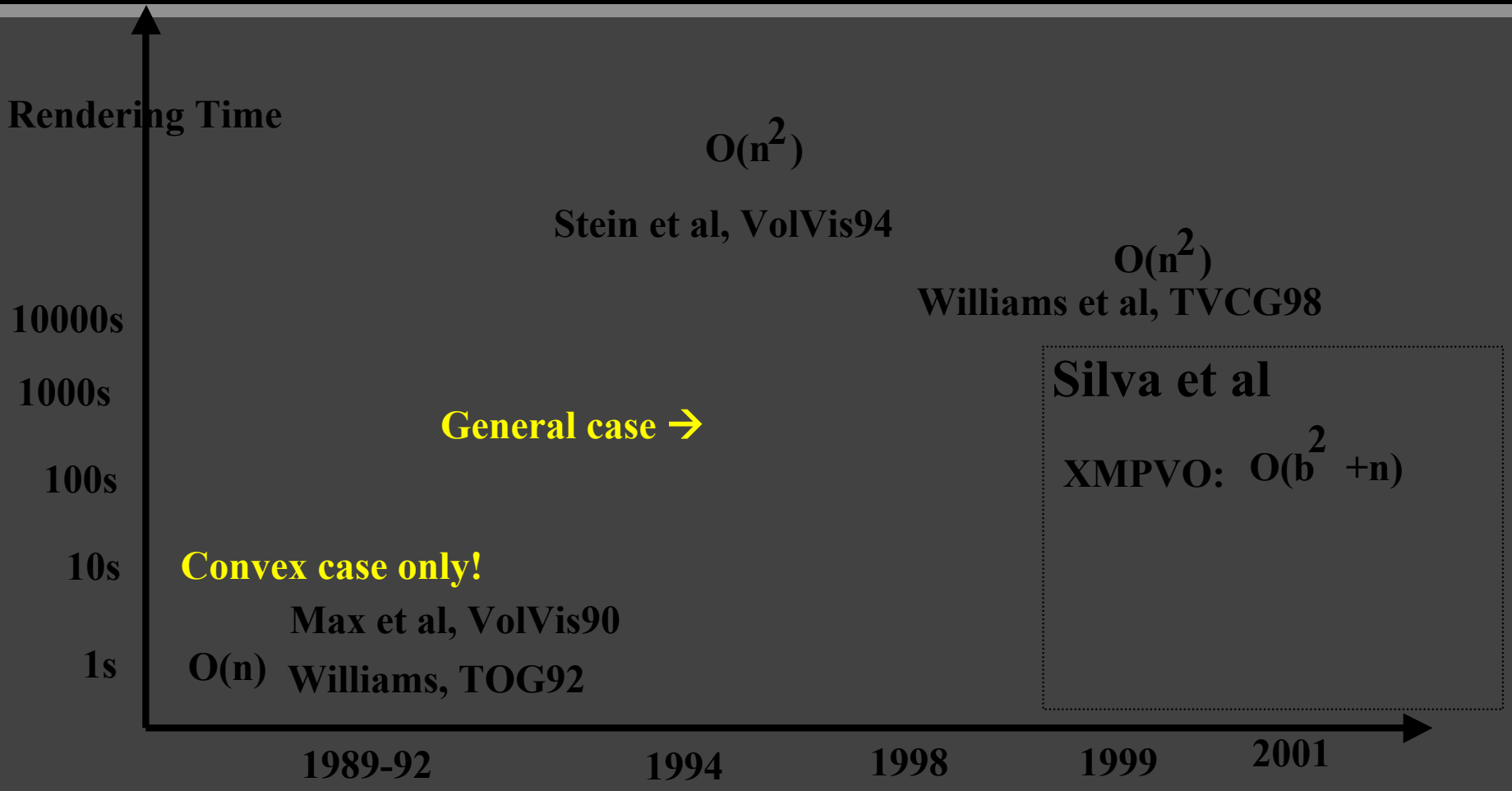
Idea: Using ray shooting queries to complement ordering relations.

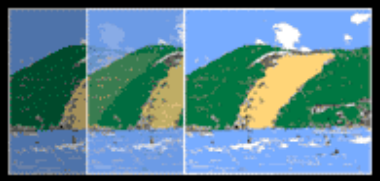




SIBGRAPI 2005

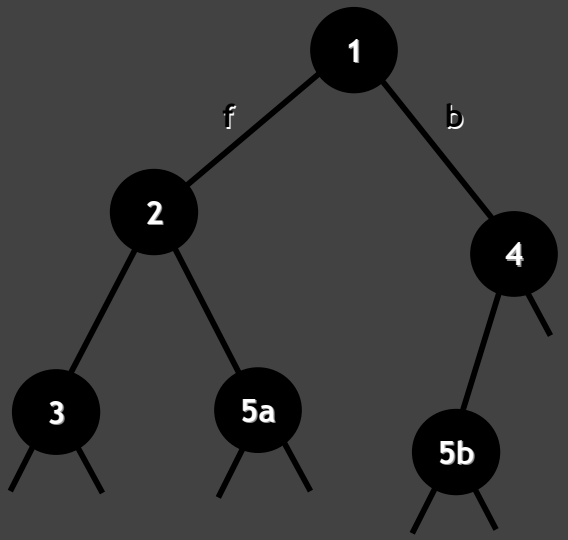
Depth-Sorting Algorithm History

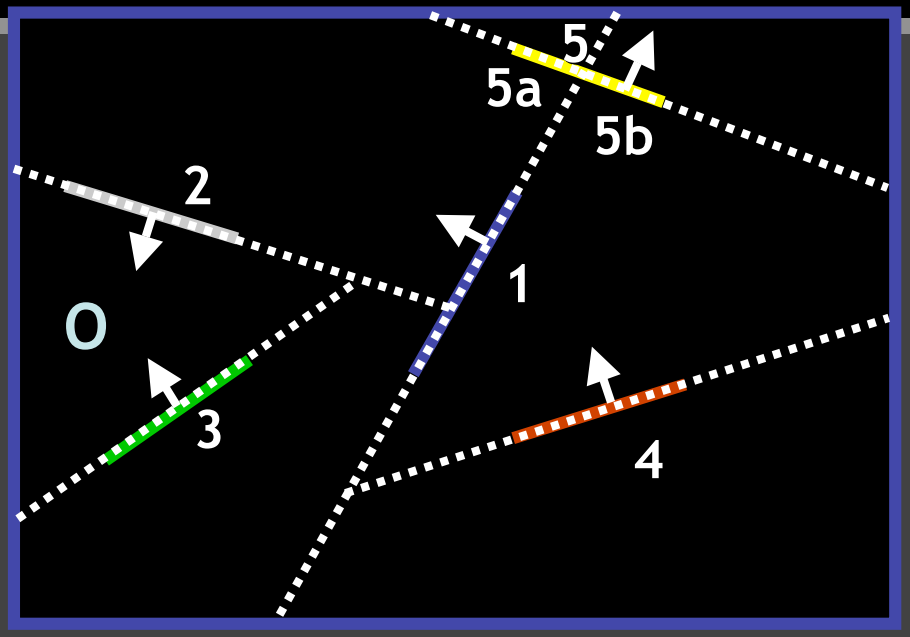




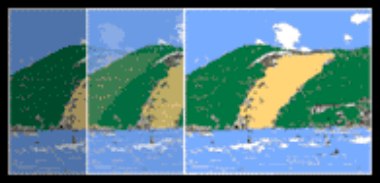
SIBGRAPI 2005

Binary Space Partitioning Trees



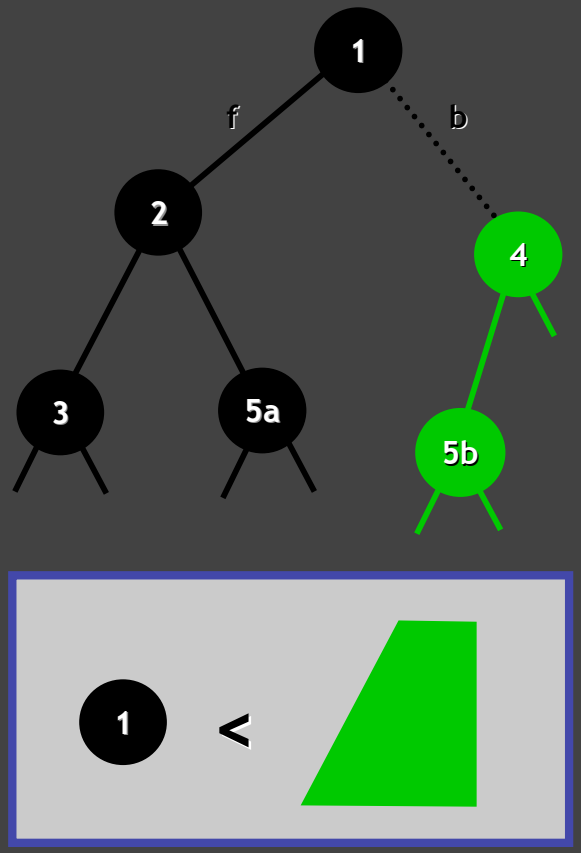


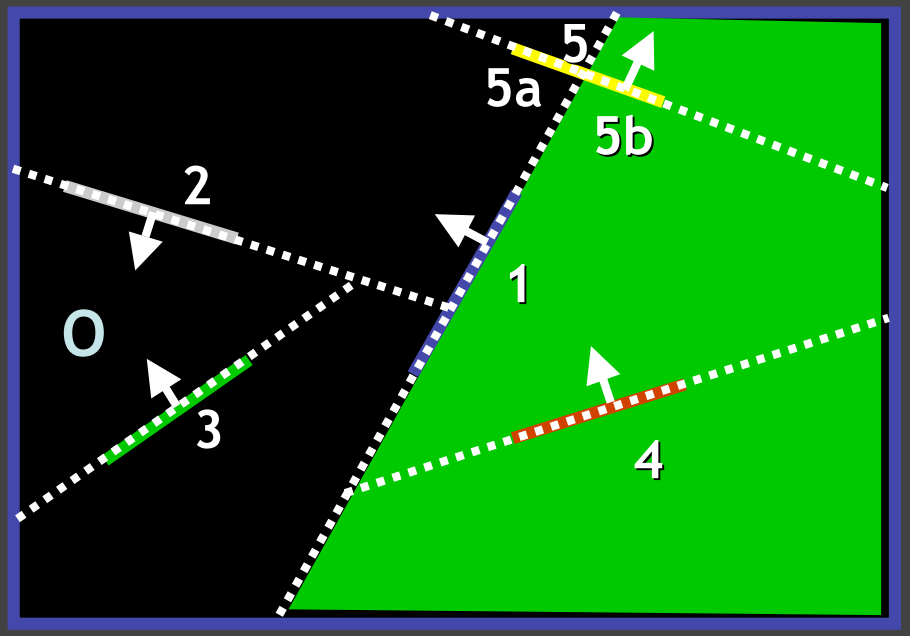
1) Breaks geometry into cycle-free **fragments**
2) Provides a mechanism for computing visibility orders

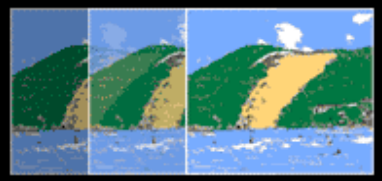


SIBGRAPI 2005

Intuition: BSP for Visibility Ordering

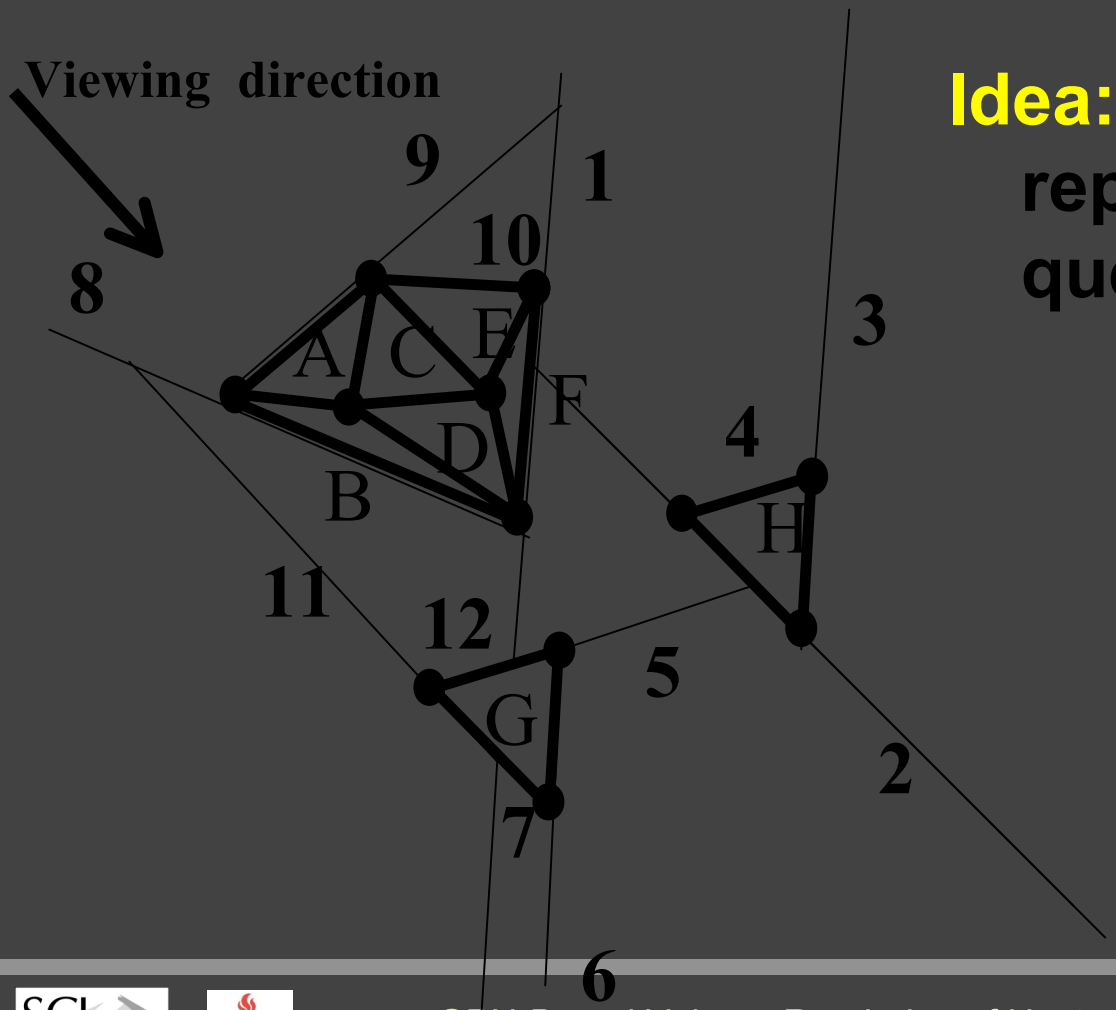




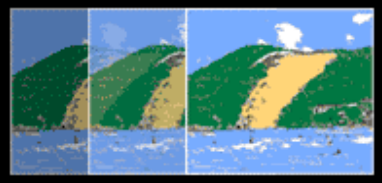


SIBGRAPI 2005

BSP-XMPVO

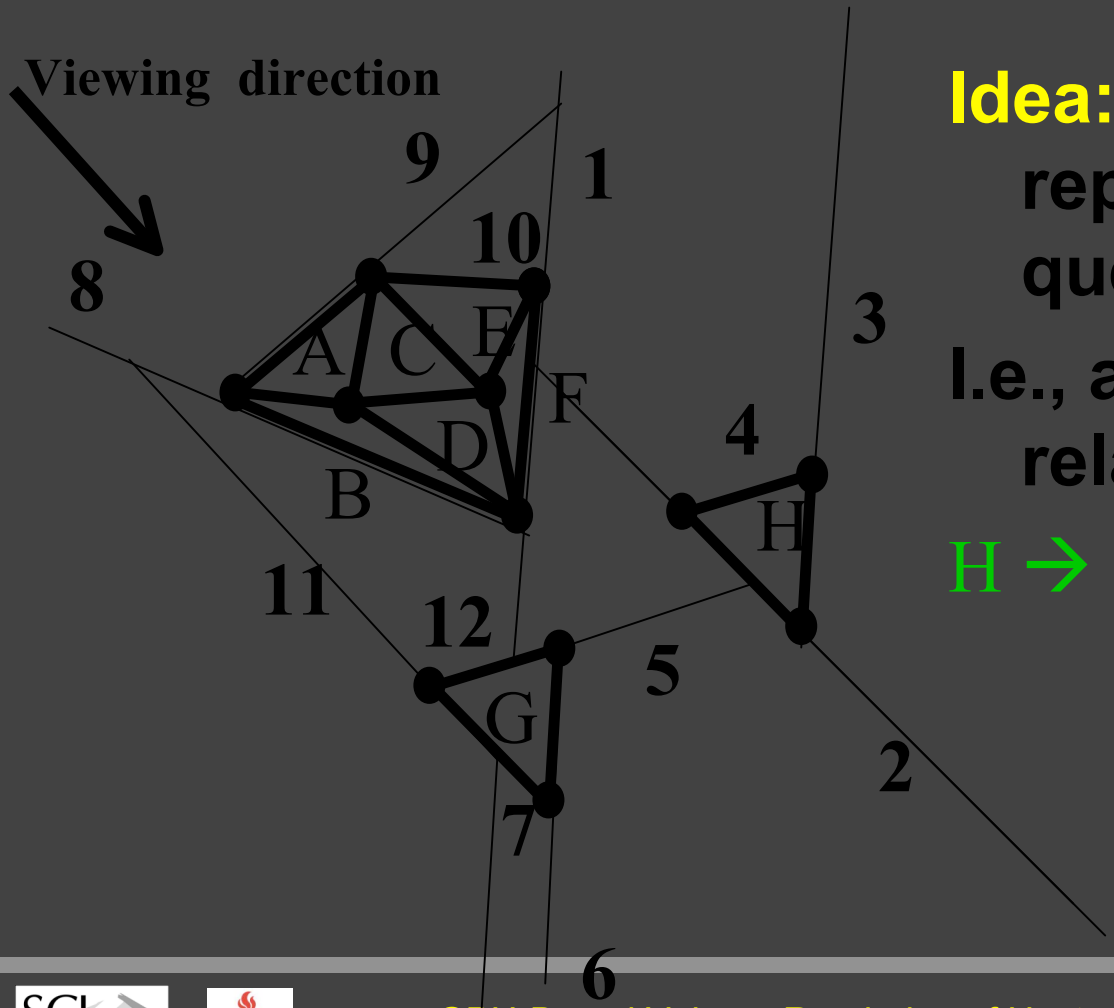


Idea: Use BSP tree to replace ray shooting queries



SIBGRAPI 2005

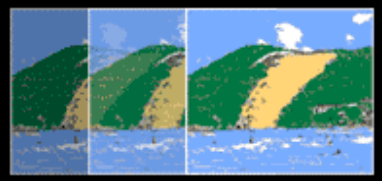
BSP-XMPVO



Idea: Use BSP tree to replace ray shooting queries

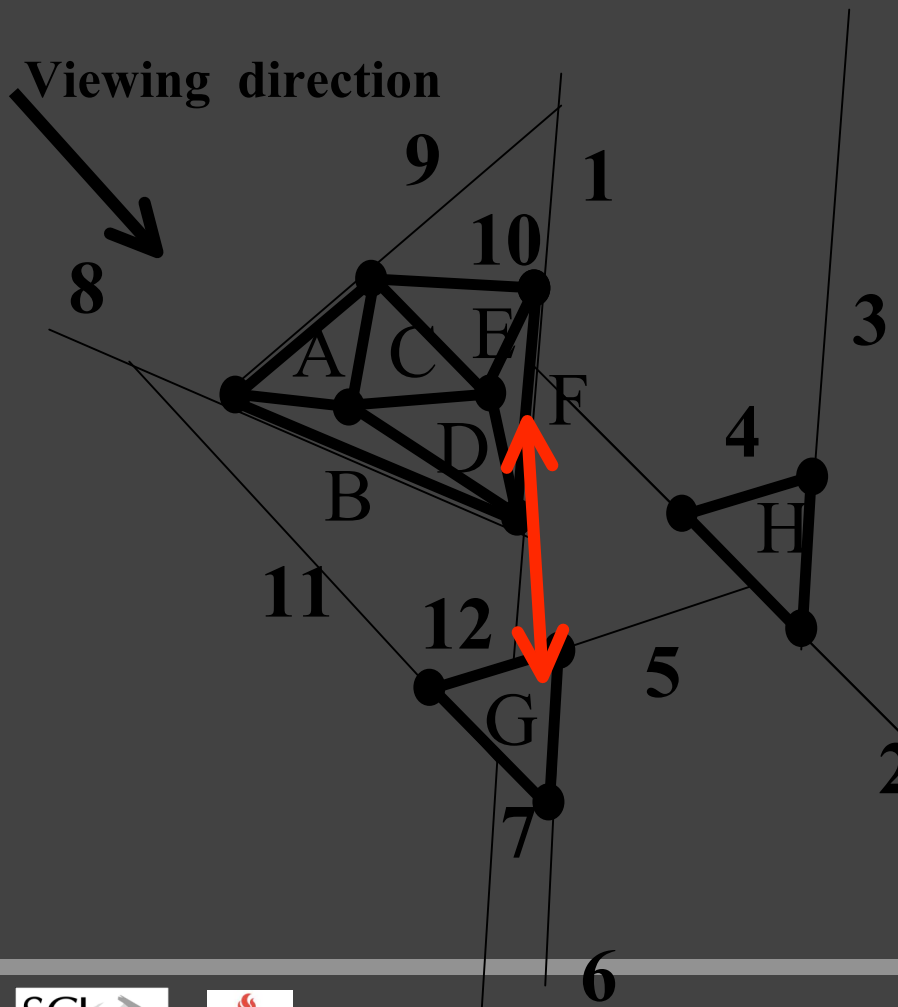
I.e., add extra ordering relation for BSP-tree

H → {2, 3, 4, 5}



SIBGRAPI 2005

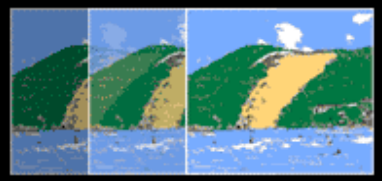
BSP-XMPVO



Idea: Use BSP tree to replace ray shooting queries

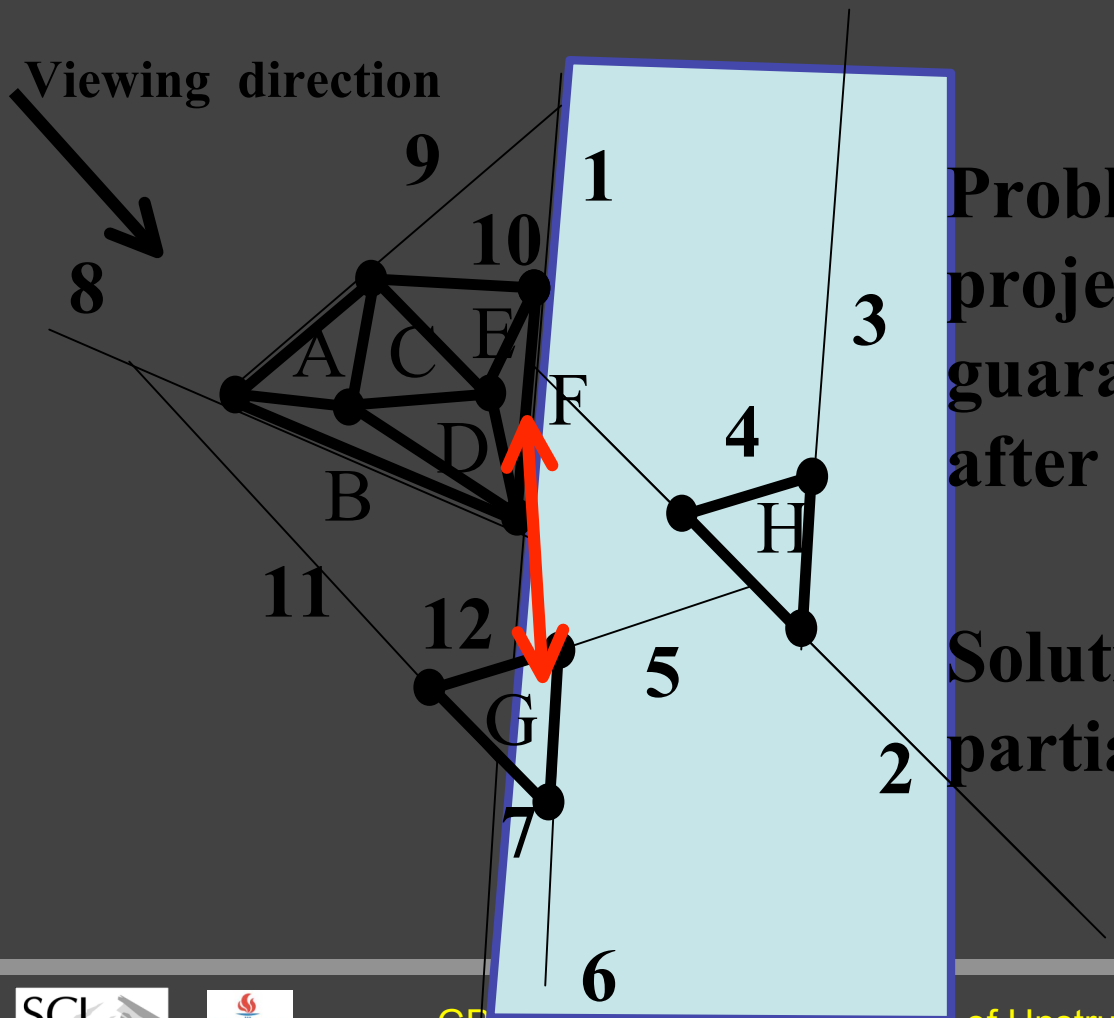
I.e., add extra ordering relation for BSP-tree

But Because of Cell Fragmentation the BSP-tree Does Not Catch all Necessary Ray Shooting Queries



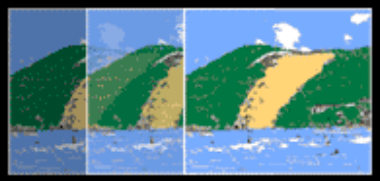
SIBGRAPI 2005

BSP-XMPVO



Problem: G is partially projected, but we need to guarantee that F is projected after G

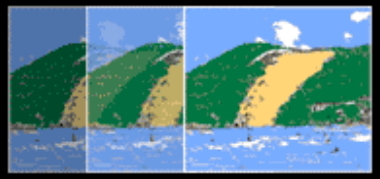
Solution: Keep a list of partially projected cells



SIBGRAPI 2005

BSP-XMPVO relations

- **MPVO dependencies ($<_{adj}$)**
  - **Adjacency relation given by mesh**
- **BSP dependencies ($<_{bsp}$)**
  - **Each fragment c' on the boundary of C define a BSP-dependency for cell C**
- **PPC dependency ($<_{ppc}$)**
  - **If C' is partially projected and C' lies behind cell C, then we create a PPC dependency for C**



SIBGRAPI 2005

BSP-XMPVO Algorithm

BSP Traversal

```
Algorithm BSP-XMPVO_traversal(node, vp)
    /* The algorithm projects in back-to-front
       order the part of the mesh S
       corresponding to BSP tree node node
       with respect to the viewpoint vp. */
1.  if (node == NULL) then
2.      return;
3.  if (vp is in front plane)
4.      BSP-XMPVO_traversal(back(node));
5.      BSP-XMPVO_update_dep(node);
6.      BSP-XMPVO_traversal(front(node);
7.  else
8.      BSP-XMPVO_traversal(front(node);
9.      BSP-XMPVO_update_dep(node);
10.     BSP-XMPVO_traversal(back(node));
```

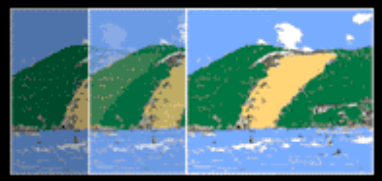
Update Graph Dep.

```
Algorithm BSP-XMPVO_update_dep(node)
    /* Updates the dependency counters
       for the cells whose faces
       lie on node's base plane. */
1.  for (i = 0; i < numPPC; i++)
2.      for (j = 0; j < numCutCells(node); j++)
3.          Check_update_ppc_dep_count (Ci,Cj);
4.  for (i = 0; i < numCutCells(node); i++)
5.      Update_PPC (Ci);
6.  for (i = 0; i < numCutCells(node); i++)
7.      Decrem_bsp_dep_count(Ci);
8.      if (num_inbound(Ci) == 0) and
9.        (bsp_dep_count(Ci) == 0) and
10.       (ppc_dep_count(Ci) == 0)
11.         enqueue(Ci);
12. MPVO_traverse();
```

Williams' MPVO

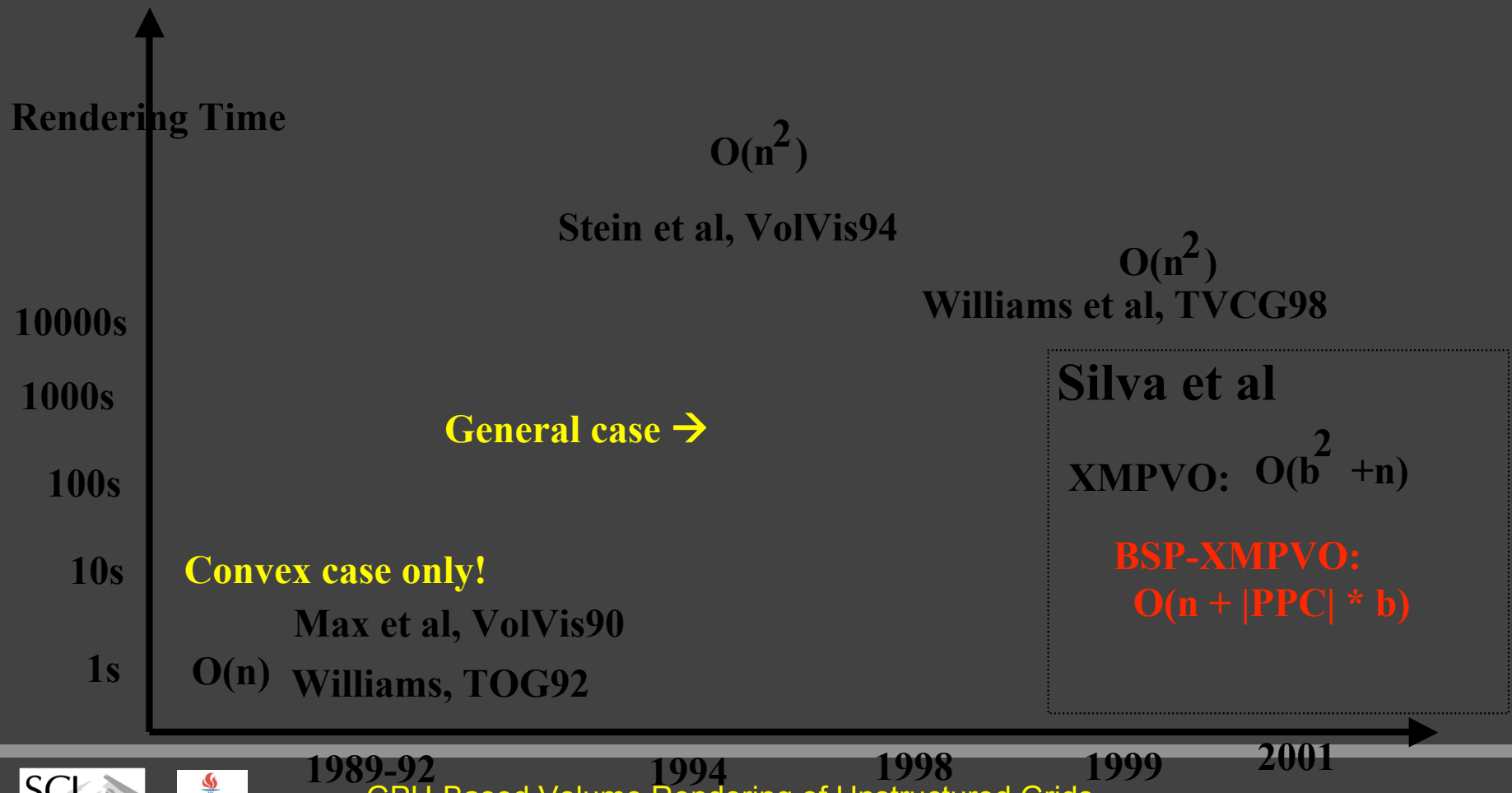
```
Algorithm MPVO_traverse()
    /* Modified MPVO traverse. */
1. while (deque(c) != false)
2.     output(C);
3.     for (i = 0; i < numFaces(C); i++)
4.         if arrow(i, C) == INBOUND
5.             continue;
6.         Ci = neighbor(C, i);
7.         Decrem_num_inbound(Ci);
8.         if ((num_inbound(Ci) == 0) and
9.           (bsp_dep_count(Ci) == 0) and
10.          (ppc_dep_count(Ci) == 0))
11.              enqueue(Ci);
```

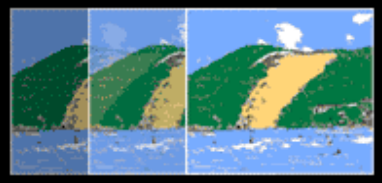
See paper for proof of correctness!



SIBGRAPI 2005

Depth-Sorting Algorithm History





SIBGRAPI 2005

Depth-Sorting Algorithm History

Rendering Time

10000s

1000s

100s

10s

1s

$O(n^2)$

Stein et al, VolVis94

$O(n^2)$

Williams et al, TVCG98

General case →

Silva et al

XMPVO: $O(b^2 + n)$

BSP-XMPVO:
O(n + |PPC| * b)

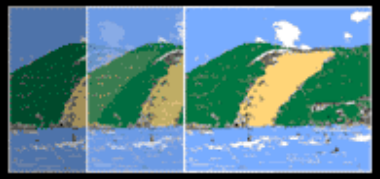
SXMPVO: O(n)

Convex case only!

Max et al, VolVis90

O(n) Williams, TOG92

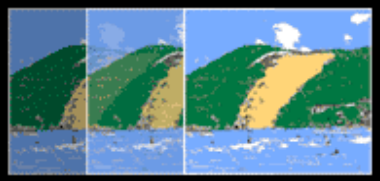
1989-92 1994 1998 1999 2001



SIBGRAPI 2005

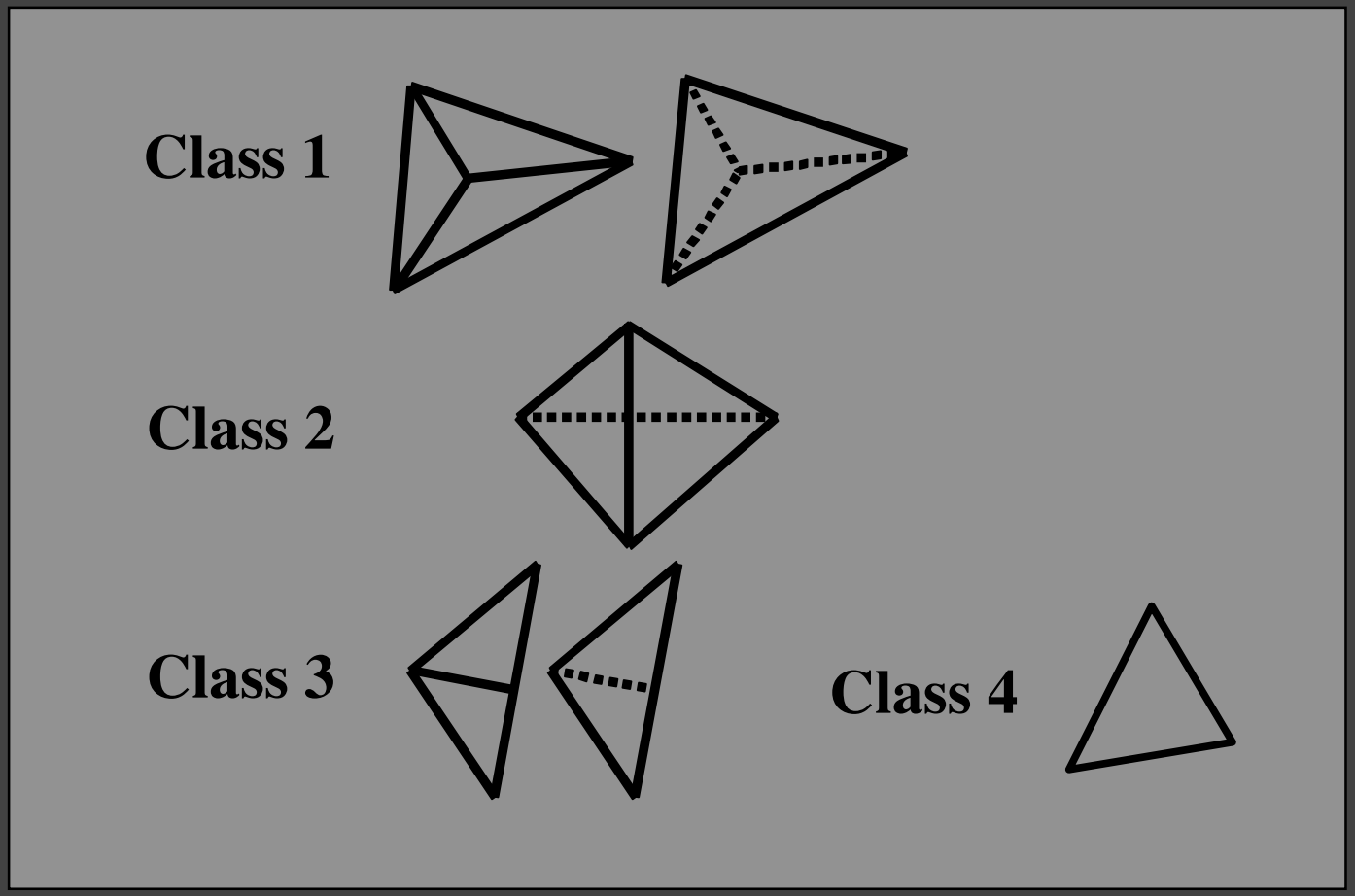
Outline

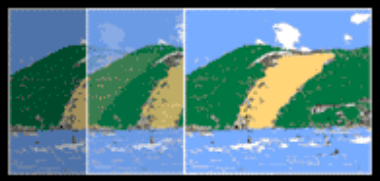
- Introduction to Volume Rendering
- Polyhedral Cell Sorting
- Hardware-Assisted Techniques



SIBGRAPI 2005

Shirley-Tuchman (ST) Algorithm

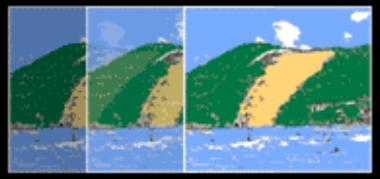




SIBGRAPI 2005

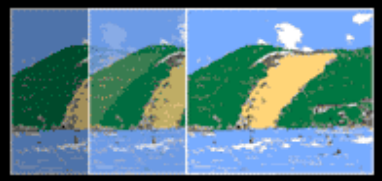
Wylie et al's GPU-based ST

- Moves all of the following functions from the CPU the GPU:
  - Transform to screen space
  - Determine projection class
  - Calculate thick vertex location
  - Determine depth at thick vertex
  - Compute color and opacity for thick vertex
  - Apply exponential attenuation texture



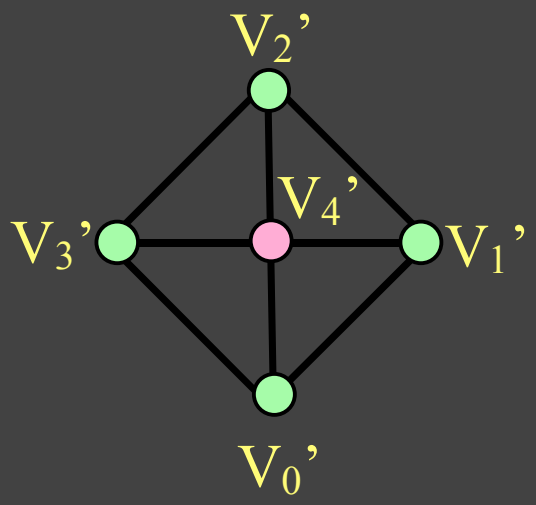
GPU Limitations

- Each instance of a *vertex shader* program works independently on a single vertex in SIMD fashion
- No support dynamic vertex creation or topology modification within the vertex program
- No branching (at the time!)
- No knowledge of neighboring vertices
- Cannot change execution based on past information



SIBGRAPI 2005

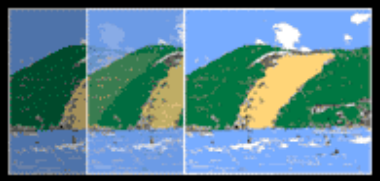
Idea: Morph a Canonical Graph



Basis Graph

Isomorphic to all projection cases

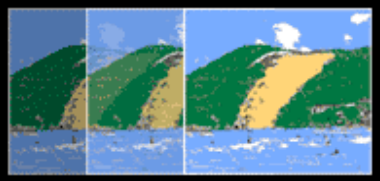
Example later…



SIBGRAPI 2005

PT algorithm in Vertex Program

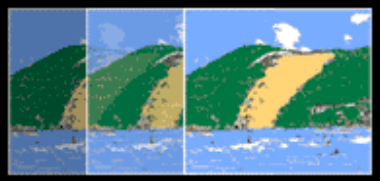
- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.



SIBGRAPI 2005

PT algorithm in Vertex Program

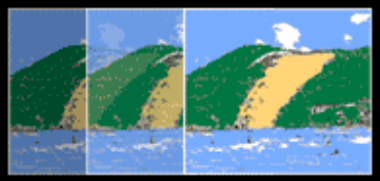
- Transform to screen space. (Trivial)
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.



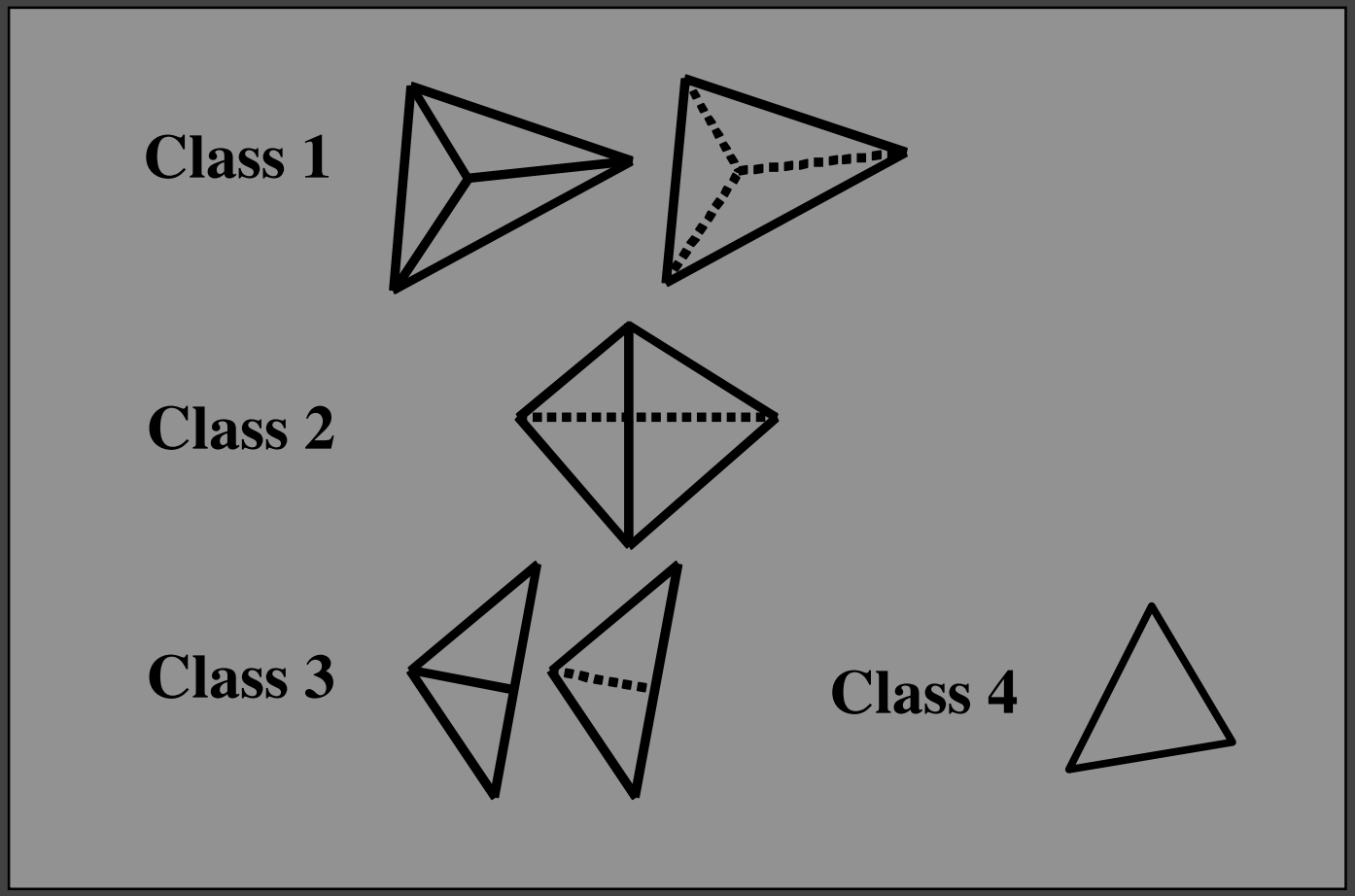
SIBGRAPI 2005

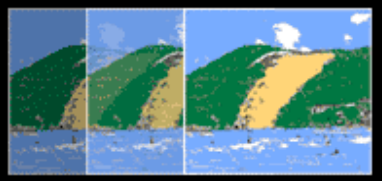
PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.



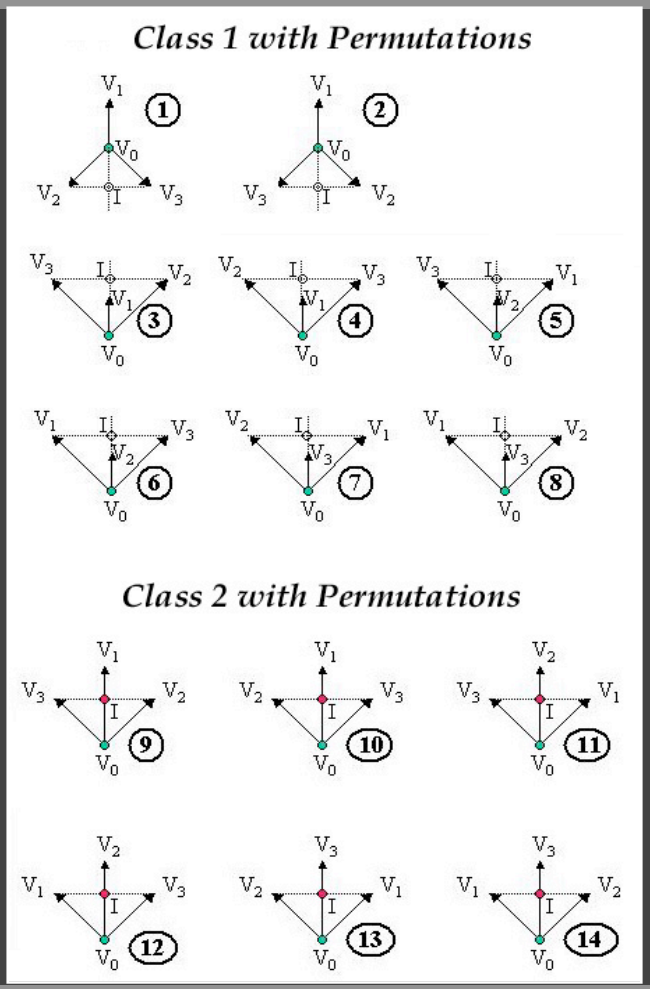
Projection Classes





SIBGRAPI 2005

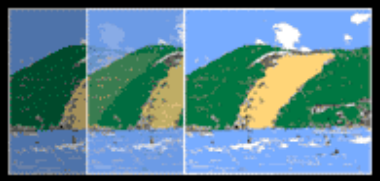
Projection Permutations



- Permutation Determination
- 14 cases need at least 4 Boolean tests

- Definitions
- vec1 = v1-v0
- vec2 = v2-v0
- vec3 = v3-v0
- cross1 = vec1 x vec2
- cross2 = vec1 x vec3
- cross3 = vec2 x vec3

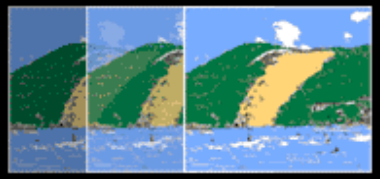
- Tests
- test1 = (cross1*cross2 < 0)
- test2 = (cross1*cross3 > 0)
- test3 = (distance from v0 to middle vertex –
- distance from v0 to Intersection) > 0
- test4 = (cross1 > 0)



SIBGRAPI 2005

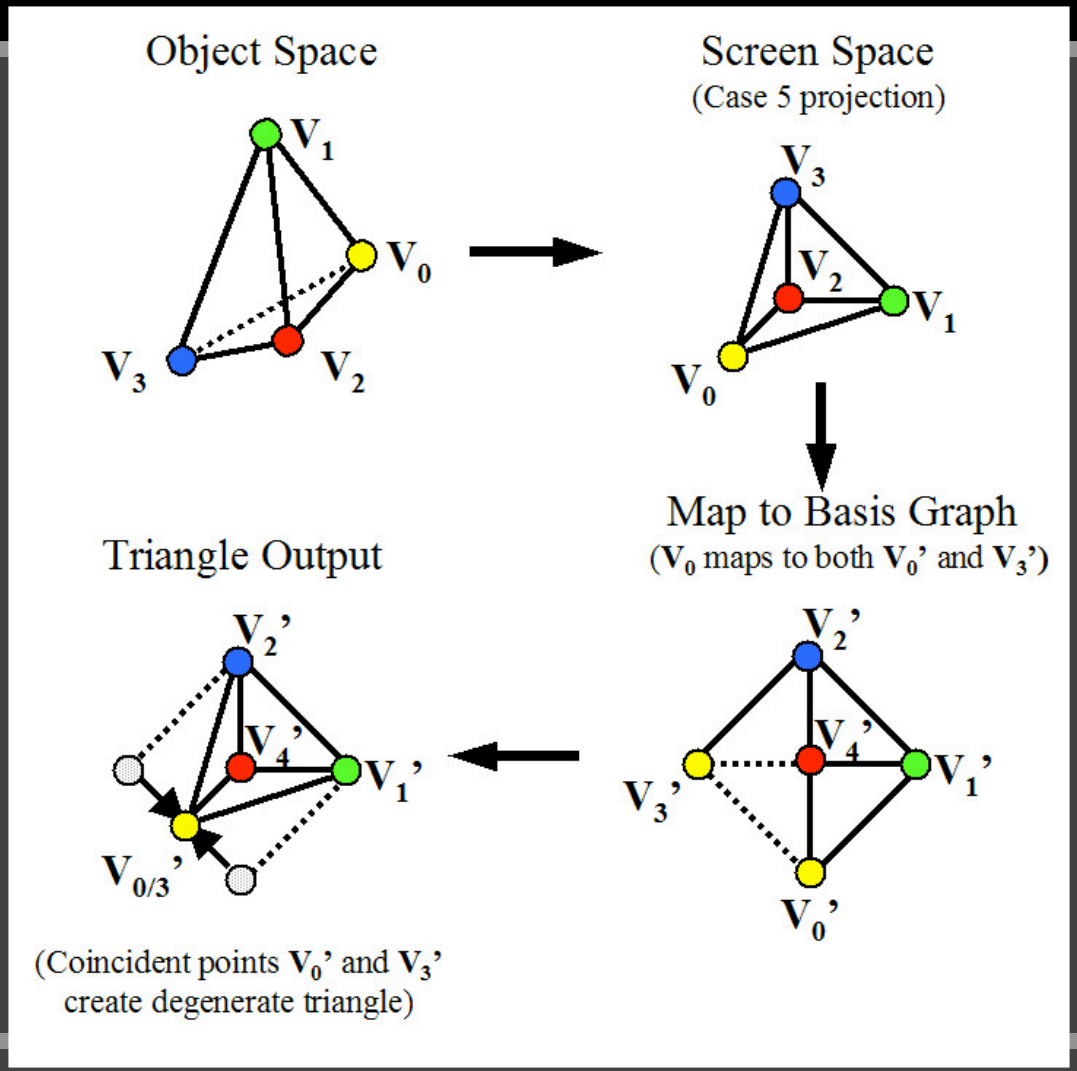
PT algorithm in Vertex Program

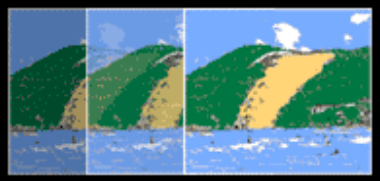
- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.



SIBGRAPI 2005

Isomorphic Property of Basis Graph

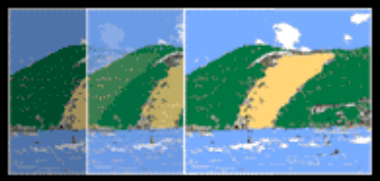




SIBGRAPI 2005

PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.

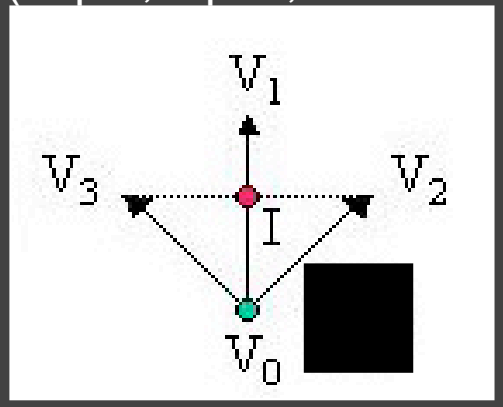


SIBGRAPI 2005

Thick Vertex Calculation

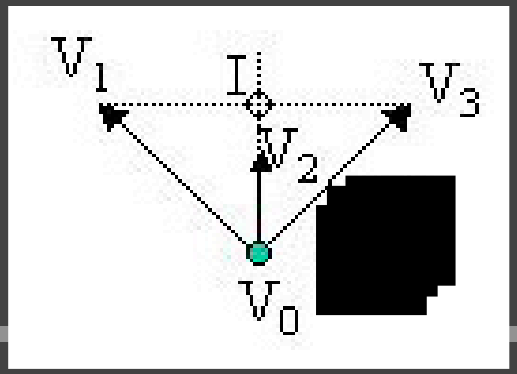
- In all cases the coordinates of the intersection point I are computed. (Intersection of lines computed ala Graphics Gems III p. 199-202). This intersection calculation gives us α and β terms that are used for interpolation (depth, alpha, and color) later on.

- Class 2:

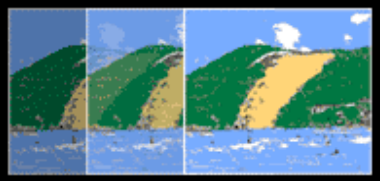



```
// Compute thick vertex "thickness"
float z1 = P1[2] + alpha * (P2[2] - P1[2]);
float z2 = P3[2] + beta  * (P4[2] - P3[2]);
float thickness = fabs(z1-z2);
```

- Class 1:



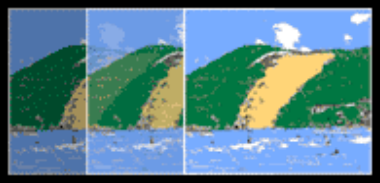

```
// Extra computation for class 1
if (!test3) thickness /= alpha;
```



SIBGRAPI 2005

PT algorithm in Vertex Program

- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.

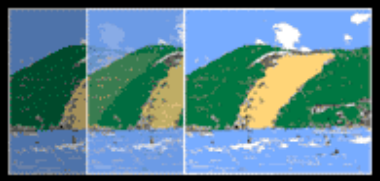


SIBGRAPI 2005

Color and Opacity Calculation

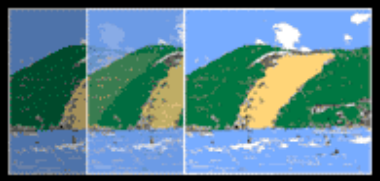
- Use same α and β terms to interpolate color and opacity along the line segments to give the front and back face intersection terms C_F, C_B and τ_F, τ_B.
- Thick vertex color $(C_F + C_B) / 2$ *
- The extinction coefficient τ is $(\tau_F + \tau_B) / 2$.
- τ and the thickness l, are then used as lookups into a 2D texture map defined as $1 - \exp(-\tau l)$. [Stein et al. 1994].

* approximate color from Shirley and Tuchman.



PT algorithm in Vertex Program

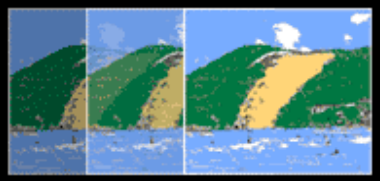
- Transform to screen space.
- Determine projection class (and permutation).
- Map the vertices to the *basis graph*.
- Calculate intersection point of line segments.
- Determine depth at thick vertex.
- Compute color and opacity for thick vertex (texture)
- Multiplex the result to correct output vertex.



Multiplex input to output

- Use a lookup table (loaded in the parameter registers)
- and an index based on the 4 tests to determine the
- output vertex.

```
// Which vertex to copy to output (using lookup table)
lookup_index = test1*8 + test2*4 + test3*2 + test4;
output_vertex = lookup_table[call_index][lookup_index];
```

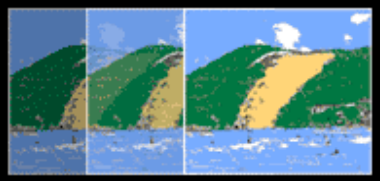


SIBGRAPI 2005

‘Feeding’ the Vertex Program

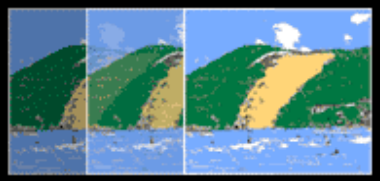
```
// Load up the 4 vertices
glVertexAttrib3fvNV(1, nodes[0]->getXYZ());
glVertexAttrib3fvNV(2, nodes[1]->getXYZ());
glVertexAttrib3fvNV(3, nodes[2]->getXYZ());
glVertexAttrib3fvNV(4, nodes[3]->getXYZ());

// Load up color for the vertices
glVertexAttrib4fvNV(5, colorvectors[0]);
glVertexAttrib4fvNV(6, colorvectors[1]);
glVertexAttrib4fvNV(7, colorvectors[2]);
glVertexAttrib4fvNV(8, colorvectors[3]);

// Writing to v[0] here invokes the vertex program.
glBegin(GL_TRIANGLE_FAN);
        glVertexAttrib3sNV(0, 0, 1, 0);
        glVertexAttrib3sNV(0, 1, 0, 1);
        glVertexAttrib3sNV(0, 2, 0, 1);
        glVertexAttrib3sNV(0, 3, 0, 1);
        glVertexAttrib3sNV(0, 4, 0, 1);
        glVertexAttrib3sNV(0, 1, 0, 1);
glEnd();
```

- These calls could easily be wrapped up into a glTetraExt () call.



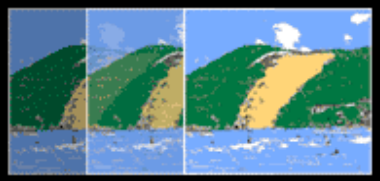
Remarks

- Wylie et al's technique can be easily extended to other computations, e.g., isosurface or isoline generation (this was a homework exercise in my graphics class last Spring)
- For isosurfaces, one can send two triangles (four vertices in a strip), since for a tetrahedral cell, the isosurface going through it has at most two triangles (see Comba's talk later today)



More Remarks

- Cell sorting is complicated, and error prone! (In our opinion, SXMPVO is probably easier to implement, and most robust technique)
- Cell projection is more stable, but still fairly complicated.
- Code available from http://www.cs.utah.edu/~csilva



SIBGRAPI 2005

Acknowledgements

- DOE VIEWS, DOE MICS, SNL, and LLNL, NSF, ARO, NIH, IBM, U of Utah.

- Joao Comba, Ricardo Farias, Brian Wylie, and others for slides and help with presentation