Direct Rendering of Feature-based Skinning Deformations

PhD Thesis

Andreas-Alexandros Vasilakis

abasilak@cs.uoi.gr

Department of Computer Science & Engineering,
University of Ioannina, Greece

January 2014

Outline

Outline

Problem statement

Direct rendering skinned approximations of self-crossing deformable objects

1. [skeletal animation] (mainly driven by segmentation)
2. [interactive rasterization]
   - multi-fragment rendering
   - self-intersection surfaces

Problem statement

Direct rendering skinned approximations of self-crossing deformable objects

1. [skeletal animation] (mainly driven by segmentation)
2. [interactive rasterization]
   - multi-fragment rendering
   - self-intersection surfaces

Wide range of applications

- animation compression (skinning), skeleton extraction, deformation transfer, volume preservation.
- post-editing, lod selection, collision detection, game engines.
- multi-fragment rendering effects:
  - interior visualization (transparency, clipping, capping, volume rendering)
  - self-trimming and constructive solid geometry (CSG) operations
  - and more: shadows, hair rendering, global illumination, voxelization...

Graphics rendering pipeline

[Main function]: generate (*render*) a 2D image given a virtual camera, 3D objects, light sources, textures, etc...

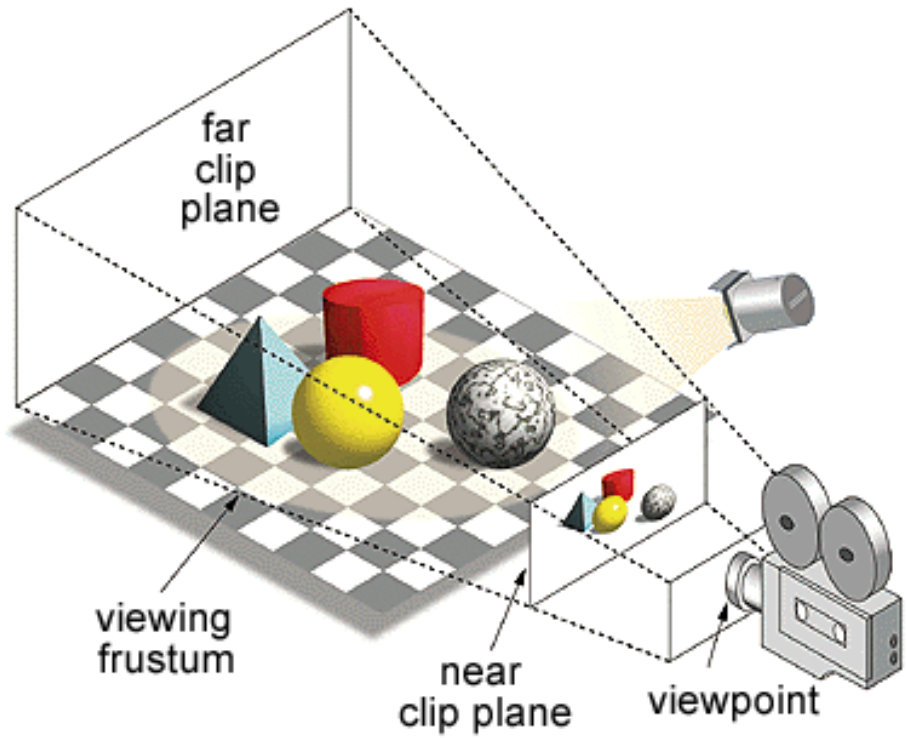


Figure: The rendering process of using this pipeline.

Graphics rendering pipeline

Division into three conceptual stages:

1. [Application]: fed next stage with the *rendering primitives*

Graphics rendering pipeline

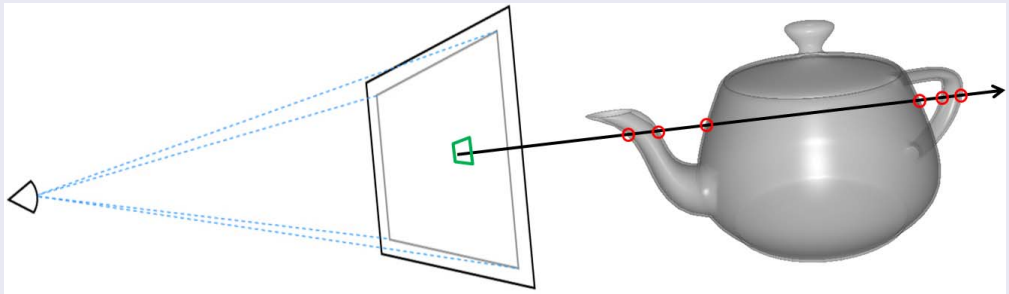
Division into three conceptual stages:

1. [Application]: fed next stage with the *rendering primitives*
2. [Geometry]: per-vertex & per-polygon operations

Graphics rendering pipeline

Division into three conceptual stages:

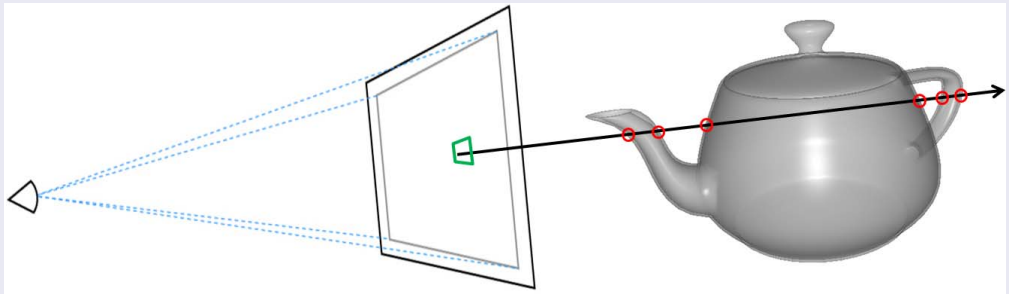
1. [Application]: fed next stage with the *rendering primitives*
2. [Geometry]: per-vertex & per-polygon operations
3. [Rasterizer]: generates the final image by computing the colors for the pixels covered by visible objects.



Graphics rendering pipeline

Division into three conceptual stages:

1. [Application]: fed next stage with the *rendering primitives*
2. [Geometry]: per-vertex & per-polygon operations
3. [Rasterizer]: generates the final image by computing the colors for the pixels covered by visible objects.

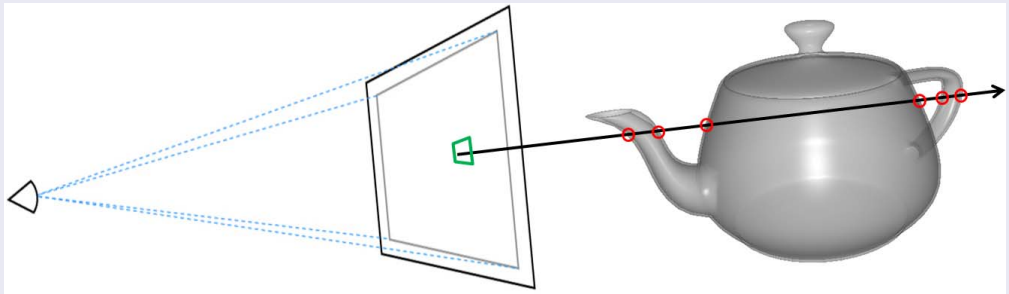


   1. [Scan-conversion]: fragment generation (vertex data interpolation)

Graphics rendering pipeline

Division into three conceptual stages:

1. [Application]: fed next stage with the *rendering primitives*
2. [Geometry]: per-vertex & per-polygon operations
3. [Rasterizer]: generates the final image by computing the colors for the pixels covered by visible objects.

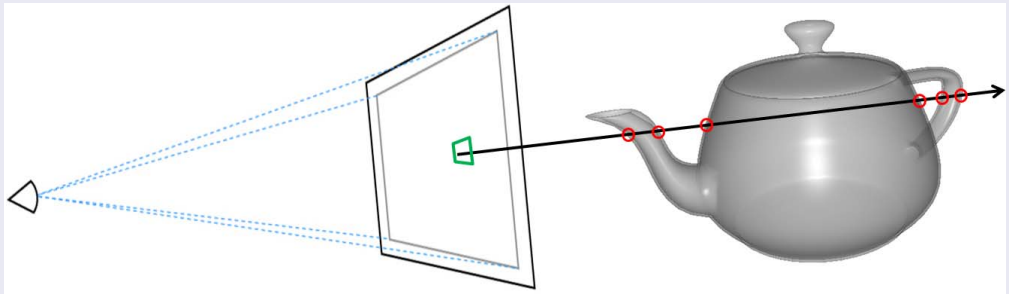


   1. [Scan-conversion]: fragment generation (vertex data interpolation)
   2. [Shading]: stores the color(s) and depth for each pixel in buffers

Graphics rendering pipeline

Division into three conceptual stages:

1. [Application]: fed next stage with the *rendering primitives*
2. [Geometry]: per-vertex & per-polygon operations
3. [Rasterizer]: generates the final image by computing the colors for the pixels covered by visible objects.



   1. [Scan-conversion]: fragment generation (vertex data interpolation)
   2. [Shading]: stores the color(s) and depth for each pixel in buffers
   3. [Merging]: resolves visibility via Z-testing and performs optionally various (*blending*, *stencil test*) operations.

Framework overview

System components

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

- {*geometry-stage*}

2. [Segmentation]: partitions MA into parts with similar motion features

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

- {*geometry-stage*}

2. [Segmentation]: partitions MA into parts with similar motion features
3. [Skinning]: transforms MA into a compact skeletal representation

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

- {*geometry-stage*}

2. [Segmentation]: partitions MA into parts with similar motion features
3. [Skinning]: transforms MA into a compact skeletal representation

- {*rasterizer-stage*}

4. [Fragment storage]: capture generated fragments in correct depth order

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

- {*geometry-stage*}

2. [Segmentation]: partitions MA into parts with similar motion features
3. [Skinning]: transforms MA into a compact skeletal representation

- {*rasterizer-stage*}

4. [Fragment storage]: capture generated fragments in correct depth order
5. [Self-trimming]: generates the trimmed surface (watertight solid)

Framework overview

System components

- {*application-stage*}

1. [Editing]: deforms an arbitrary static 3D object - mesh animation (MA)

- {*geometry-stage*}

2. [Segmentation]: partitions MA into parts with similar motion features
3. [Skinning]: transforms MA into a compact skeletal representation

- {*rasterizer-stage*}

4. [Fragment storage]: capture generated fragments in correct depth order
5. [Self-trimming]: generates the trimmed surface (watertight solid)
6. [Extra effects]: performs interior visualization or CSG operations

Mesh animation (MA)

A sequence of key-frames $\{p_t, t = 1, \ldots, k\}$, representing how a static 3D shape p_0 is evolving through time (may also self-intersect).

Mesh animation (MA)

A sequence of key-frames $\{p_t, t = 1, \ldots, k\}$, representing how a static 3D shape p_0 is evolving through time (may also self-intersect).

Figure: Horse, skirt and self-crossing sphere mesh animations.

Mesh animation (MA)

A sequence of key-frames $\{p_t, t = 1, \ldots, k\}$, representing how a static 3D shape p_0 is evolving through time (may also self-intersect).

Production

1. [automatic]: scanning machinery, multiple video cameras
2. [human intervention]: editing tools (skeleton, free-form)

Mesh animation (MA)

A sequence of key-frames $\{p_t, t = 1, \ldots, k\}$, representing how a static 3D shape p_0 is evolving through time (may also self-intersect).

Production

1. [automatic]: scanning machinery, multiple video cameras
2. [human intervention]: editing tools (skeleton, free-form)

Categories

1. time-varying versus [temporal-coherent]
2. [off-line] vs [real-time] (streamed or dynamically generated)

Mesh animation (MA)

A sequence of key-frames $\{p_t, t = 1, \ldots, k\}$, representing how a static 3D shape p_0 is evolving through time (may also self-intersect).

Production

1. [automatic]: scanning machinery, multiple video cameras
2. [human intervention]: editing tools (skeleton, free-form)

Categories

1. time-varying versus [temporal-coherent]
2. [off-line] vs [real-time] (streamed or dynamically generated)

Types

1. [rigid] (human, quadrupeds)
2. [highly-deformable] (cloth, facial)

} [hybrid] animation

Segmenting animated meshes

[Main goal]: partitions the animated mesh into clusters $\{C^0, \ldots, C^l\}$ with similar motion features

Segmenting animated meshes

[Main goal]: partitions the animated mesh into clusters $\{C^0, \ldots, C^l\}$ with similar motion features

Figure: Segmenting a horse mesh animation into 42 parts.

Segmenting animated meshes

[Main goal]: partitions the animated mesh into clusters $\{C^0, \ldots, C^l\}$ with similar motion features

Global segmentation methods

1. [clustering technique]: k-means, mean shift, hierarchical, spectral,...
2. [feature space]: euclidean, geodesic & angular distances, ...

Segmenting animated meshes

[Main goal]: partitions the animated mesh into clusters $\{C^0, \ldots, C^l\}$ with similar motion features

Global segmentation methods

1. [clustering technique]: k-means, mean shift, hierarchical, spectral,...
2. [feature space]: euclidean, geodesic & angular distances, ...

Limitations

- focus on detecting segments with mostly *rigid* behavior
- fail when feature space is *flat* or *anisotropic*
- work only with *off-line* animations (not-editable)
- cannot offer smooth multi-resolution & variable[1] segmentations

[1] [Arcila et al, GM'13]

Skeletal animation

Use a simpler structure (skeleton) to animate mesh instead of editing each primitive independently.

Skeletal animation

Use a simpler structure (skeleton) to animate mesh instead of editing each primitive independently.

Representation

(a) reference 3D surface mesh

(b) endoskeleton (joints, bones) attached to the skin (*rigging*)

(c) anatomy layer (musculature)

(d) direct manipulation, inverse kinematics or motion data

Mesh Skinning

- efficient (GPU-accelerated), widely used from game engines

Mesh Skinning

- efficient (GPU-accelerated), widely used from game engines

Skin deformation algorithm

- reference 3D surface mesh (rest-pose)
- a list of joint transformations (no actual skeleton needed)
- two lists of vertex indices and influencing weights

Mesh Skinning

- efficient (GPU-accelerated), widely used from game engines

Skin deformation algorithm

- reference 3D surface mesh (rest-pose)
- a list of joint transformations (no actual skeleton needed)
- two lists of vertex indices and influencing weights

Predominant skinning techniques

- Linear blend skinning (LBS[1]): fastest, elongations & interpenetrations
- Dual quaternion skinning (DQS[2]): rigid motion, less storage, extra cost

[1] [Magnenat-Thalmann et al, GI'88], [2] [Kavan et al, TOG'08]

Mesh Skinning

- efficient (GPU-accelerated), widely used from game engines

Skin deformation algorithm

- reference 3D surface mesh (rest-pose)
- a list of joint transformations (no actual skeleton needed)
- two lists of vertex indices and influencing weights

Predominant skinning techniques

- Linear blend skinning (LBS[1]): fastest, elongations & interpenetrations
- Dual quaternion skinning (DQS[2]): rigid motion, less storage, extra cost

Approximate MA with skinning

- [Goal]: animation compression (data reduction)
- [Applications]: GPU, collision detection, lod selection, pose editing

Self-intersected Mesh Animations

Geometry processing pipeline:

- [deformation], smoothing, subdivision and decimation
- [skinning approximation]

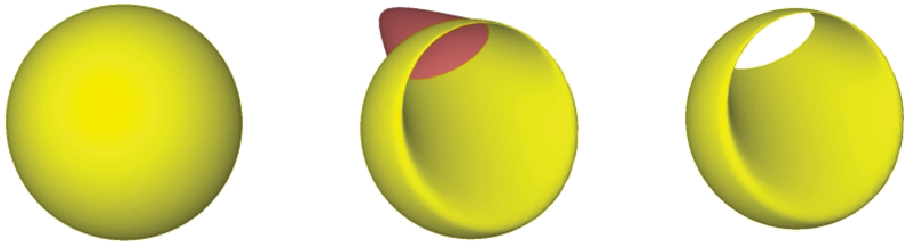
Self-intersected Mesh Animations

Geometry processing pipeline:

- [deformation], smoothing, subdivision and decimation
- [skinning approximation]

Problems

1. Given a *self-crossing surface* (SCS) $\rightarrow$ [trim]
2. Given a SCS derived by deforming an initial non-SCS surface $\rightarrow$ [trim]



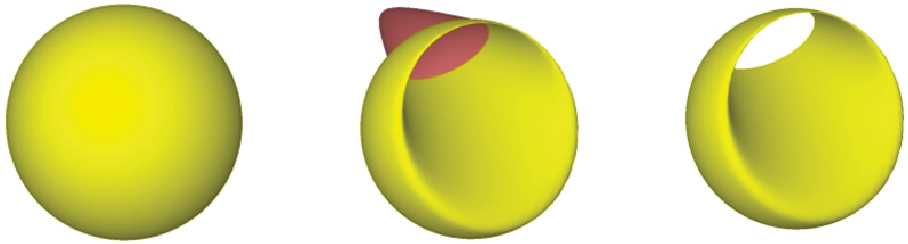
Self-intersected Mesh Animations

Geometry processing pipeline:

- [deformation], smoothing, subdivision and decimation
- [skinning approximation]

Problems

1. Given a *self-crossing surface* (SCS) $\rightarrow$ [trim]
2. Given a SCS derived by deforming an initial non-SCS surface $\rightarrow$ [trim]



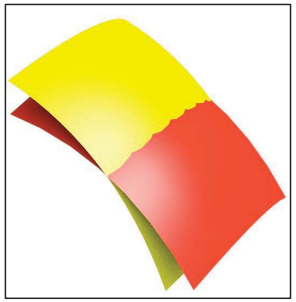
[geometry space] versus [image space]

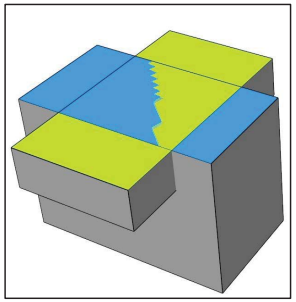
Multi-fragment rendering

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces



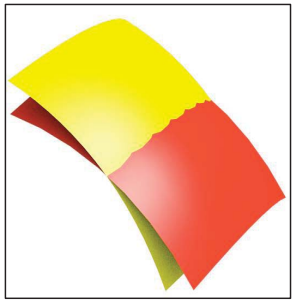


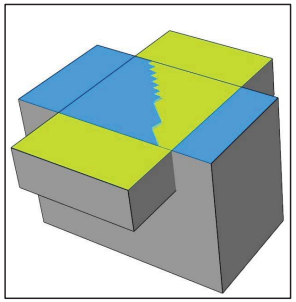
Multi-fragment rendering

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces





Storing multiple fragments

- multi-pass depth peeling (F2B[1], Dual[2], Bucket[3])

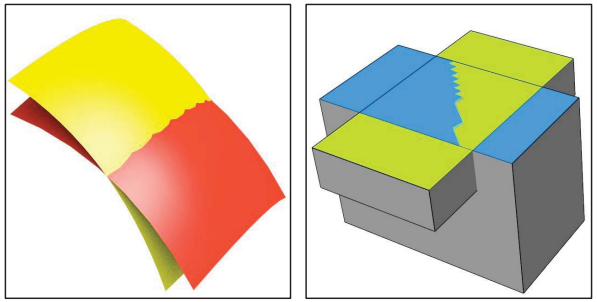
[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Multi-fragment rendering

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces



Storing multiple fragments

- multi-pass depth peeling (F2B[1], Dual[2], Bucket[3]) [z-fighting flaws]

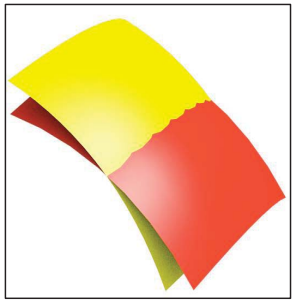
[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

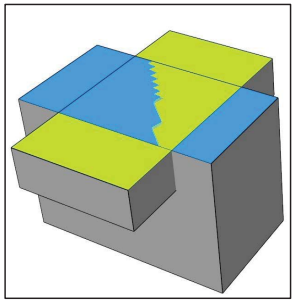
Multi-fragment rendering

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces





Storing multiple fragments

- multi-pass depth peeling (F2B[1], Dual[2], Bucket[3]) [z-fighting flaws]
- hardware buffers (A-buffer[4], k-buffer[5])

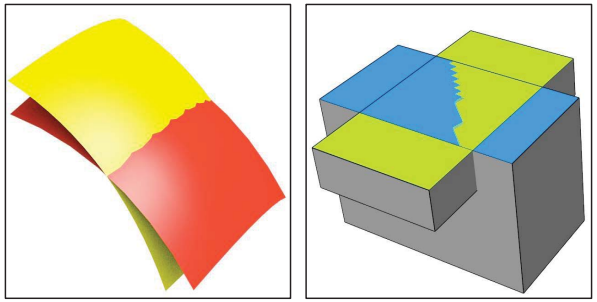
[4] [Carpenter, SIGGRAPH'84], [5] [Bavoil et al, I3D'07]

Multi-fragment rendering

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces



Storing multiple fragments

- multi-pass depth peeling (F2B[1], Dual[2], Bucket[3]) [z-fighting flaws]
- hardware buffers (A-buffer[4], k-buffer[5]) [fragment congestion & overflow]

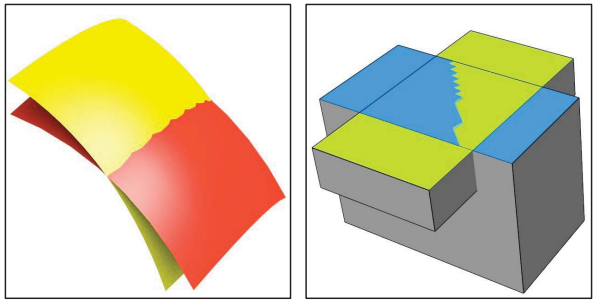
[4] [Carpenter, SIGGRAPH'84], [5] [Bavoil et al, I3D'07]

Multi-fragment rendering

Z-fighting phenomenon

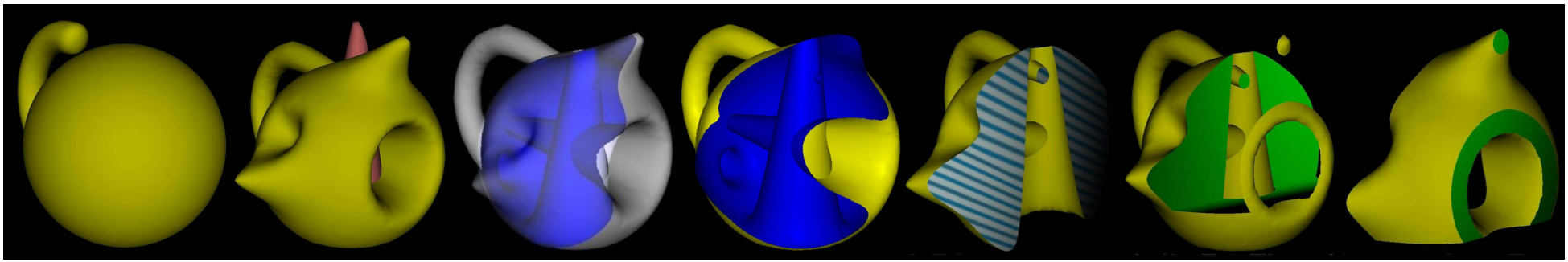
more

- two or more primitives have same depth
- intersecting or overlapping surfaces



Storing multiple fragments

- multi-pass depth peeling (F2B[1], Dual[2], Bucket[3]) [z-fighting flaws]
- hardware buffers (A-buffer[4], k-buffer[5]) [fragment congestion & overflow]



Outline

Revising segmentation of MAs

Segmentation Problem

go back

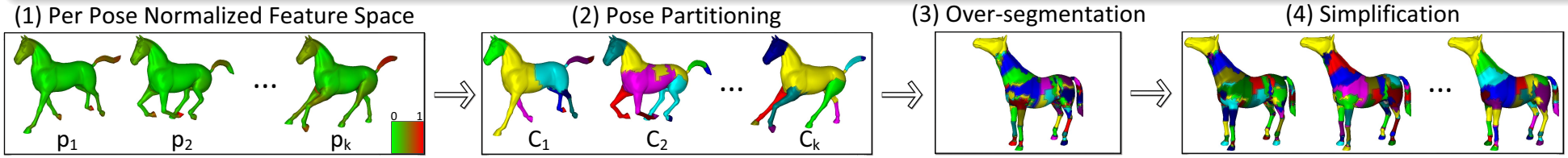
- handles all types and categories of MAs
- supports multi-resolution and variable segmentations
- efficiency and quality

Revising segmentation of MAs

Segmentation Problem

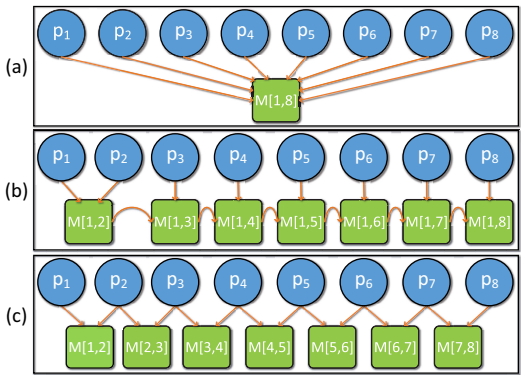
go back

- handles all types and categories of MAs
- supports multi-resolution and variable segmentations
- efficiency and quality



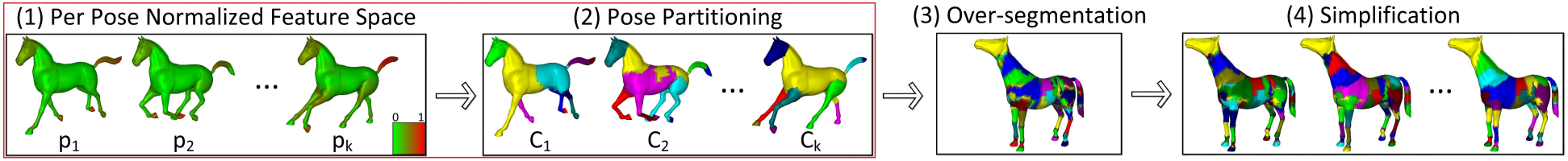
Our contribution†

1. computes per-pose partitioning (feature)
2. computes over-segmentation (OS)
3. performs progressive simplification

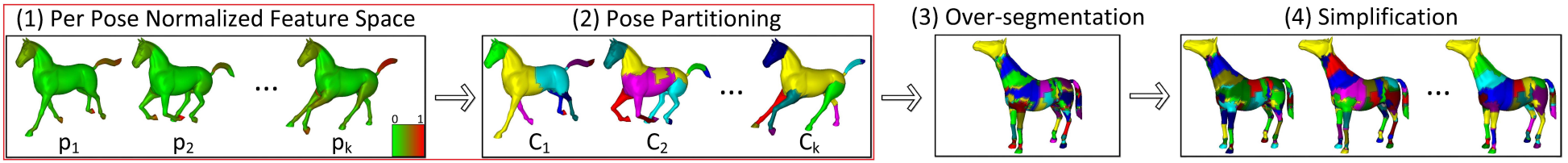


† **A. Vasilakis** & I. Fudos, *Pose Partitioning for Multi-resolution Segmentation of Arbitrary Mesh Animations*, Computer Graphics Forum (Proceedings of Eurographics 2014), vol. 33(2):?-?

(1,2) Per-pose segmentation



(1,2) Per-pose segmentation

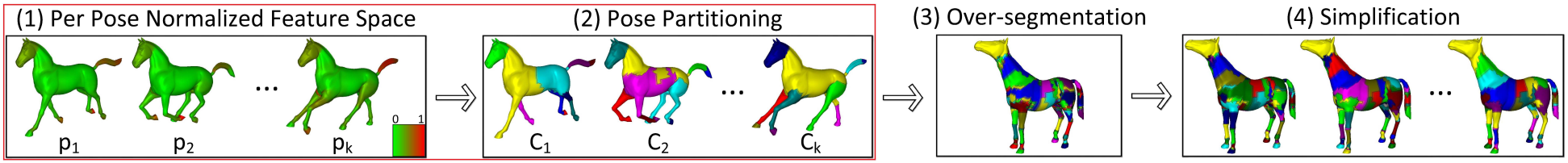


(1) Feature space computation

- a deformation measure from reference (p_0 or p_{t-1}) to target pose (p_t)
- deformation gradient $\rightarrow$ [rotation angles][1]

[1] [Sumner et al., TOG'04] , [2] [Gunther et al., CGF'06]

(1,2) Per-pose segmentation



(1) Feature space computation

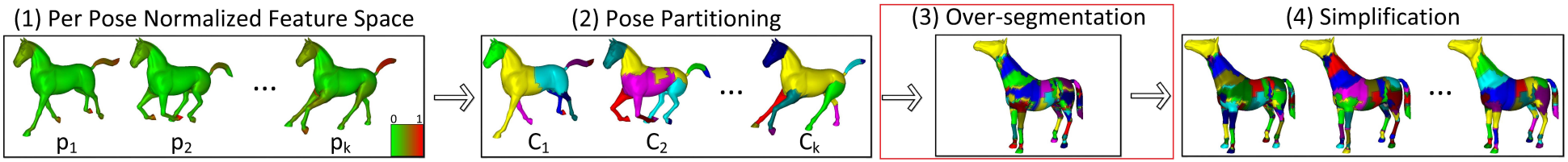
- a deformation measure from reference (p_0 or p_{t-1}) to target pose (p_t)
- deformation gradient $\rightarrow$ [rotation angles][1]

(2) Partitioning computation

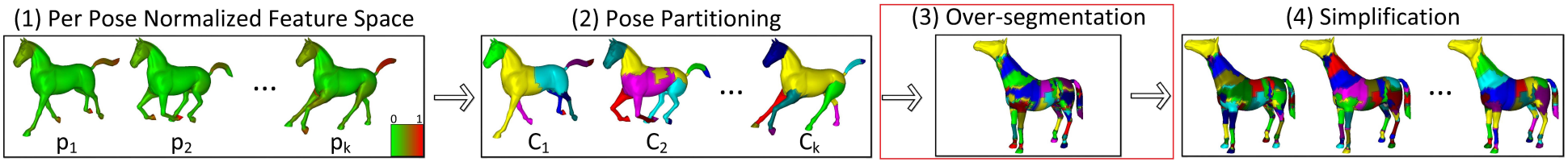
- numerous clustering methods (in [parallel])
- [top-down hierarchical clustering][2]
- [output]: $C_i = \{C_i^0, \ldots, C_i^{l_i}\}, \forall p_i \in [1, k]$

[1] [Sumner et al., TOG'04] , [2] [Gunther et al., CGF'06]

(3) Pose partitioning-aware OS



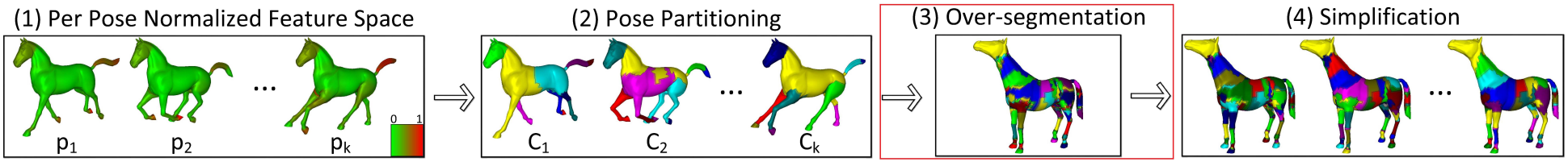
(3) Pose partitioning-aware OS



Clustering Animation Vector (CAV)

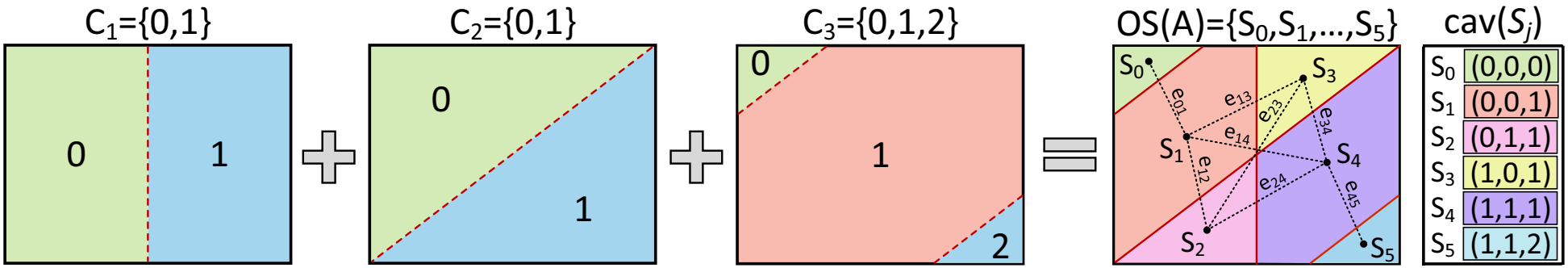
- $cav(v) = \{m_1, m_2, \ldots, m_k\},\ m_j \in 1, \ldots, |C_j|,\ \forall v \in V$
- [Pruned-BFS]: detect connected vertices with same CAV - $O(|V|)$ alg
- [output]: $OS(A) = \{S_0, S_1, \ldots, S_l\}$

(3) Pose partitioning-aware OS

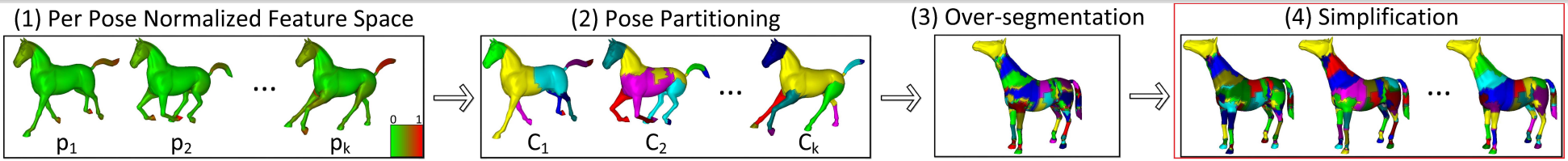


Clustering Animation Vector (CAV)

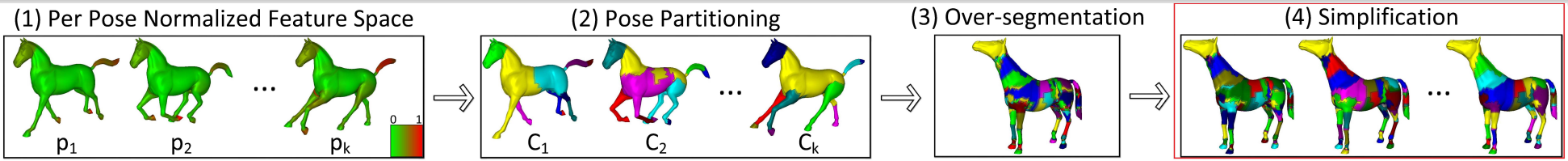
- $cav(v) = \{m_1, m_2, \ldots, m_k\},\ m_j \in 1, \ldots, |C_j|,\ \forall v \in V$
- [Pruned-BFS]: detect connected vertices with same CAV - $O(|V|)$ alg
- [output]: $OS(A) = \{S_0, S_1, \ldots, S_l\}$



(4) Progressive decimation of OS



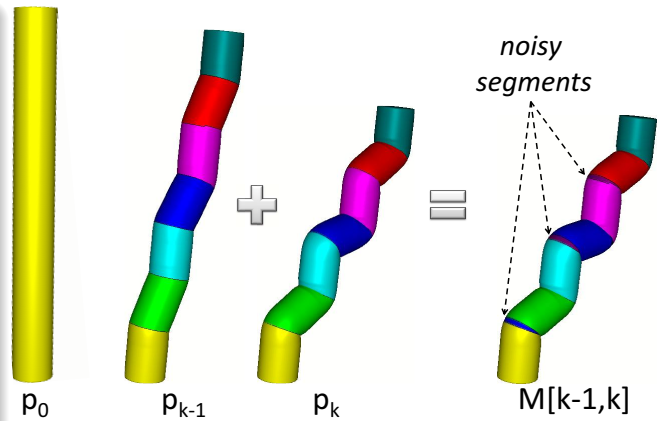
(4) Progressive decimation of OS



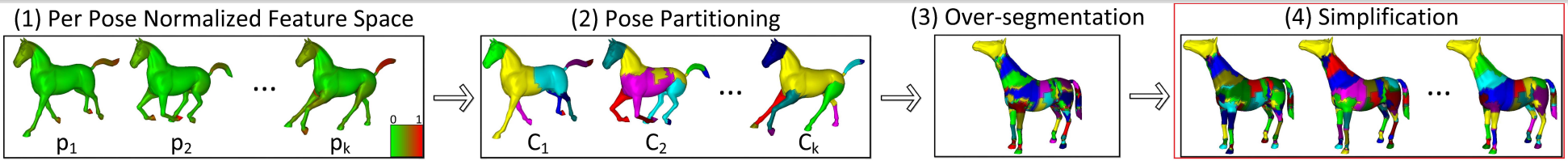
Pose-to-pose cleaning $R_i(h),\ h \in [0,1]$

alg

- $p_k \longmapsto, \ldots, \longmapsto p_1$
- reduction rule for (S_A, S_B):
  - $a(S_A) \leq h \cdot a(C_i^{cav(S_A)[i]})$
  - $a(S_A) \leq h \cdot a(segment(S_A, i-1))$
  - $S_B \in N(S_A) \wedge cav(S_A)[1, i-1] = cav(S_B)[1, i-1] \wedge cav(S_A)[i] \neq cav(S_B)[i]$
  - $a(S_B) > a(S_A)$



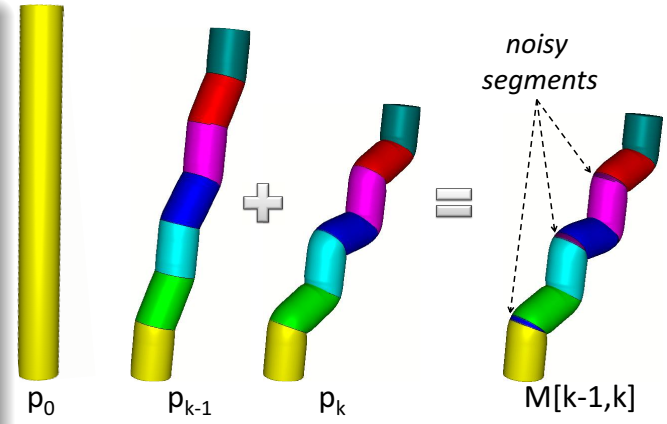
(4) Progressive decimation of OS

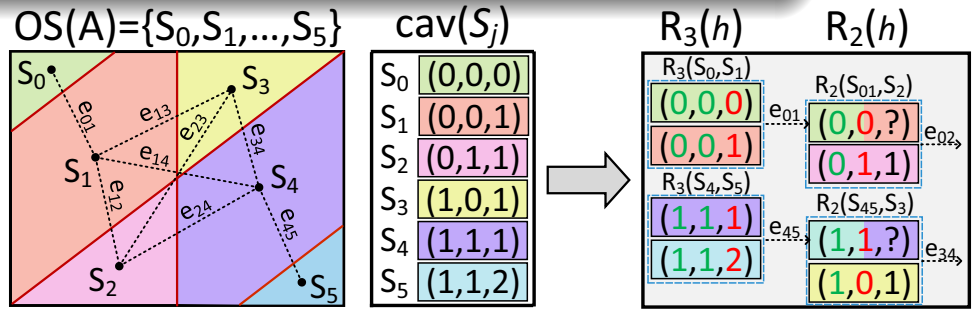


Pose-to-pose cleaning $R_i(h),\ h \in [0,1]$

alg

- $p_k \longmapsto, \ldots, \longmapsto p_1$
- reduction rule for (S_A, S_B):
  - $a(S_A) \leq h \cdot a(C_i^{cav(S_A)[i]})$
  - $a(S_A) \leq h \cdot a(segment(S_A, i-1))$
  - $S_B \in N(S_A) \wedge cav(S_A)[1, i-1] = cav(S_B)[1, i-1] \wedge cav(S_A)[i] \neq cav(S_B)[i]$
  - $a(S_B) > a(S_A)$

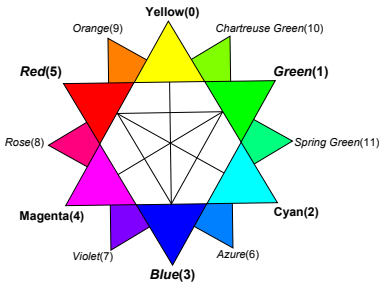




Applications (1)

Smooth Visualization of Cluster Transitions

- minimize "close" colors assigned to neighbors
- propagate colors between consecutive frames



Applications (1)

Smooth Visualization of Cluster Transitions

- minimize "close" colors assigned to neighbors
- propagate colors between consecutive frames

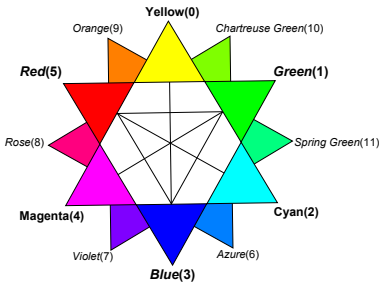


Figure: (left) initial painting and (right) color propagation.

Applications (2)

Multi-resolution segmentation

- High $\longmapsto$ low: adjusting parameter $h : 0 \longmapsto 1$
- Bounded by resolutions of $[C_1, OS]$ - [High similarity] ?
- Efficient: do not depend on mesh size

Applications (2)

Multi-resolution segmentation

- High $\longmapsto$ low: adjusting parameter $h : 0 \longmapsto 1$
- Bounded by resolutions of $[C_1, OS]$ - [High similarity] ?
- Efficient: do not depend on mesh size

Applications (3)

Variable (temporal consistent) segmentation

back

(a) remain faithful to past-data
(b) alter to future data

} [pair merging] $(C_{t-1}, C_t) \longmapsto$ real-time

Applications (3)

Variable (temporal consistent) segmentation

back

(a) remain faithful to past-data
(b) alter to future data

} [pair merging] $(C_{t-1}, C_t) \longmapsto$ real-time

Figure: (left) color propagation and (right) variable segmentation.

Applications (4)

Real-time segmentation

back

- streamed/dynamic MA
- out-of-core algorithm

} [merge] $(OS(0, t-1), C_t) \longmapsto$ cost of C_t

Applications (4)

Real-time segmentation

back

- streamed/dynamic MA
- out-of-core algorithm

} [merge] $(OS(0, t-1), C_t) \longmapsto$ cost of C_t

Figure: Real-time segmentation (left) without and (right) with p2p-cleaning

Applications (5)

back

Modifying MA

- add new poses $\longmapsto$ real-time segmentation

example

Applications (5)

back

Modifying MA

- add new poses $\longmapsto$ real-time segmentation example
- edit arbitrary pose p_t: [replace] $C_t \longmapsto C'_t$ & [recompute OS] example

Applications (5)

back

Modifying MA

- add new poses $\longmapsto$ real-time segmentation example
- edit arbitrary pose p_t: [replace] $C_t \longmapsto C'_t$ & [recompute OS] example

Segmentation transfer

- MA 1
- MA 2
- ...
- MA k

$\}$ [merge MAs] $\longmapsto$ [merge segmentations] example

Experimental Study (1)

Performance Analysis (1)

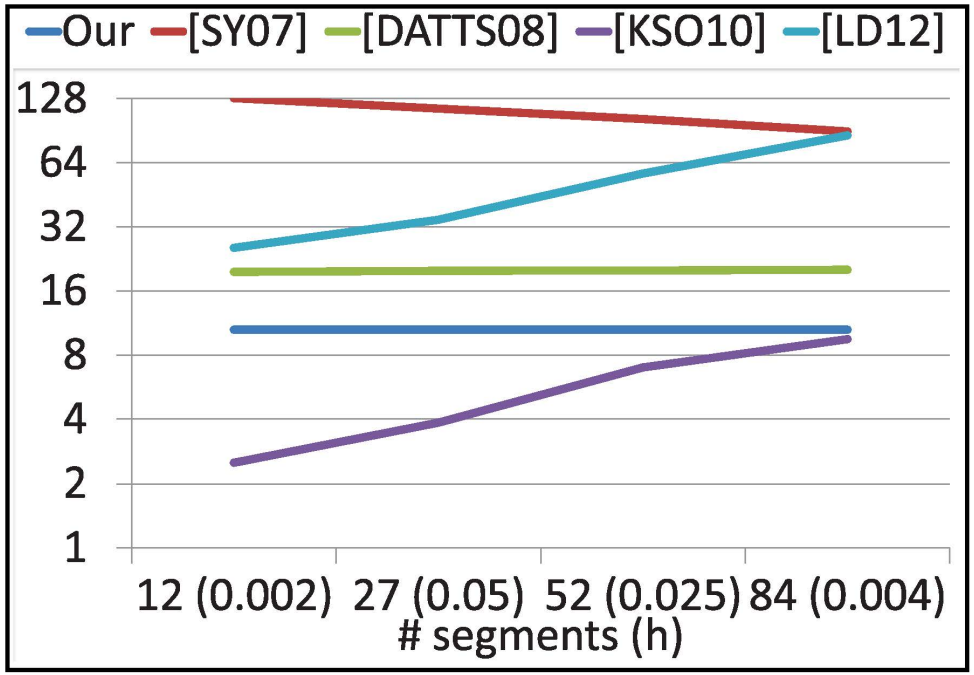
- linear behavior on the geometry size
- linear scale when increasing per-pose partitionings
- pose decompositions consumes most of the time
- interactive performance $\longmapsto$ [GPU-accelerated clustering]

				Per pose						Mesh Animation			
					Feature	Clustering		Variable Segmentation		Over-segmentation	p2p-Cleaning	Total	
Mesh Animation	Vertices	Faces	Poses	Clusters	Compute	Compute	Propagate Colors	Compute	Propagate Colors	Compute (segments)	Compute (segments)	real-time	off-line
Hand	7929	15855	22	12	0.116	1.603	0.0014	0.0046	0.00084	0.215 (379)	0.03 (24)	1.72	37.95
Elephant Gallop	42321	84638	24	5	0.483	3.327	0.0053	0.012	0.0041	0.735 (743)	0.07 (18)	3.844	92.25
				10	0.483	6.463	0.0053	0.012	0.0041	1.475 (1729)	0.09 (18)	7.281	174.7
Flowing Cloth	25921	51200	19	[2,5]	0.295	1.873	0.0028	0.0074	0.0025	0.698 (400)	0.02 (24)	2.205	41.91
Samba	9971	19938	24	5	0.158	0.734	0.0013	0.024	0.0009	0.78 (1016)	0.07 (12)	0.927	22.26

Experimental Study (1)

Performance Analysis (2)

- tiny cost when moving from high to low resolutions
- only multi-source region growing method[1] is faster



[1] [Kavan et al., CGF'10]

Experimental Study (2)

Quality Analysis (1)

- E_{RMS}: skinning approximation error
- [rigid] $\longmapsto$ [volumetric parts]
- [highly-deformable] $\longmapsto$ [surface parts]

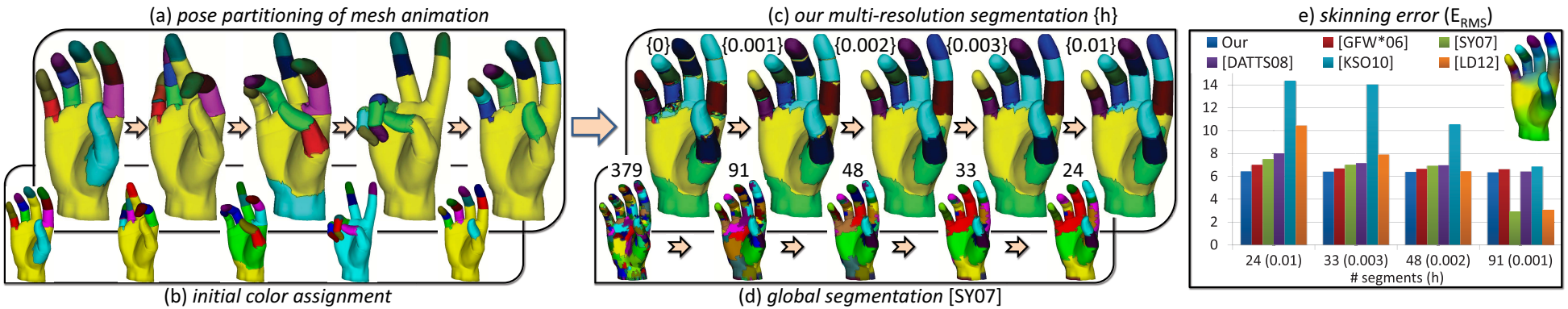
} [hybrid]

Experimental Study (2)

Quality Analysis (1)

- E_{RMS}: skinning approximation error
- [rigid] $\longmapsto$ [volumetric parts]
- [highly-deformable] $\longmapsto$ [surface parts]

} [hybrid]



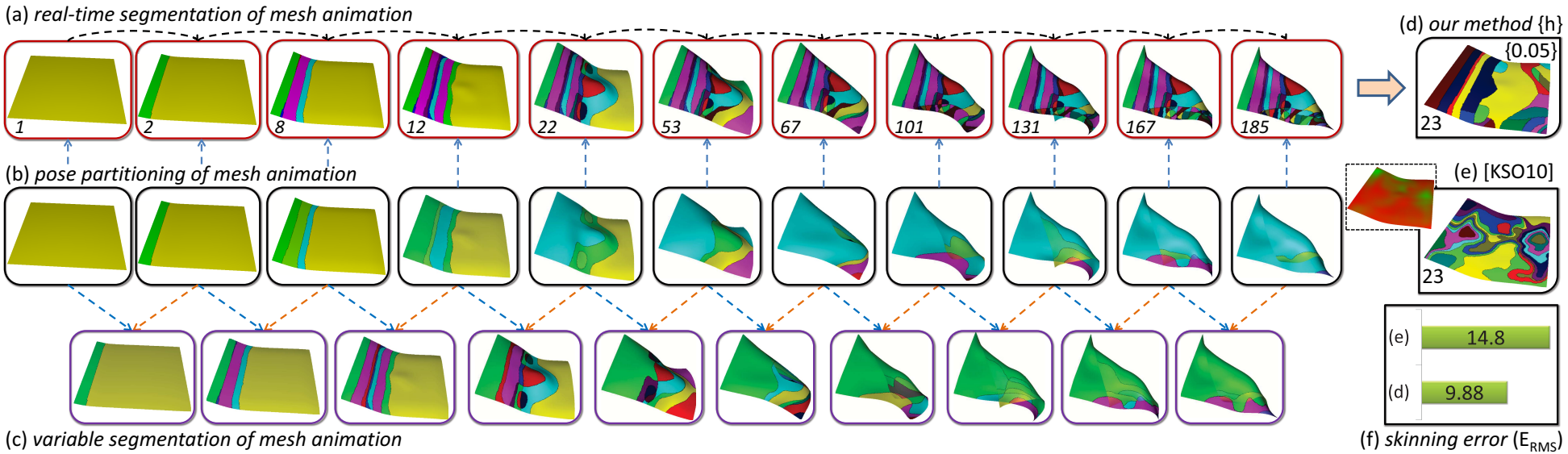
[1] [Kavan et al., CGF'10], [2] [Schaeffer et al., GP'07]

Experimental Study (2)

Quality Analysis (1)

- E_{RMS}: skinning approximation error
- [rigid] $\longmapsto$ [volumetric parts]
- [highly-deformable] $\longmapsto$ [surface parts]

} [hybrid]



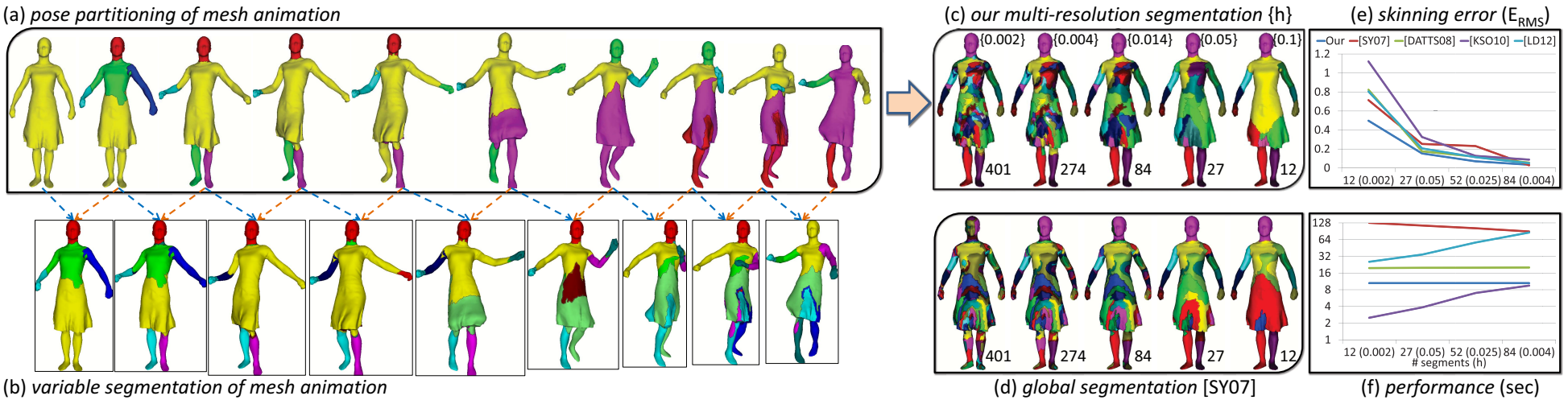
[1] [Kavan et al., CGF'10], [2] [Schaeffer et al., GP'07]

Experimental Study (2)

Quality Analysis (1)

- E_{RMS}: skinning approximation error
- [rigid] $\longmapsto$ [volumetric parts]
- [highly-deformable] $\longmapsto$ [surface parts]

} [hybrid]



[1] [Kavan et al., CGF'10], [2] [Schaeffer et al., GP'07]

Discussion

Quality

- [our method] (merge & simplify) VS [global segmentation]

Discussion

Quality

- [our method] (merge & simplify) VS [global segmentation]
- depends on quality/number of individual pose partitionings:
  1. higher-quality clustering
  2. boundary smoothing
  3. confidence measures

  } parameter tuning & extra computation

Discussion

Quality

- [our method] (merge & simplify) VS [global segmentation]
- depends on quality/number of individual pose partitionings:
  1. higher-quality clustering
  2. boundary smoothing
  3. confidence measures

  } parameter tuning & extra computation

Performance

- applications run at real-time speed

Discussion

Quality

- [our method] (merge & simplify) VS [global segmentation]
- depends on quality/number of individual pose partitionings:
  1. higher-quality clustering
  2. boundary smoothing
  3. confidence measures

  } parameter tuning & extra computation

Performance

- applications run at real-time speed
- reduce computation cost of pose partitioning:
  1. cluster poses with high partitioning similarity

Discussion

Quality

- [our method] (merge & simplify) VS [global segmentation]
- depends on quality/number of individual pose partitionings:
  1. higher-quality clustering
  2. boundary smoothing
  3. confidence measures

  } parameter tuning & extra computation

Performance

- applications run at real-time speed
- reduce computation cost of pose partitioning:
  1. cluster poses with high partitioning similarity
  2. GPU-accelerated clustering

Outline

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$
- [additional applications]: GPU reproduction, lod selection, etc

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$
- [additional applications]: GPU reproduction, lod selection, etc

Our contribution - *p2p-skinning*†

1. [Idea:] $p^s_{t-1} \longmapsto p^a_t$ instead of $p_0 \longmapsto p^a_t$ - (reproduction scheme)

† **A. Vasilakis** & G. Antonopoulos & I. Fudos, *Pose-to-Pose Skinning of Animated Meshes*, Symposium on Computer Animation (Posters), Vancouver, Canada, August 5-7, 2011.

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$
- [additional applications]: GPU reproduction, lod selection, etc

Our contribution - *p2p-skinning*†

1. [Idea:] $p^s_{t-1} \longmapsto p^a_t$ instead of $p_0 \longmapsto p^a_t$ - (reproduction scheme)
2. supports both [linear] (LBS) and [non-linear] (DQS) skinning

† **A. Vasilakis** & G. Antonopoulos & I. Fudos, *Pose-to-Pose Skinning of Animated Meshes*, Symposium on Computer Animation (Posters), Vancouver, Canada, August 5-7, 2011.

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$
- [additional applications]: GPU reproduction, lod selection, etc

Our contribution - *p2p-skinning*†

1. [Idea:] $p^s_{t-1} \longmapsto p^a_t$ instead of $p_0 \longmapsto p^a_t$ - (reproduction scheme)
2. supports both [linear] (LBS) and [non-linear] (DQS) skinning
3. enables prior applications + [arbitrary pose editing]

† **A. Vasilakis** & G. Antonopoulos & I. Fudos, *Pose-to-Pose Skinning of Animated Meshes*, Symposium on Computer Animation (Posters), Vancouver, Canada, August 5-7, 2011.

Pose-to-pose skinning of MAs

Skinning MAs

go back

- [key-frame compression]: $3n \cdot k \longmapsto (3n + 4n) + y \cdot k,\ y = \{8, 12\}$
- [additional applications]: GPU reproduction, lod selection, etc

Our contribution - *p2p-skinning*†

1. [Idea:] $p^s_{t-1} \longmapsto p^a_t$ instead of $p_0 \longmapsto p^a_t$ - (reproduction scheme)
2. supports both [linear] (LBS) and [non-linear] (DQS) skinning
3. enables prior applications + [arbitrary pose editing]
4. offers [refinements] to improve skinning quality

† **A. Vasilakis** & G. Antonopoulos & I. Fudos, *Pose-to-Pose Skinning of Animated Meshes*, Symposium on Computer Animation (Posters), Vancouver, Canada, August 5-7, 2011.

Approximating MAs with Skinning

Notation & Equations

- [rest-pose]: $\{v_j^0\}$
- [weights]: $[w_{1,j}, \ldots, w_{B,j}], w_{i,j} > 0, \sum_{i=1}^{B} w_{i,j} = 1$

$\Big\}$ known

- [transformation matrices]: $[M_1^t, \ldots, M_B^t]$ - unknown

$$v_j^t = T_j^t \cdot v_j^0, \forall t \in [0, k], \text{ where } T_j^t = \sum_{b=1}^{B} w_{b,j} M_b^t \quad (1)$$

Approximating MAs with Skinning

Notation & Equations

- [rest-pose]: $\{v_j^0\}$
- [weights]: $[w_{1,j}, \ldots, w_{B,j}], w_{i,j} > 0, \sum_{i=1}^{B} w_{i,j} = 1$

} known

- [transformation matrices]: $[M_1^t, \ldots, M_B^t]$ - unknown

$$v_j^t = T_j^t \cdot v_j^0, \forall t \in [0, k], \text{ where } T_j^t = \sum_{b=1}^{B} w_{b,j} M_b^t \quad (1)$$

- [dual quaternions]: less storage, only rigid motion.

Approximating MAs with Skinning

Notation & Equations

- [rest-pose]: $\{v_j^0\}$
- [weights]: $[w_{1,j}, \ldots, w_{B,j}], w_{i,j} > 0, \sum_{i=1}^{B} w_{i,j} = 1$

} known

- [transformation matrices]: $[M_1^t, \ldots, M_B^t]$ - unknown

$$v_j^t = T_j^t \cdot v_j^0, \forall t \in [0, k], \text{ where } T_j^t = \sum_{b=1}^{B} w_{b,j} M_b^t \quad (1)$$

- [dual quaternions]: less storage, only rigid motion.

Fitting Formula

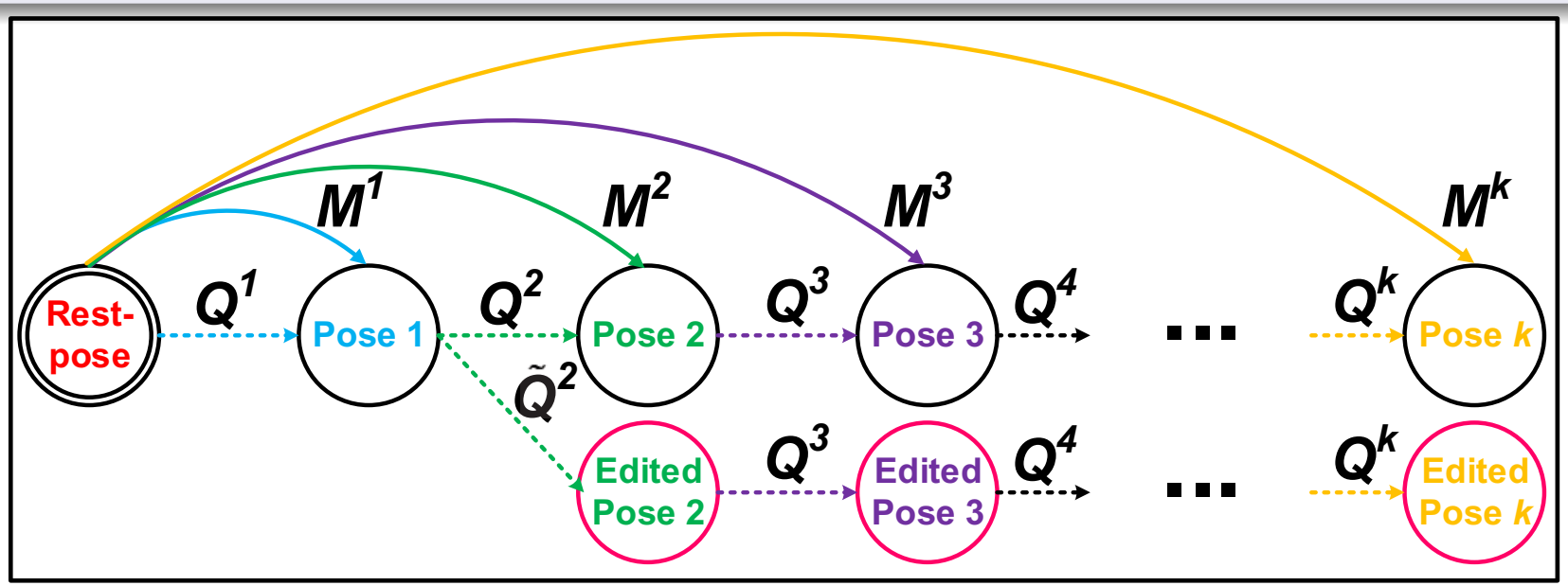
$$\min\{\sum_{j=0}^{n-1} \left\| \mathbf{v}_j^t - v_j^t \right\|^2\}, \text{ where } \mathbf{v}_j^t \in p_t$$

Pose-to-pose Fitting

Our solution

- small deformation variations will occur between sequential poses
- reformulate equation (1):

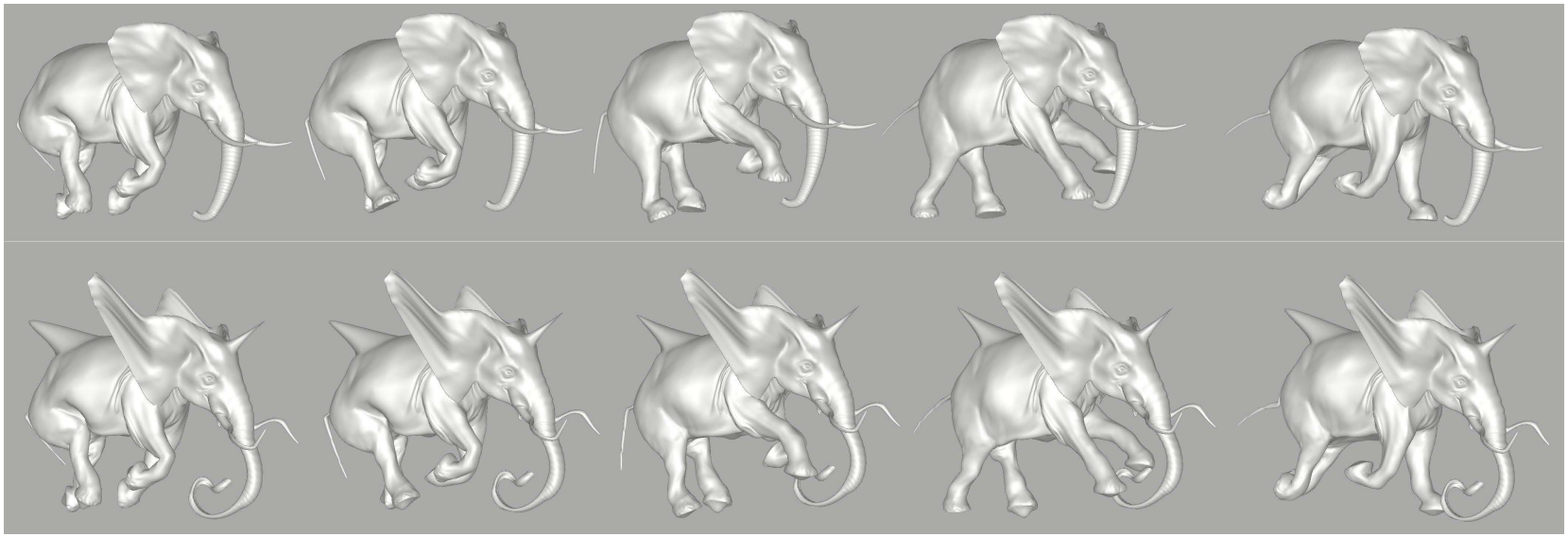
$$v_j^t = \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right) v_j^{t-1} \;,\; T_j^t = \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right) T_j^{t-1} \qquad (2)$$



Animation Editing

Rest-pose editing

- simply modify $\{v_j^0\}$

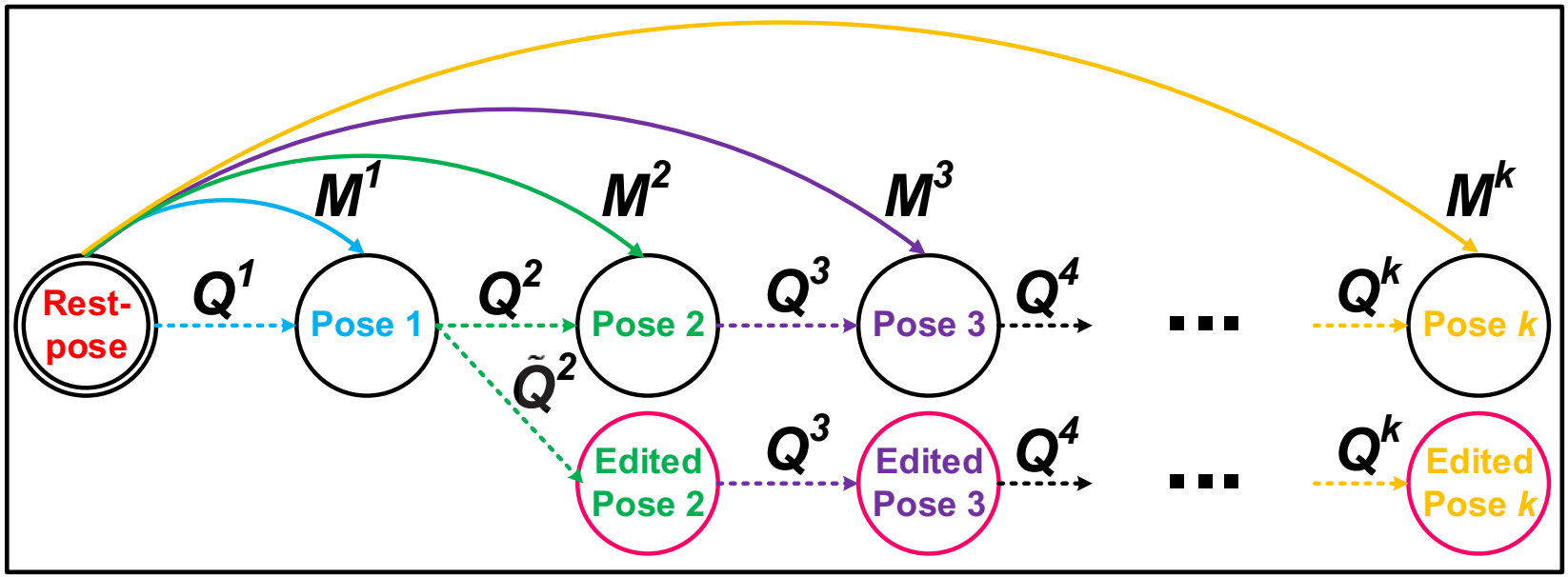


Animation Editing

Arbitrary pose editing

- $\{\widetilde{Q}_b^e\}$ matrices: $p_{e-1} \longmapsto p_e \Rightarrow$

$$T_j^t = T_j^{t,e-1} \left(\sum_{b=1}^{B} w_{b,j} \widetilde{Q}_b^e \right) T_i^{e-1,1}, \quad T_j^{t_0,t_1} = \prod_{t=t_0}^{t_1} \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right)$$

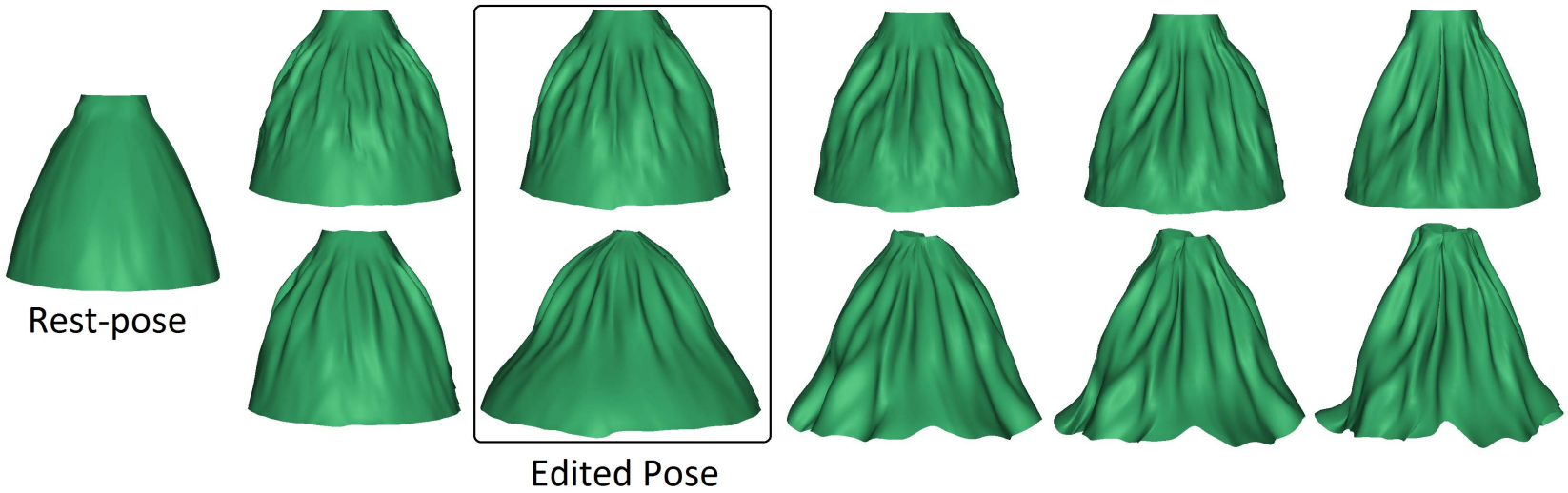


Animation Editing

Arbitrary pose editing

- $\{\widetilde{Q}_b^e\}$ matrices: $p_{e-1} \longmapsto p_e \Rightarrow$

$$T_j^t = T_j^{t,e-1} \left(\sum_{b=1}^{B} w_{b,j} \widetilde{Q}_b^e \right) T_i^{e-1,1}, \quad T_j^{t_0,t_1} = \prod_{t=t_0}^{t_1} \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right)$$

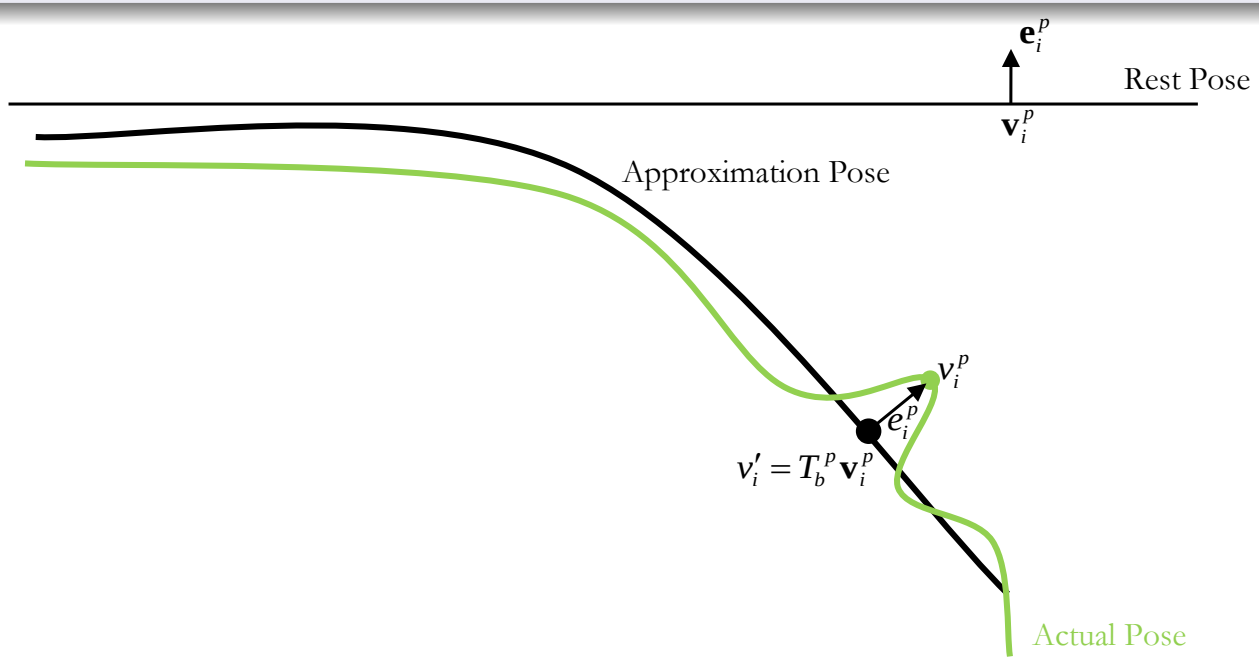


Rest-pose Vertex Corrections

Problem formulation

- [displacement field]: $e^V = [e_0^V, \ldots, e_{n-1}^V] \in \mathbb{R}^{4\times n}$:

$$\min\{\sum_{t=0}^{k-1} \left\| T^t(v_j^0 + e_j^V) - \mathbf{v}_j^t \right\|^2\}, \; j \in [0, \ldots, n) \tag{3}$$



Weight Corrections

Problem formulation

- [displacement field]: $e^W = [e_1^W, \ldots, e_B^W] \in \mathbb{R}^{1\times B}$

$$\min\{\sum_{t=0}^{k-1} \left\| \sum_{b=1}^{B} (w_{b,j} + e_{b,j}^W) Q_b^t \mathbf{v}_j^{t-1} - \mathbf{v}_j^t \right\|^2 \}, \; j \in [0, \ldots, n) \qquad (4)$$

Weight Corrections

Problem formulation

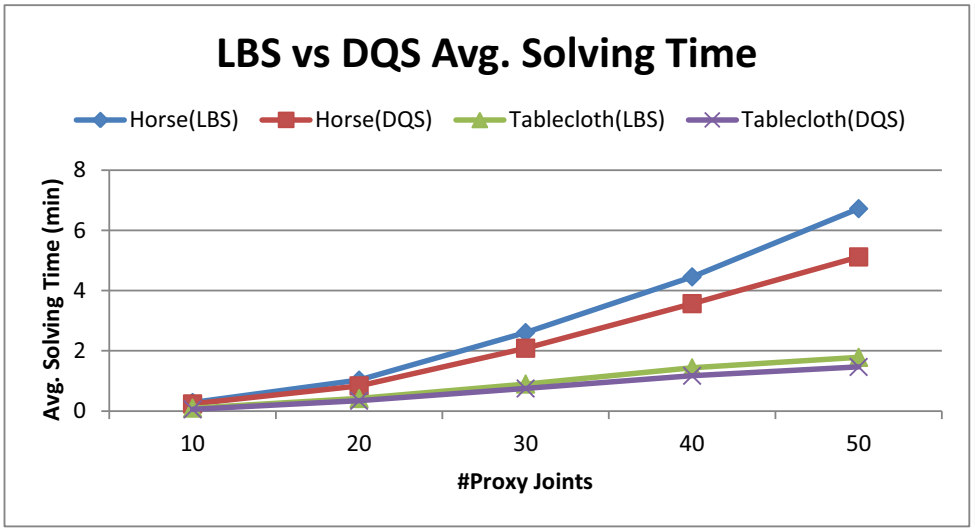
- [displacement field]: $e^W = [e_1^W, \ldots, e_B^W] \in \mathbb{R}^{1\times B}$

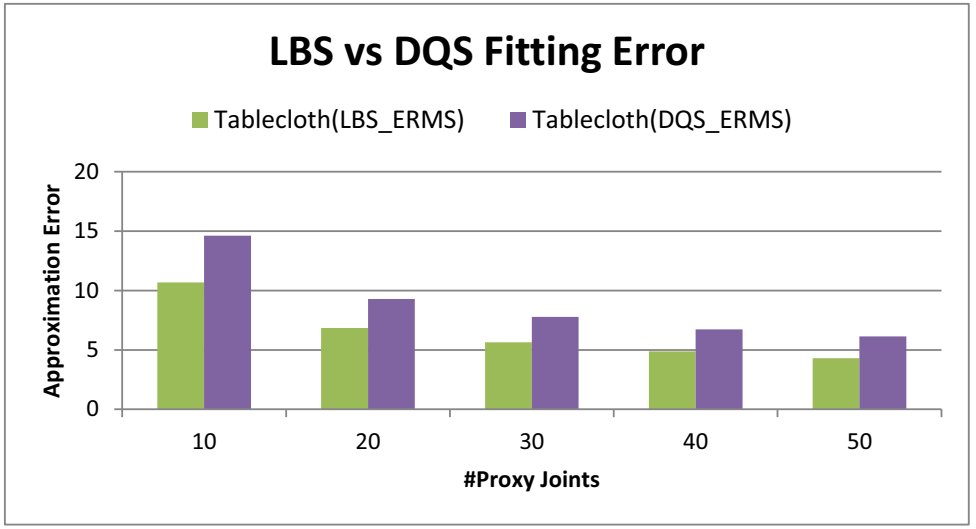
$$\min\{\sum_{t=0}^{k-1} \left\| \sum_{b=1}^{B} (w_{b,j} + e_{b,j}^W) Q_b^t \mathbf{v}_j^{t-1} - \mathbf{v}_j^t \right\|^2 \}, \; j \in [0, \ldots, n) \tag{4}$$

Avoid Over-fitting

- [convexity constraint]:
  $\sum_{b=1}^{B} (w_{b,j} + e_{b,j}^W) = 1 \Leftrightarrow \sum_{b=1}^{B-1} e_{b,j}^W = -e_{B,j}^W, \; j \in [0, \ldots, n)$
- [non-negativity constraint]:
  1. $w_{b,j} + e_{b,j}^W \geq 0 \Leftrightarrow e_{b,j}^W \geq -w_{b,j}, \; b \in [1, \ldots, B-1]$
  2. $w_{B,j} + e_{B,j}^W \geq 0 \Leftrightarrow \sum_{b=1}^{B-1} e_{b,j}^W \leq w_{B,j}$

Performance/Quality Evaluation (LBS vs DQS)





Performance/Quality Evaluation (P2P-Skin vs Prior Art)

Source Data				Skinning Methods					
				SAD		FESAM		p2p-skinning	
Name	Vertices	Poses	Proxy-Joints	Error	Time	Error	Time	Error	Time
Elephant	42321	48	25	13.3	368.1	1.39	778.9	2.2	420.1
Samba	9971	175	30	12.6	380	1.17	344.2	1.98	443.5
Skirt	5095	60	75	4.8	422.6	1.72	264.9	2.81	550.9
Facial Expression	23725	23	50	38.7	283.8	5.2	551.1	7.5	332.1

[1] SAD: [Kavan et al. I3D'07], [2] FESAM:[Kavan et al. CGF'10]

Performance/Quality Evaluation (P2P-Skin vs Prior Art)

Figure: (From left to right) An approximated pose using SAD[1], using FESAM[2], using p2p-skinning, using p2p-skinning with corrections and finally the original pose

[1] SAD: [Kavan et al. I3D'07], [2] FESAM:[Kavan et al. CGF'10]

Conclusions

Approximating MAs with pose-to-pose skinning

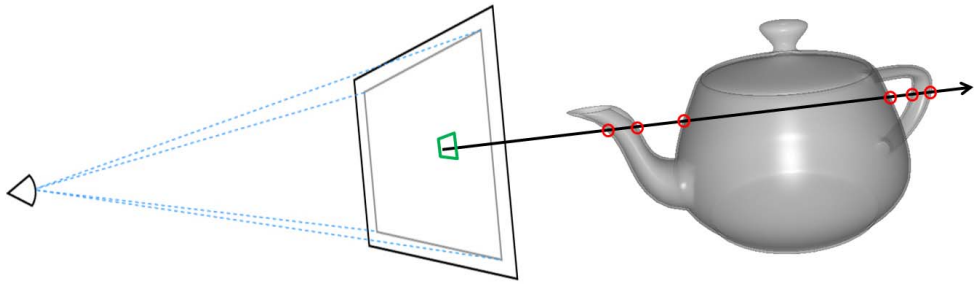
- handles both linear and non-linear fitting
- supports rigid, highly-deformable and hybrid MAs
- enables pose editing of arbitrary animation frames
- increases skinning quality via novel vertex and weight corrections

Outline

Revisiting A-buffer architecture

Memory-friendly Multi-fragment Rendering

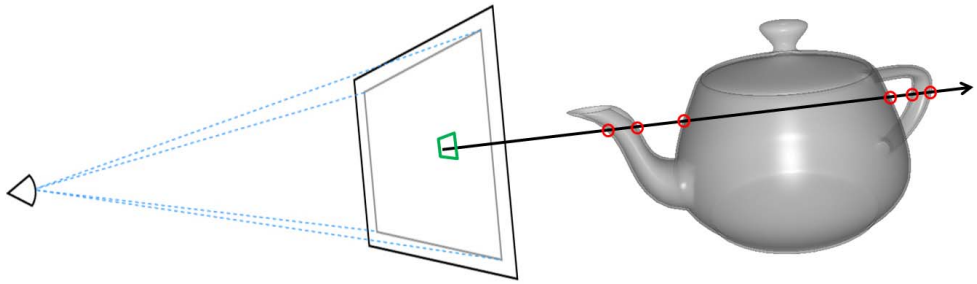
- capture all generated fragments
- memory overflow & performance bottlenecks



Revisiting A-buffer architecture

Memory-friendly Multi-fragment Rendering

- capture all generated fragments
- memory overflow & performance bottlenecks



Our contribution - *S-buffer*†

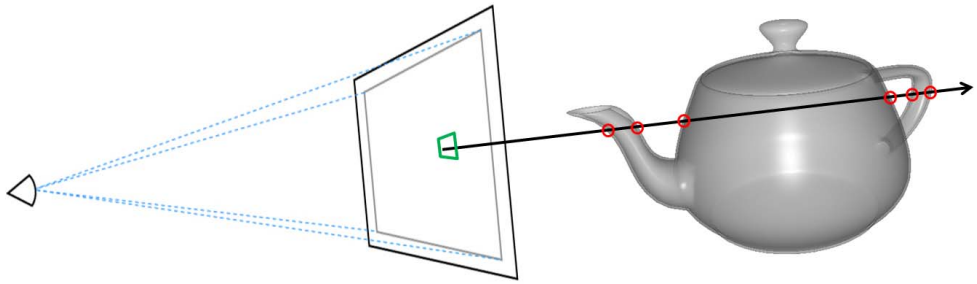
1. precise memory allocation

† **A. Vasilakis** & I. Fudos, *S-buffer: Sparsity-aware Multi-fragment Rendering*, Eurographics 2012 (Short Papers), Cagliari, Italy, May 13-18, 2012.

Revisiting A-buffer architecture

Memory-friendly Multi-fragment Rendering

- capture all generated fragments
- memory overflow & performance bottlenecks



Our contribution - *S-buffer*†

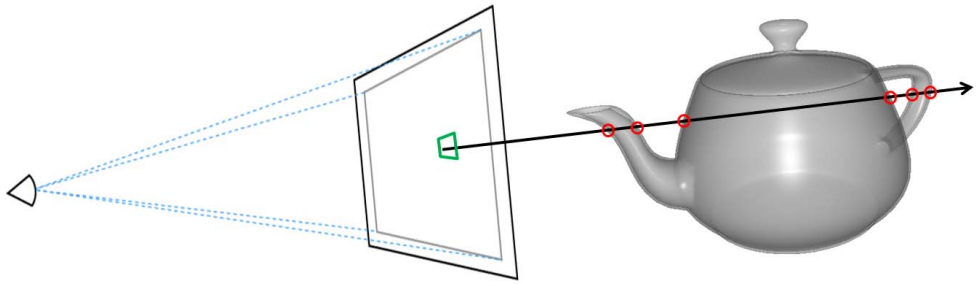
1. precise memory allocation
2. reduces fragment & pixel contention

† **A. Vasilakis** & I. Fudos, *S-buffer: Sparsity-aware Multi-fragment Rendering*, Eurographics 2012 (Short Papers), Cagliari, Italy, May 13-18, 2012.

Revisiting A-buffer architecture

Memory-friendly Multi-fragment Rendering

- capture all generated fragments
- memory overflow & performance bottlenecks



Our contribution - *S-buffer*†

1. precise memory allocation
2. reduces fragment & pixel contention
3. two-rendering-pass method

† **A. Vasilakis** & I. Fudos, *S-buffer: Sparsity-aware Multi-fragment Rendering*, Eurographics 2012 (Short Papers), Cagliari, Italy, May 13-18, 2012.

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Framework design goals

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Framework design goals

- Quality: Fragment extraction accuracy (A)

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Framework design goals

- Quality: Fragment extraction accuracy (A)
- Time performance (P)

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Framework design goals

- Quality: Fragment extraction accuracy (A)
- Time performance (P)
- Memory allocation (Ma) and caching (Mc)

Related Work

Geometry sorting methods

- object sorting
- primitive sorting

Fragment sorting methods

- depth-peeling
- hardware-implemented buffers

Framework design goals

- Quality: Fragment extraction accuracy (A)
- Time performance (P)
- Memory allocation (Ma) and caching (Mc)
- GPU capabilities (G)

Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]
- Dual direction (DUAL)[2]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]
- Dual direction (DUAL)[2]
- Uniform bucket (BUN)[3]

Characteristics

- A: z-fighting [↓] - P: multi-pass [↓]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]
- Dual direction (DUAL)[2]
- Uniform bucket (BUN)[3]

Characteristics

- A: z-fighting [↓] - P: multi-pass [↓]
- Ma: low [↑] - Mc: fast [↑]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]
- Dual direction (DUAL)[2]
- Uniform bucket (BUN)[3]

Characteristics

- A: z-fighting [↓] - P: multi-pass [↓]
- Ma: low [↑] - Mc: fast [↑]
- G: old & modern [↑]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

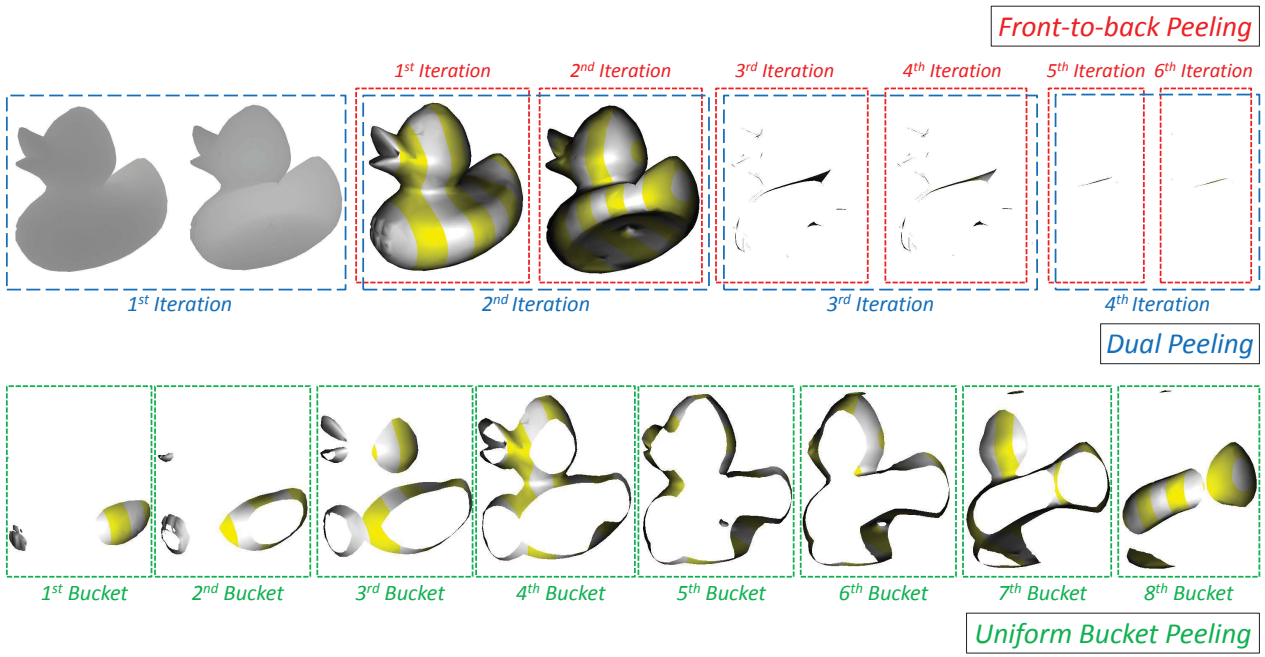
Related Work - Depth Peeling

Methods

- Front-to-back (F2B)[1]
- Dual direction (DUAL)[2]
- Uniform bucket (BUN)[3]

Characteristics

- A: z-fighting [↓] - P: multi-pass [↓]
- Ma: low [↑] - Mc: fast [↑]
- G: old & modern [↑]

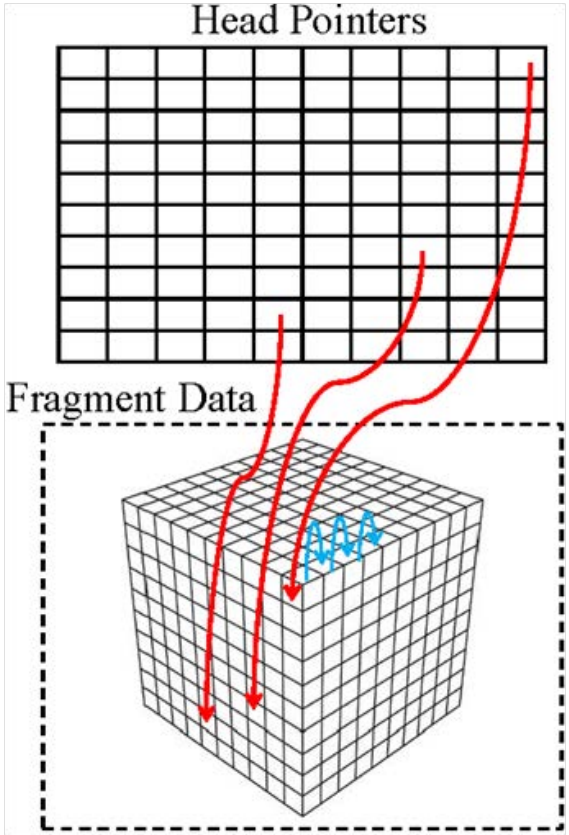


[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, HPG'09]

Related Work - Hardware Buffers (1)

Per-pixel Fixed-size Arrays

- Ma: huge (unused) [↓]
- Mc: very fast [↑]
- G: [commodity]: KB[1], KB-SR[2]
  - A: RMW hazards - 8 fragments [↓]
  - P: fast [↑]
- G: [modern]: AB_{FP}[3,4]
  - A: 100% if enough memory [↑]
  - P: fastest (single pass) [↑]

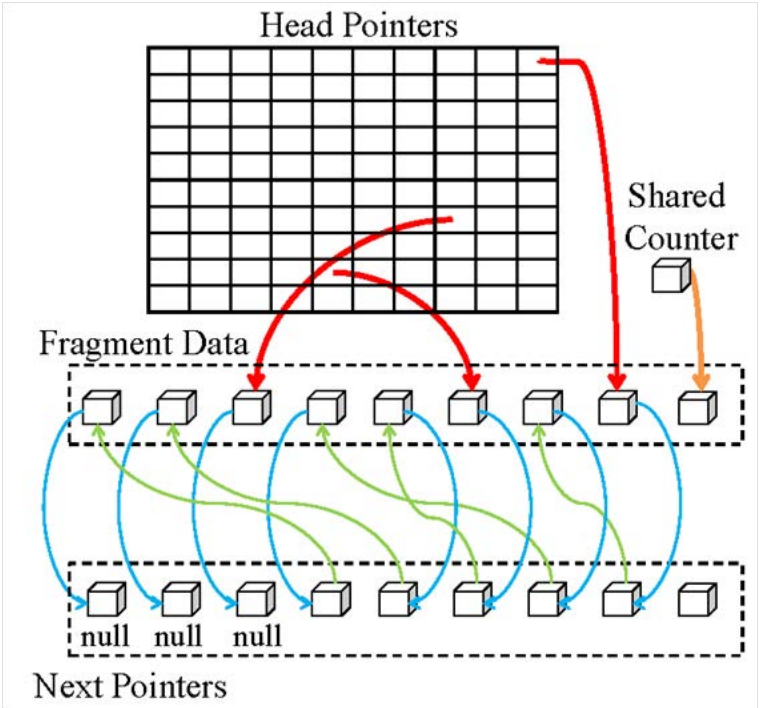


[1] [Bavoil et al, I3D'07], [2] [Myers et al, SIGGRAPH'07], [3] [Liu et al, I3D'10], [4] [Crassin, blog'11]

Related Work - Hardware Buffers (2)

Per-pixel Linked Lists (AB_{LL})[1]

- A: 100% if enough memory [↑]
- P: fast (fragment contention) [↓]
- Mc: low cache hit ratio [↓]
- Ma: high
  - if [overflow]: accurate reallocation (extra pass needed)[↓]
  - wasted memory [↓]
- G: modern cards [↓]

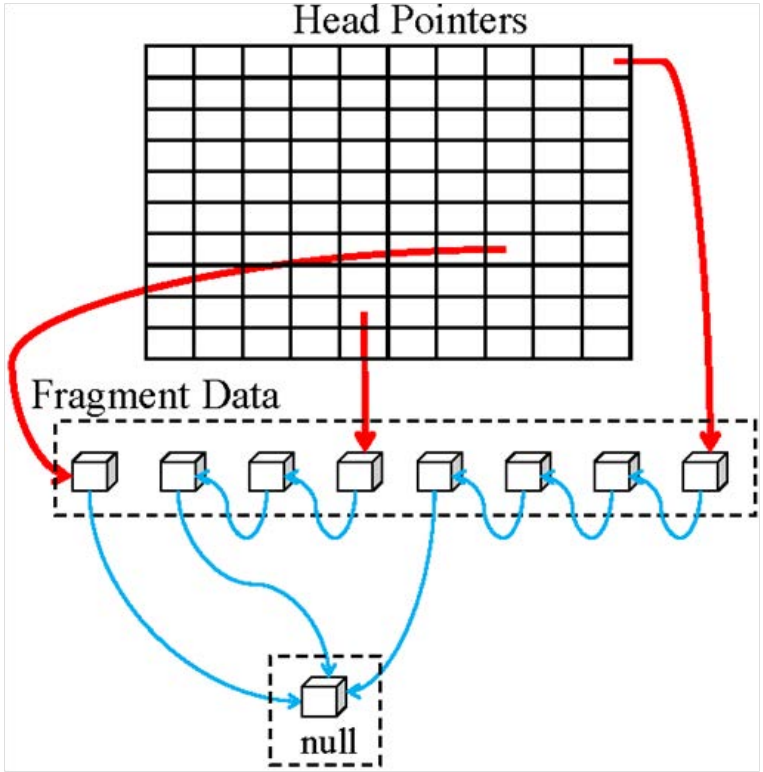


[1] [Yang et al, CGF'10]

Related Work - Hardware Buffers (3)

Per-pixel Variable-length Arrays

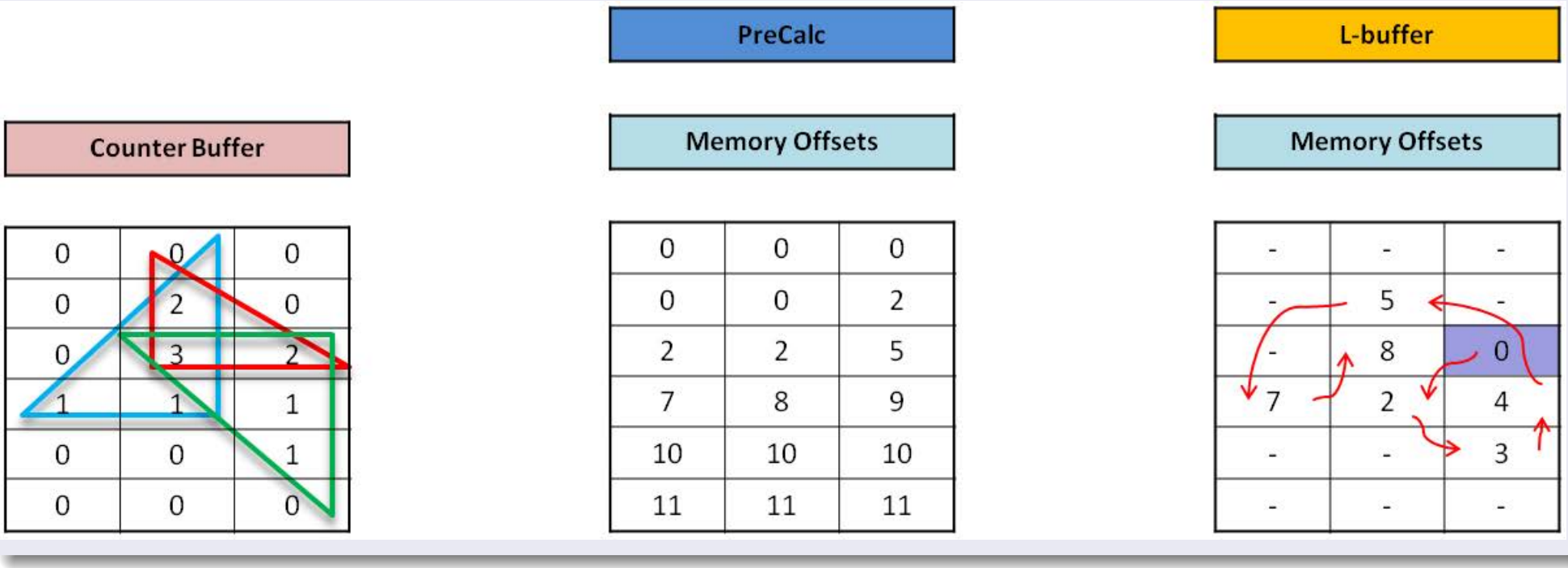
- A: 100% if enough memory [↑]
- P: fast (2 passes needed) [↑]
- Ma: precise allocation [↑]
- Mc: fast [↑]
- G: [commodity]
  - PreCalc[1] (prefix sum)
  - I-buffer[2] (randomized prefix sum)
- G: [modern]
  - DFB[3] (parallel prefix sum)



[1] [Peeper, patent'08], [2] [Lipowski, ICCVG'10], [3] [Maule et al, CG'14]

Related Work - Hardware Buffers (3)

Example: PreCalc[1] & l-buffer[2]



[1] [Peeper, patent'08], [2] [Lipowski, ICCVG'10]

S-buffer Pipeline

1. Fragment Count Rendering Pass

- # fragments per pixel
- total generated fragments

S-buffer Pipeline

2. Memory Referencing Full-screen Pass

[parallelized randomized prefix sum]:

- S multiple [shared counters]: $C = \{C(0), \ldots, C(S-1)\}$

S-buffer Pipeline

2. Memory Referencing Full-screen Pass

[parallelized randomized prefix sum]:

- S multiple [shared counters]: $C = \{C(0), \ldots, C(S-1)\}$
- simple [hash function]: $H(p) = (p.\mathrm{x} + sc.\mathrm{width} * p.\mathrm{y})\%S$

S-buffer Pipeline

2. Memory Referencing Full-screen Pass

[parallelized randomized prefix sum]:

- S multiple [shared counters]: $C = \{C(0), \ldots, C(S-1)\}$
- simple [hash function]: $H(p) = (p.\mathrm{x} + sc.\mathrm{width} * p.\mathrm{y})\%S$
- [sequential prefix sum] on shared counters: $C_{pr}(i) = \sum_0^{i-1} C(i)$

S-buffer Pipeline

2. Memory Referencing Full-screen Pass

[parallelized randomized prefix sum]:

- S multiple [shared counters]: $C = \{C(0), \ldots, C(S-1)\}$
- simple [hash function]: $H(p) = (p.\mathrm{x} + sc.\mathrm{width} * p.\mathrm{y})\%S$
- [sequential prefix sum] on shared counters: $C_{pr}(i) = \sum_0^{i-1} C(i)$
- [inverse mapping extension]:
  - [split] C: $G_1 = \{C(0), \ldots, C(\lfloor \frac{S}{2} \rfloor)\}, G_2 = \{C(\lfloor \frac{S}{2} \rfloor + 1), \ldots, C(S-1)\}$

S-buffer Pipeline

2. Memory Referencing Full-screen Pass

[parallelized randomized prefix sum]:

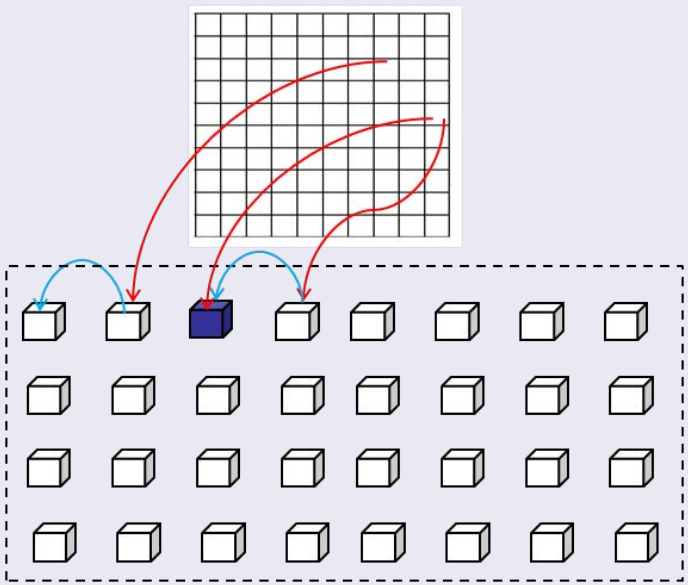
- S multiple [shared counters]: $C = \{C(0), \ldots, C(S-1)\}$
- simple [hash function]: $H(p) = (p.\text{x} + sc.\text{width} * p.\text{y})\%S$
- [sequential prefix sum] on shared counters: $C_{pr}(i) = \sum_0^{i-1} C(i)$
- [inverse mapping extension]:
  - [split] C: $G_1 = \{C(0), \ldots, C(\lfloor \frac{S}{2} \rfloor)\}$, $G_2 = \{C(\lfloor \frac{S}{2} \rfloor + 1), \ldots, C(S-1)\}$
  - [final memory offset]:

$$p.\text{offset} = \begin{cases} A(p), & \text{if } p \in G_1 \\ total\_fragments\text{-}1\text{-}A(p), & \text{otherwise} \end{cases}$$

$$\text{where } A(p) = p.\text{address} + C_{pr}(H(p))$$

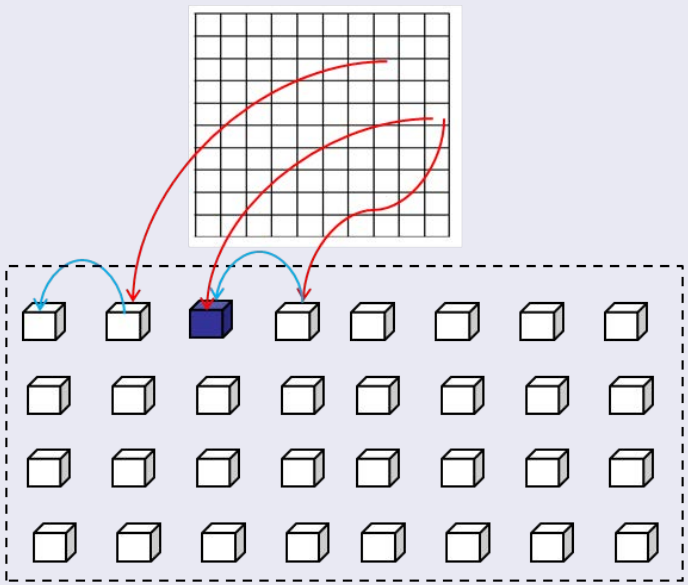
S-buffer Pipeline

3. Fragment Storing Rendering Pass



S-buffer Pipeline

3. Fragment Storing Rendering Pass

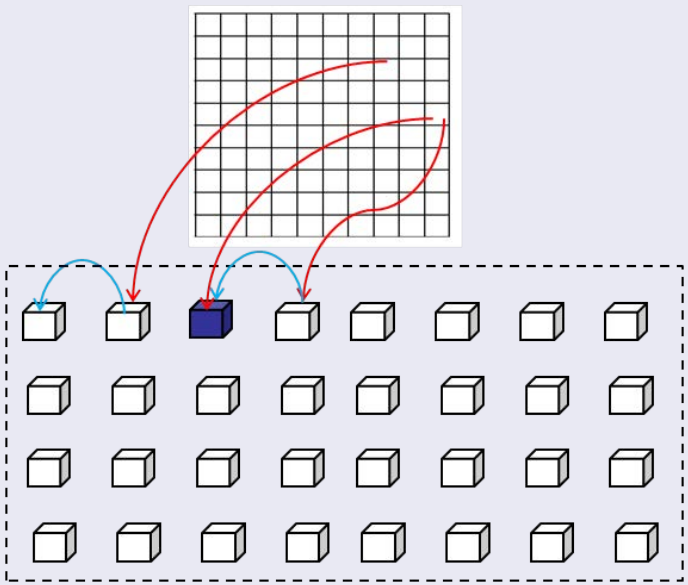


4. Resolve Full-screen Pass

1. [Fragment sorting]
2. [Effect]

S-buffer Pipeline

3. Fragment Storing Rendering Pass

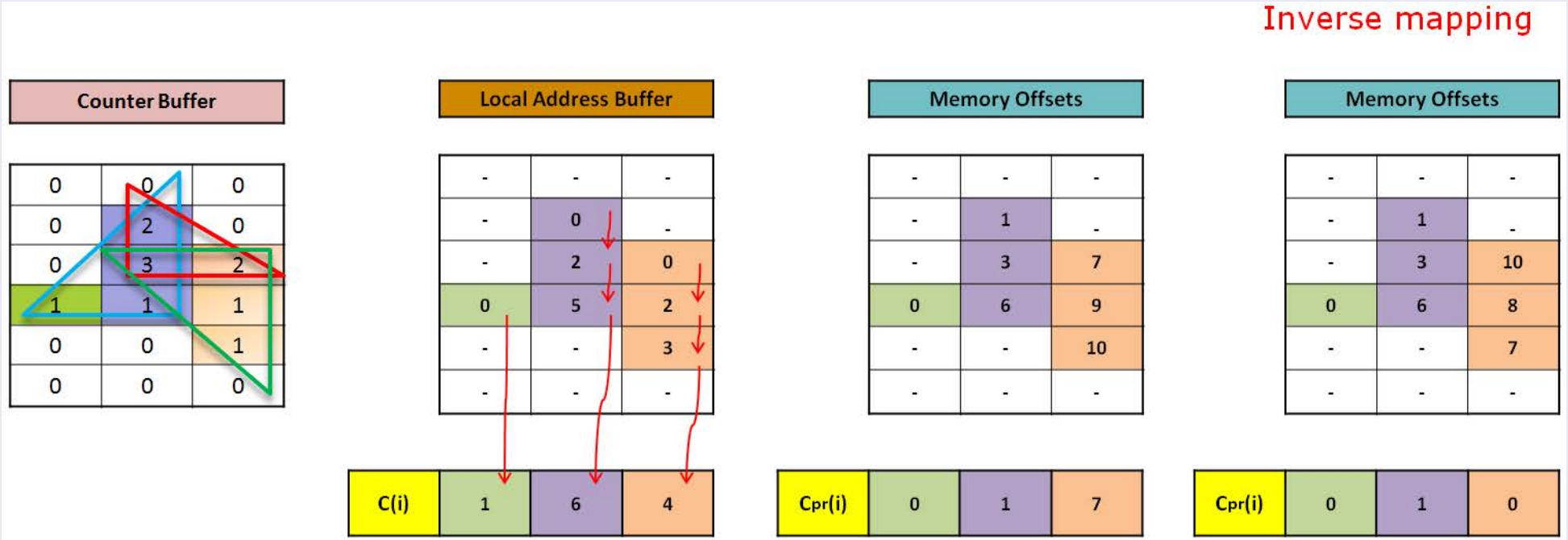


4. Resolve Full-screen Pass

1. [Fragment sorting]
2. [Effect]

S-buffer Pipeline

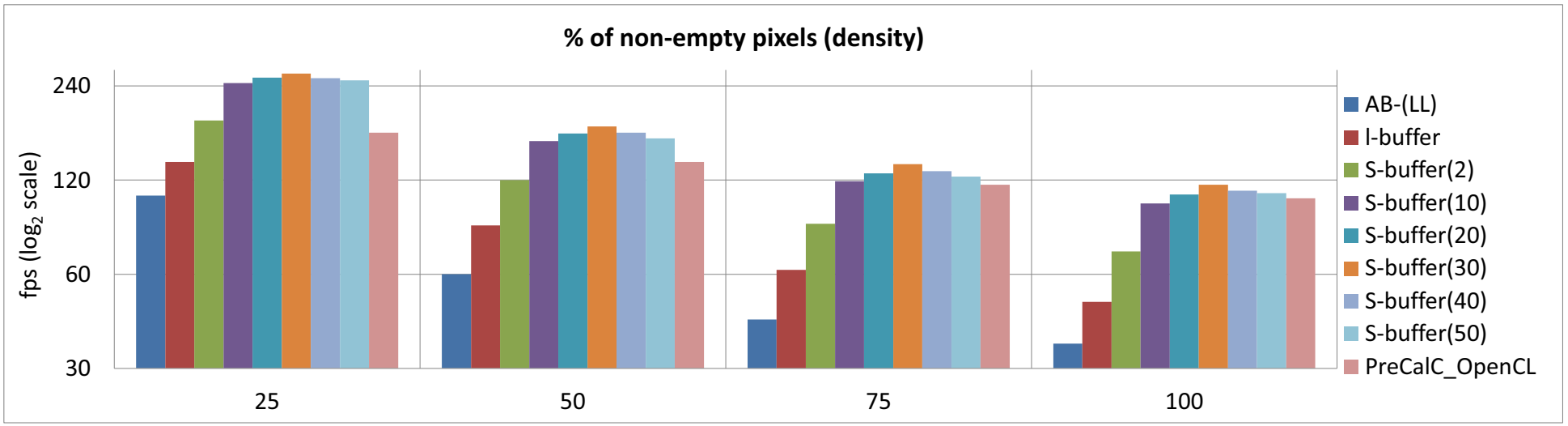
Example: using 3 shared counters



Performance Analysis (1)

Impact of Sparsity

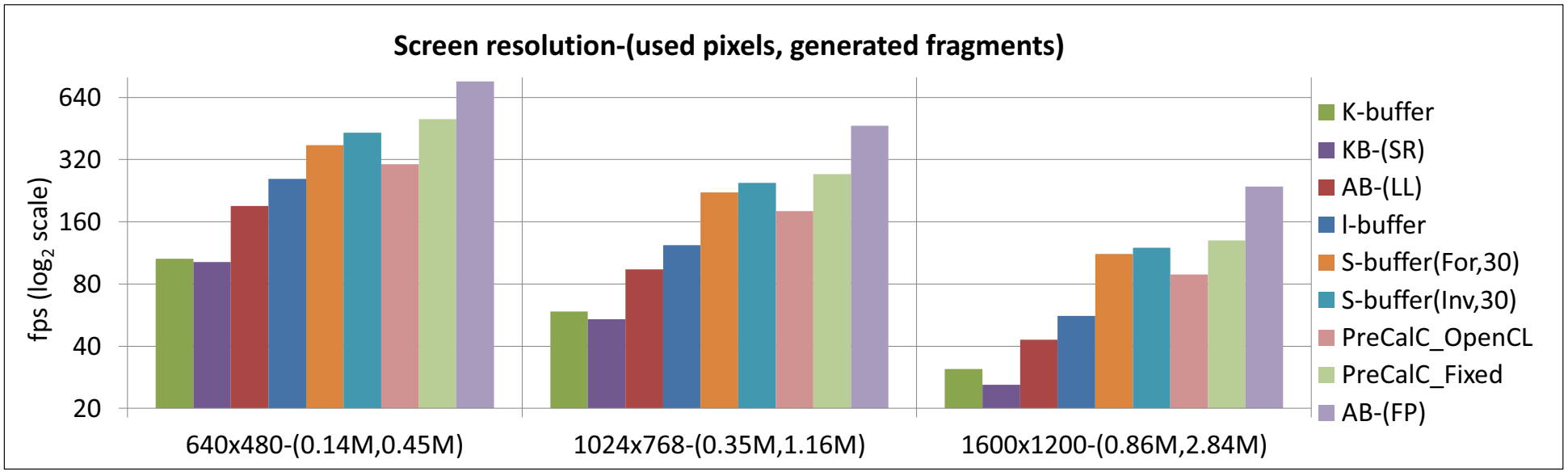
- [Scene]: $70,000$ triangles, 12 layers, 1024^2 viewport
- [AB_{LL}]: $O(m)$, $m(> n)$ = total fragments
- [l-buffer]: $O(n)$, n = non-empty pixels
- [S-buffer] speeds up: n/S, S shared counters
- [PreCalc_OpenCL]: OpenGL/OpenCL syncing time



Performance Analysis (2)

Impact of Resolution

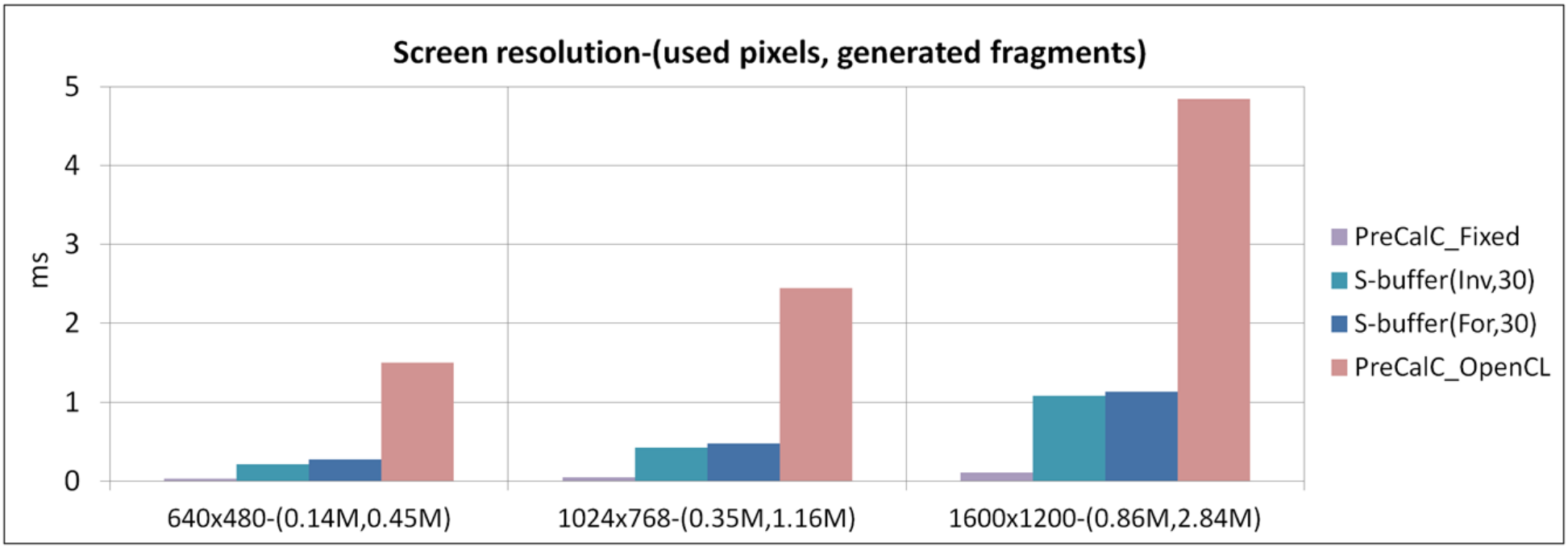
- [Scene]: 110,000 triangles, 25 layers, $p_d = 45\%$
- [S-buffer] = 85% of PreCalc_Fixed speed
- [forward] vs [inverse mapping]



Performance Analysis (2)

Impact of Resolution

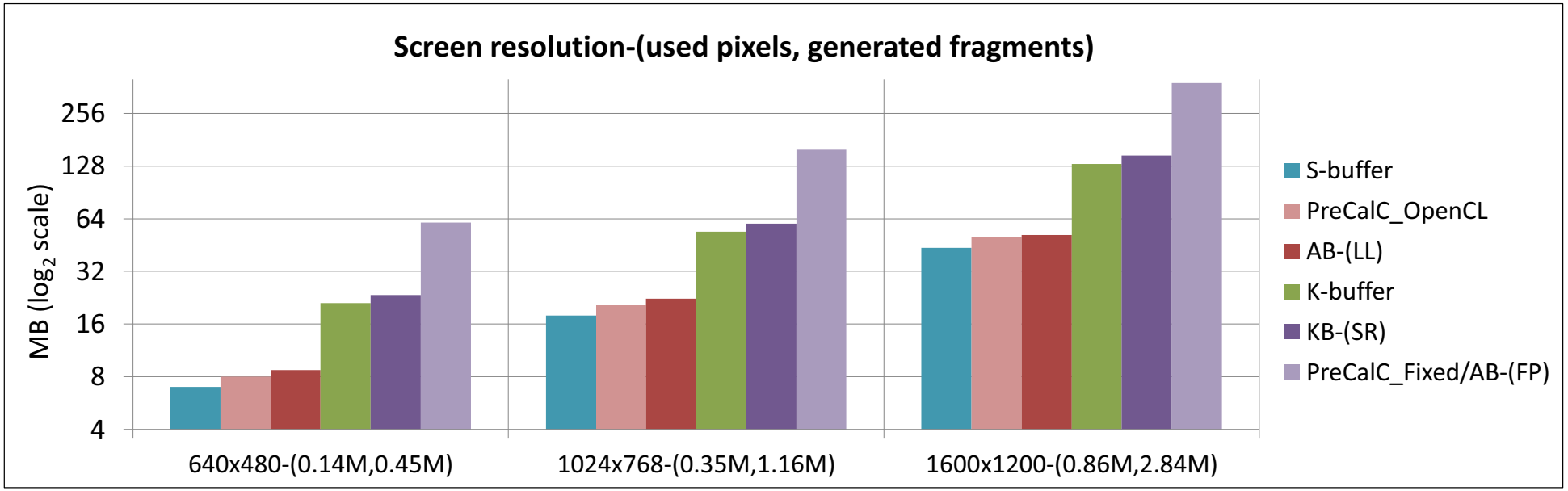
- [Scene]: $110,000$ triangles, 25 layers, $p_d = 45\%$
- [S-buffer] $= 85\%$ of PreCalc_Fixed speed
- [forward] vs [inverse mapping]



Memory Allocation Analysis

Impact of Resolution

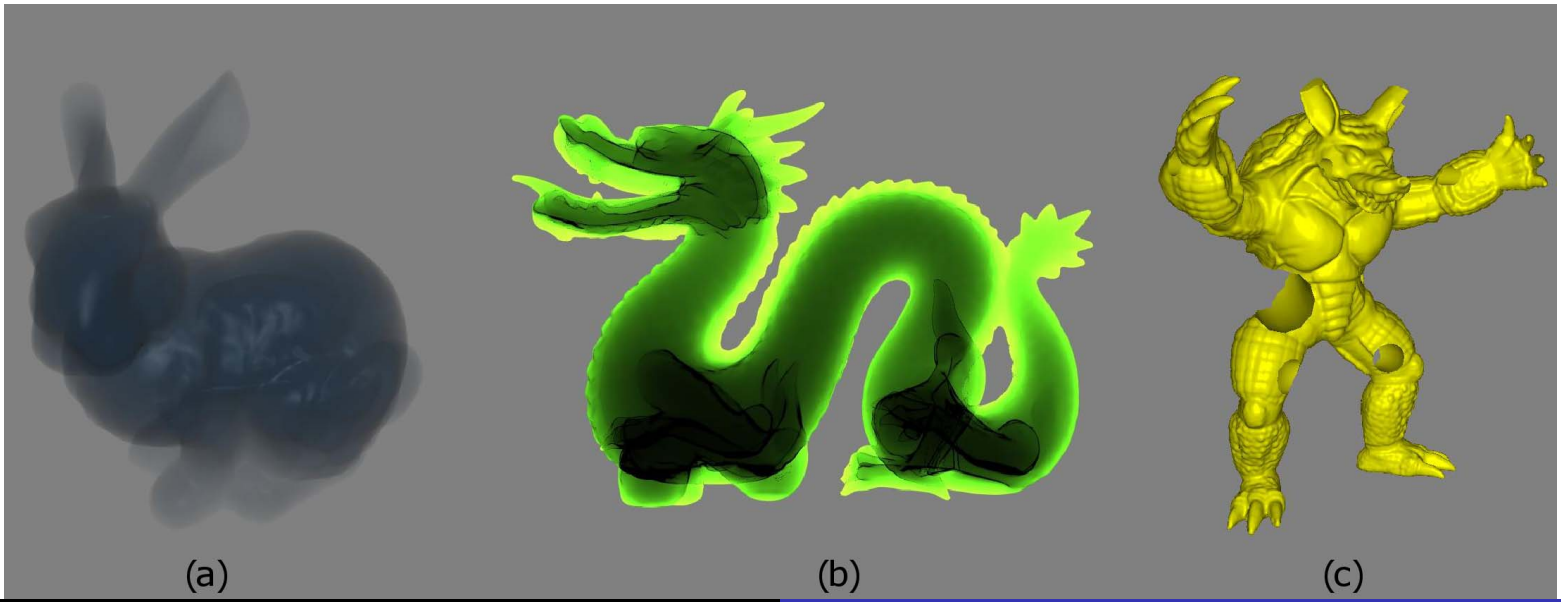
- [Scene]: 110,000 triangles, 25 layers, $p_d = 45\%$
- [fixed-sized arrays]:
  - AB_{FP}: Wasted resources (88%)
  - KB,KB-SR: 30% less memory due to 8 fragments/pixel
- [AB_{LL}]: extra memory (pointer list)



Conclusions

Multi-fragment rendering via S-buffer

- two-geometry-pass GPU-accelerated A-buffer that take advantage of
  - the fragment distribution (exact memory allocation)
  - the sparsity of the pixel-space (improved performance)
- organizes storage into variable contiguous regions for each pixel
- integrates into the standard graphics pipeline



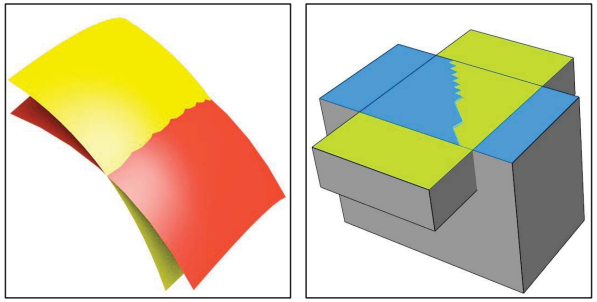
Outline

Extracting coplanar fragments

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces
- multifragment rasterization is even more susceptible

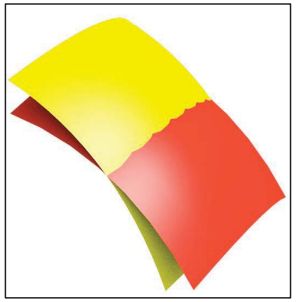


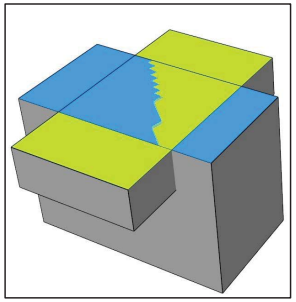
Extracting coplanar fragments

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces
- multifragment rasterization is even more susceptible





Our contribution†

1. [robust] & [approximate] methods

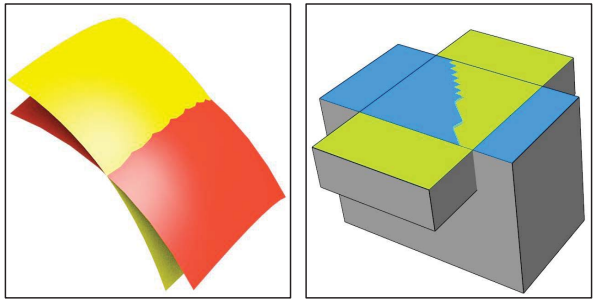
† **A. Vasilakis** & I. Fudos, *Depth-Fighting Aware Methods for Multi-fragment Rendering*, IEEE TVCG, vol. 19, no. 6, pp. 967-977, June, 2013.

Extracting coplanar fragments

Z-fighting phenomenon

more

- two or more primitives have same depth
- intersecting or overlapping surfaces
- multifragment rasterization is even more susceptible



Our contribution†

1. [robust] & [approximate] methods
2. [geometry culling] mechanism for multi-pass rendering

† **A. Vasilakis** & I. Fudos, *Depth-Fighting Aware Methods for Multi-fragment Rendering*, IEEE TVCG, vol. 19, no. 6, pp. 967-977, June, 2013.

Correcting Raster-based Pipelines

Adapting depth peeling methods based on

1. primitive identifiers

Correcting Raster-based Pipelines

Adapting depth peeling methods based on

1. primitive identifiers
2. buffer-based solutions

Correcting Raster-based Pipelines

Adapting depth peeling methods based on

1. primitive identifiers
2. buffer-based solutions

- MSAA - Tessellation - Instancing

Correcting Raster-based Pipelines

Adapting depth peeling methods based on

1. primitive identifiers
2. buffer-based solutions

- MSAA - Tessellation - Instancing

Robustness ratio

1. [robust]: Ma: low [↑] - P: slow [↓]

Correcting Raster-based Pipelines

Adapting depth peeling methods based on

1. primitive identifiers
2. buffer-based solutions

- MSAA - Tessellation - Instancing

Robustness ratio

1. [robust]: Ma: low [↑] - P: slow [↓]
2. [approximate]: Ma: high [↓] - P: efficient [↑]

Robust Algorithms (1)

Extending F2B[1],DUAL[2] $\rightarrow$ F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08]

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)
- [Idea]: extra accumulation rendering pass
  - primitive ID (OpenGl: gl_PrimitiveID)
  - store min/max IDs of the remaining non-peeled fragments: $id^1_{\text{min}} < id^2_{\text{min}} < \cdots < id^2_{\text{max}} < id^1_{\text{max}}$

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08]

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)
- [Idea]: extra accumulation rendering pass
  - primitive ID (OpenGl: gl_PrimitiveID)
  - store min/max IDs of the remaining non-peeled fragments: $id^1_{\text{min}} < id^2_{\text{min}} < \cdots < id^2_{\text{max}} < id^1_{\text{max}}$
- subsequent rendering pass:
  1. extract fragment information using captured IDs

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08]

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)
- [Idea]: extra accumulation rendering pass
  - primitive ID (OpenGl: gl_PrimitiveID)
  - store min/max IDs of the remaining non-peeled fragments: $id^1_{\text{min}} < id^2_{\text{min}} < \cdots < id^2_{\text{max}} < id^1_{\text{max}}$
- subsequent rendering pass:
  1. extract fragment information using captured IDs
  2. move or not to next depth layer (fragment coplanarity counter)

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08]

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)
- [Idea]: extra accumulation rendering pass
  - primitive ID (OpenGl: gl_PrimitiveID)
  - store min/max IDs of the remaining non-peeled fragments: $id^1_{\text{min}} < id^2_{\text{min}} < \cdots < id^2_{\text{max}} < id^1_{\text{max}}$
- subsequent rendering pass:
  1. extract fragment information using captured IDs
  2. move or not to next depth layer (fragment coplanarity counter)

Extending F2B-2P → F2B-3P

- additional rendering pass: (double speed depth, early-z culling)

Robust Algorithms (1)

Extending F2B[1],DUAL[2] → F2B-2P,DUAL-2P

example

- base methods extract only one coplanar fragment
- extracts 2 fragments/iteration Ma: constant (low)
- [Idea]: extra accumulation rendering pass
  - primitive ID (OpenGl: gl_PrimitiveID)
  - store min/max IDs of the remaining non-peeled fragments: $id^1_{\text{min}} < id^2_{\text{min}} < \cdots < id^2_{\text{max}} < id^1_{\text{max}}$
- subsequent rendering pass:
  1. extract fragment information using captured IDs
  2. move or not to next depth layer (fragment coplanarity counter)

Extending F2B-2P → F2B-3P

- additional rendering pass: (double speed depth, early-z culling)
- P: [↑] Ma: [-]

Robust Algorithms (2)

Combining F2B[1],DUAL[2] with AB_{LL}[3] → F2B-LL,DUAL-LL

example

- handle fragment coplanarity of arbitrary length per pixel

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Yang et al, CGF'10]

Robust Algorithms (2)

Combining F2B[1],DUAL[2] with AB_{LL}[3] → F2B-LL,DUAL-LL

example

- handle fragment coplanarity of arbitrary length per pixel
- rendering workflow (2 passes/depth layer)
  1. double speed depth pass
  2. fragment linked lists at the current depth layer

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Yang et al, CGF'10]

Robust Algorithms (2)

Combining F2B[1],DUAL[2] with AB_{LL}[3] → F2B-LL,DUAL-LL

example

- handle fragment coplanarity of arbitrary length per pixel
- rendering workflow (2 passes/depth layer)
  1. double speed depth pass
  2. fragment linked lists at the current depth layer

Limitations

- P: [↓]
  1. multi-pass rendering
  2. fragment storing (reduced congestion)
- G: only modern cards - [↓]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature
- uniformly split of the depth range
  - render bounding box
  - max 5 consecutive subintervals

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature
- uniformly split of the depth range
  - render bounding box
  - max 5 consecutive subintervals
- assign a linked list to each subdivision

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature
- uniformly split of the depth range
  - render bounding box
  - max 5 consecutive subintervals
- assign a linked list to each subdivision

Limitations

- P:
  - always faster than AB_{LL} – [↑]
  - uniform depth distribution - [↓]

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature
- uniformly split of the depth range
  - render bounding box
  - max 5 consecutive subintervals
- assign a linked list to each subdivision

Limitations

- P:
  - always faster than AB_{LL} – [↑]
  - uniform depth distribution - [↓]
- Ma: (extra memory) - [↓], Mc: [↓]

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Robust Algorithms (3)

Combining Uniform Buckets[1] with AB_{LL}[2] $\rightarrow$ BUN-LL

example

- single-pass nature
- uniformly split of the depth range
  - render bounding box
  - max 5 consecutive subintervals
- assign a linked list to each subdivision

Limitations

- P:
  - always faster than AB_{LL} – [↑]
  - uniform depth distribution - [↓]
- Ma: (extra memory) - [↓], Mc: [↓]
- G: only modern cards - [↓]

[1] [Liu, HPG'09], [2] [Yang et al, CGF'10]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known
    2. no memory overflow

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known
    2. no memory overflow
- G: commodity - [KB][4] → F2B-KB,DUAL-KB

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known
    2. no memory overflow
- G: commodity - [KB][4] → F2B-KB,DUAL-KB
  - 8 coplanar fragments/layer

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known
    2. no memory overflow
- G: commodity - [KB][4] → F2B-KB,DUAL-KB
  - 8 coplanar fragments/layer
  - [data packing]: 32 coplanar fragments/layer

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Approximate Algorithms

Combining F2B[1],DUAL[2] with fixed-size arrays → F2B-B,DUAL-B

example

- G: modern - $[AB_{FP}]$[3] → F2B-FP,DUAL-FP
  - bounded-length vectors per pixel
  - A: 100% - [↑]
    1. max{coplanar fragments/depth layer} known
    2. no memory overflow
- G: commodity - [KB][4] → F2B-KB,DUAL-KB
  - 8 coplanar fragments/layer
  - [data packing]: 32 coplanar fragments/layer
  - no sorting needed: [RMW hazard-free]

[1] [Everitt, Nvidia'01], [2] [Bavoil et al, Nvidia'08], [3] [Liu et al, I3D'10], [4] [Baboil et al, I3D'07]

Optimizing multi-pass rendering of multiple objects

Occlusion culling mechanism

- geometry is not rendered when is hidden by objects closer to camera

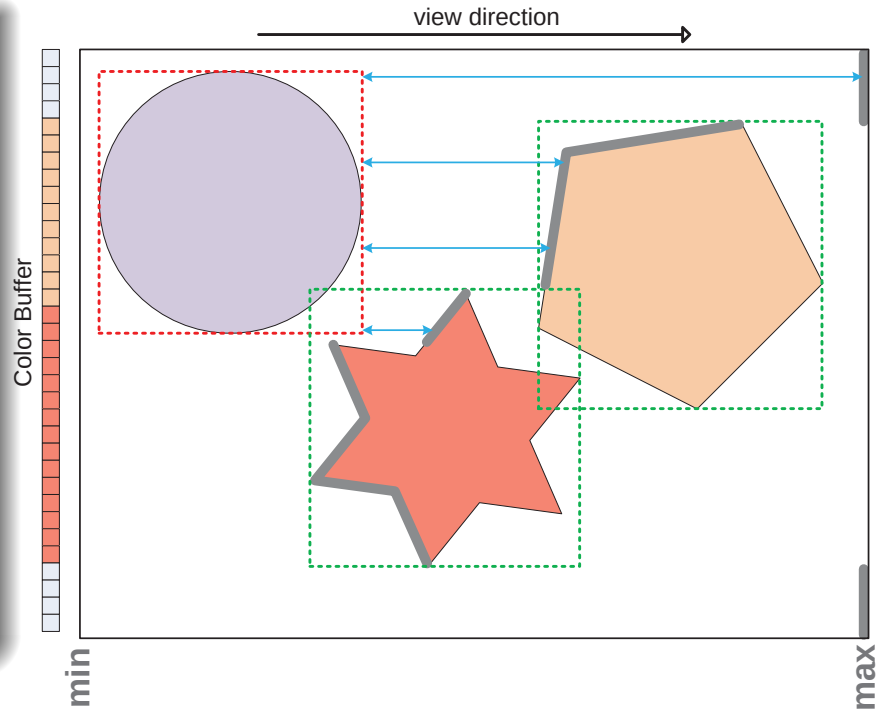
Optimizing multi-pass rendering of multiple objects

Occlusion culling mechanism

- geometry is not rendered when is hidden by objects closer to camera

Avoid rendering *completely-peeled* objects

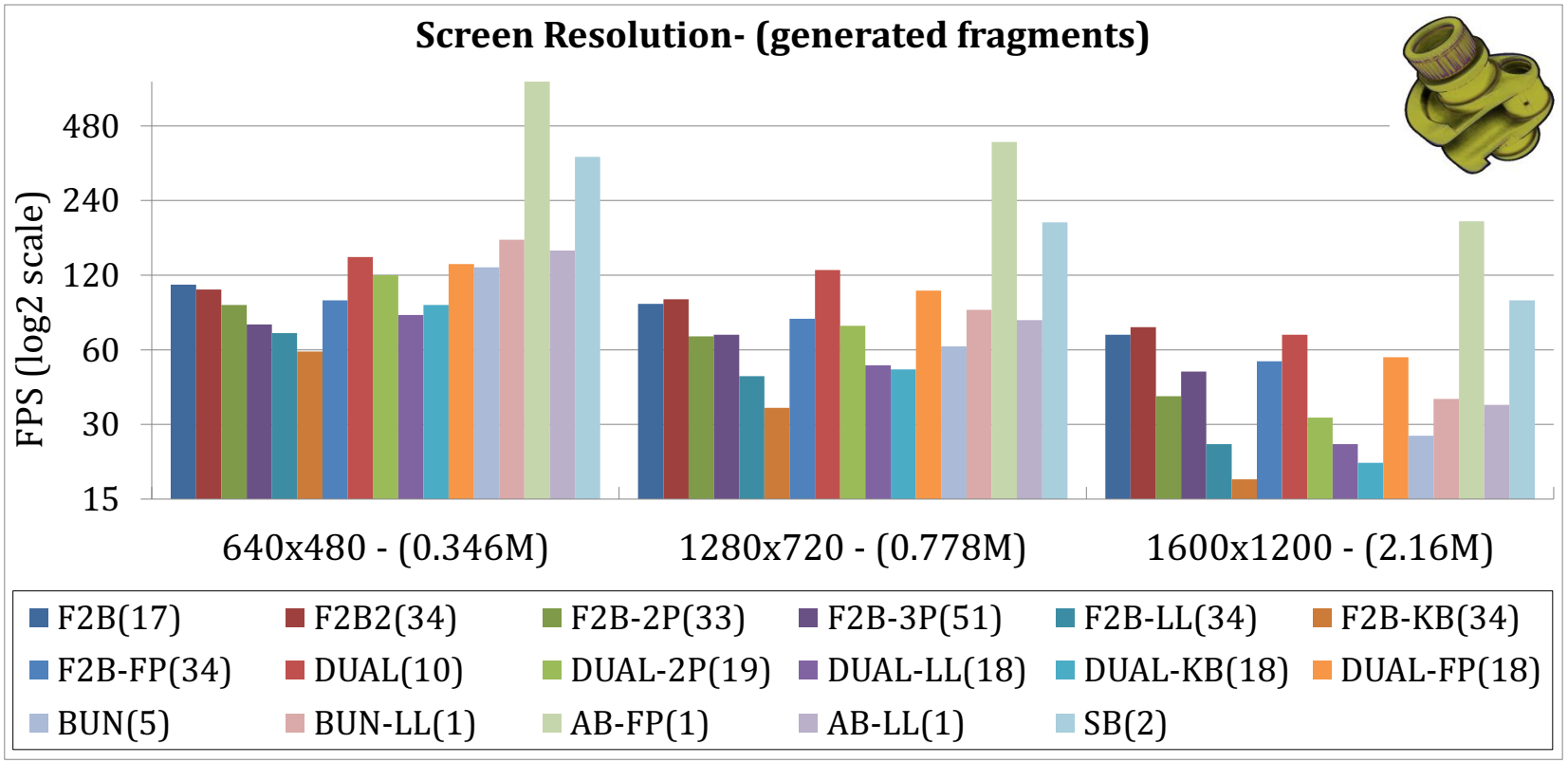
- [Goal]: rendering load reduction of the following passes
- if objects bounding box is behind current depth layer then [cull]
- hardware occlusion queries
- reuse query results from previous iterations



Performance Analysis (1)

Impact of Screen Resolution

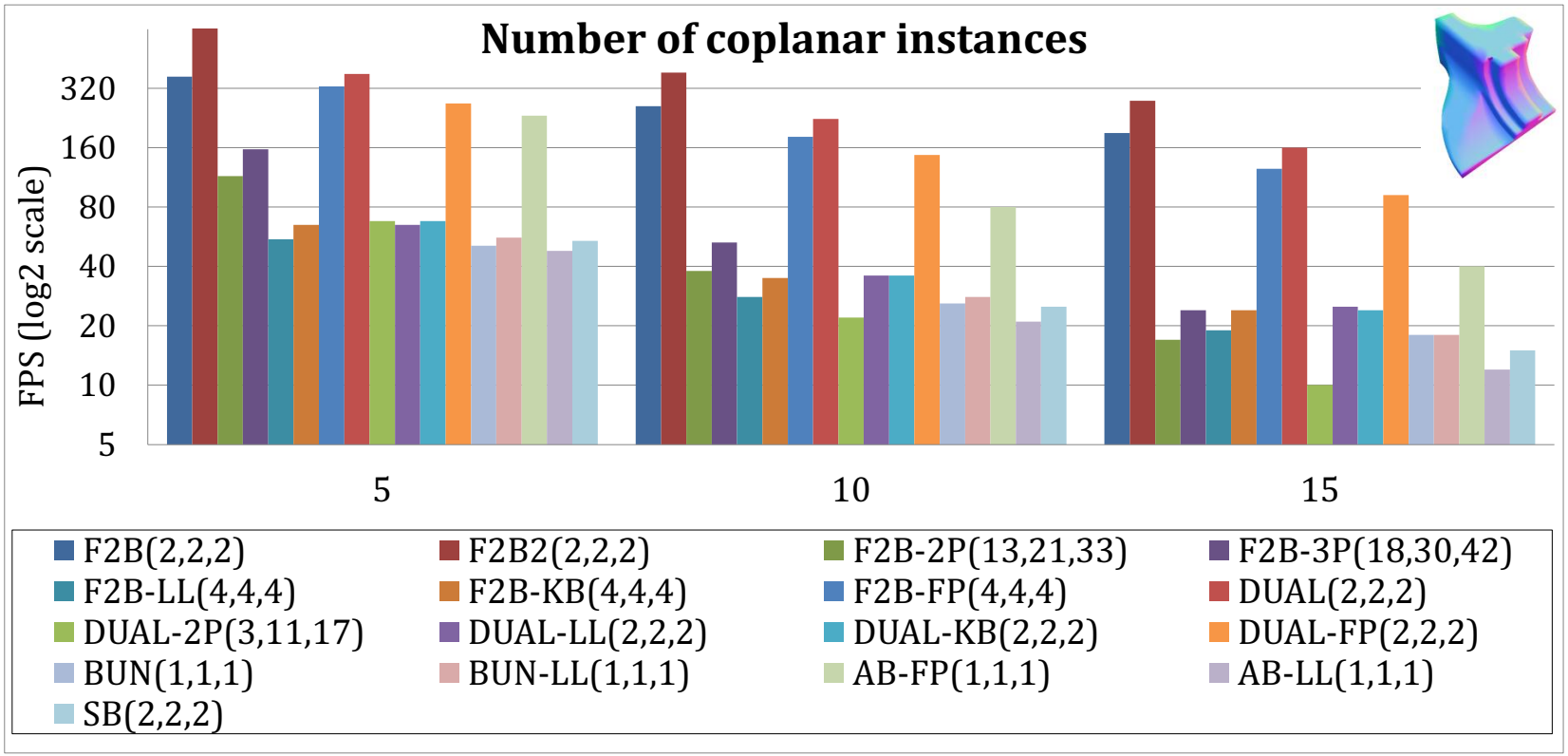
- [Crank]: 10K triangles, 17 depth layers, no coplanarity



Performance Analysis (2)

Impact of Coplanarity

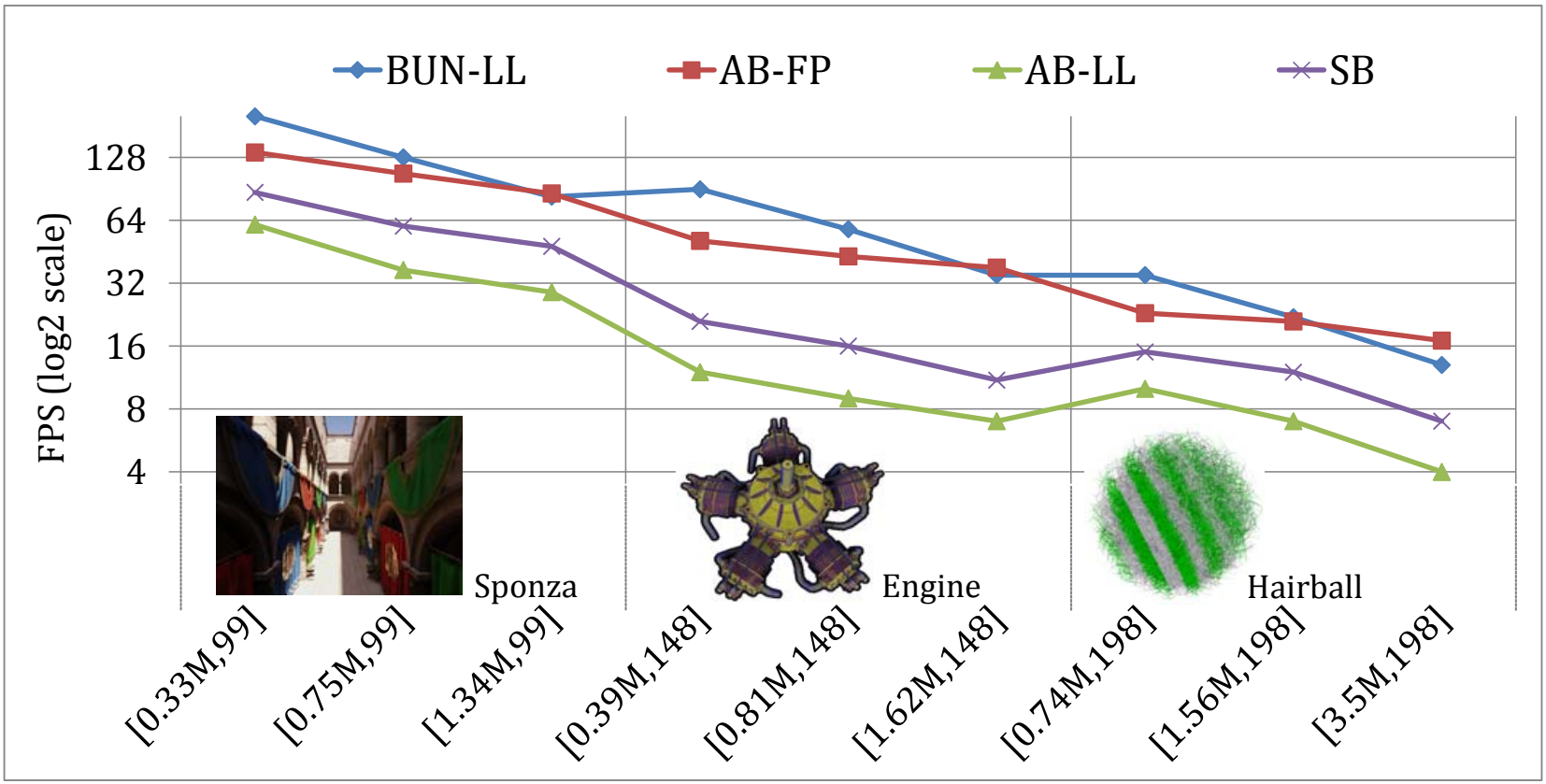
- [Fandisk]: 2K triangles, 2 depth layers, fragments/layer=instances



Performance Analysis (3)

Impact of High Depth Complexity

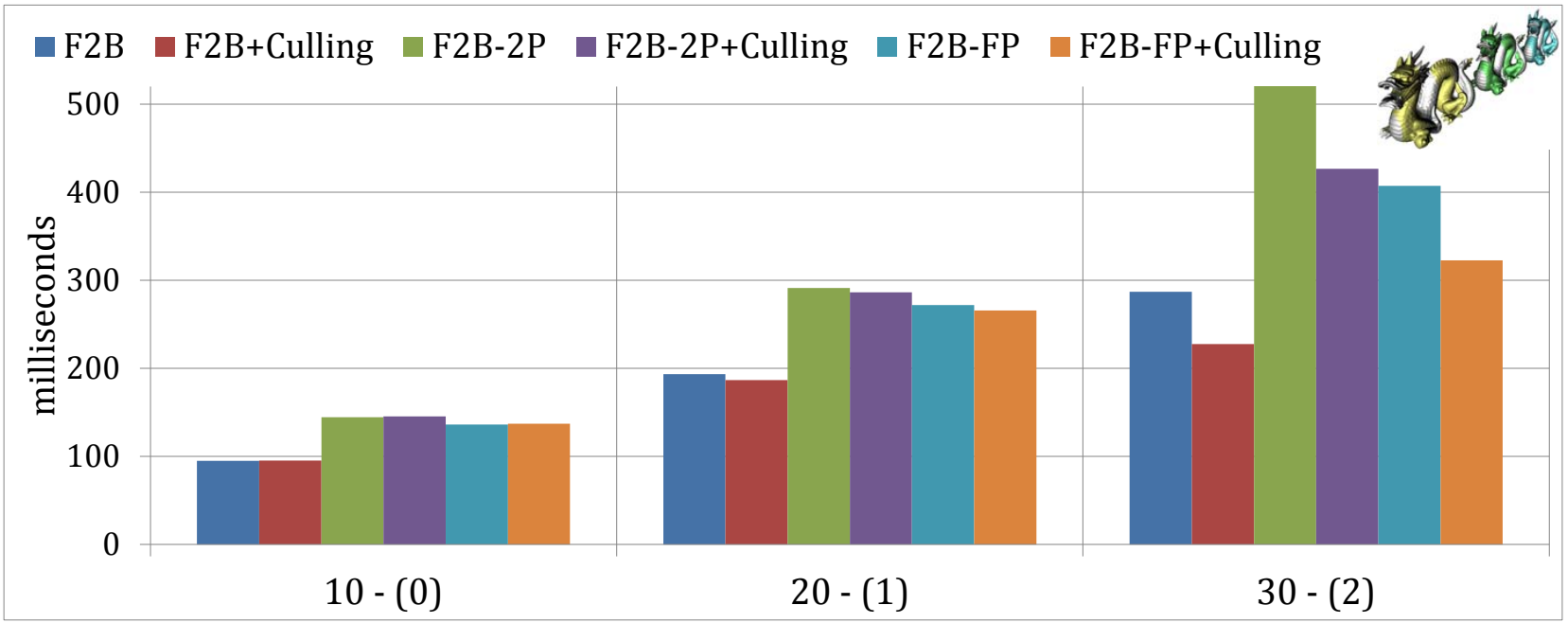
- [Sponza] (279K) [Engine] (203K) [Hairball] (2.85M) triangles



Performance Analysis (4)

Impact of Culling

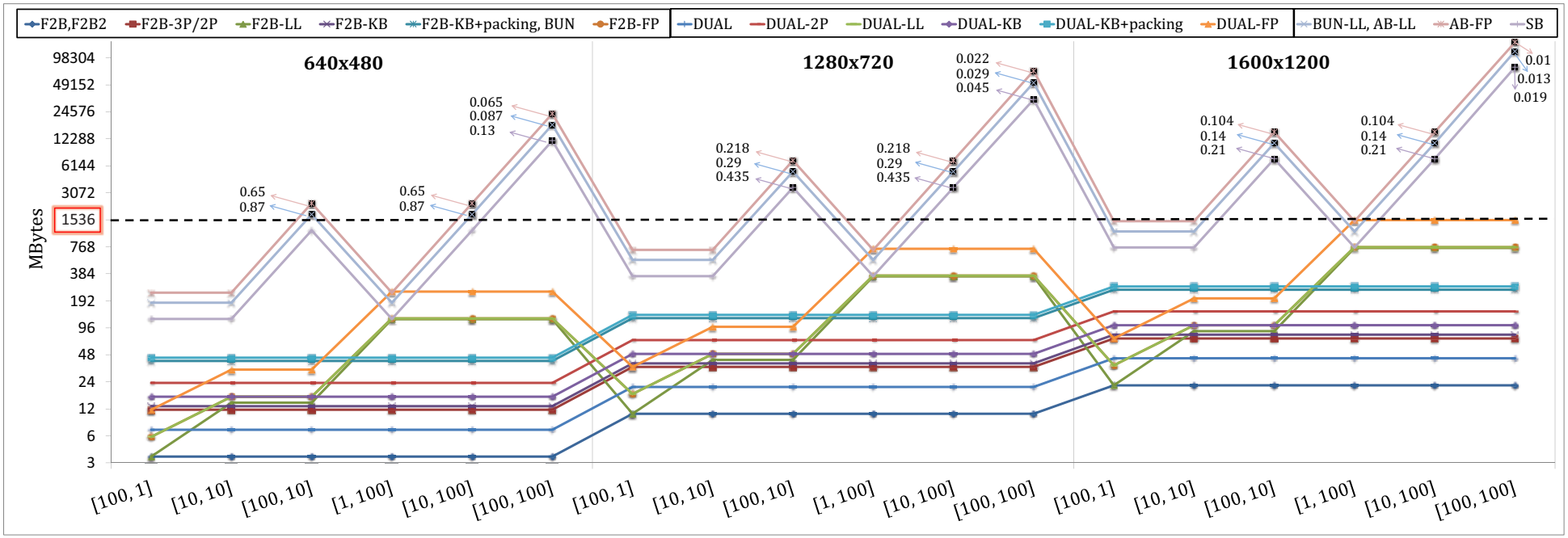
- [Dragon]: 870K triangles, 10 depth layers



Memory Allocation Analysis

Impact of # Generated Fragments

- [Robustness ratio] ???



Discussion

G: modern hardware

- [Low Memory]: Winner(FAB)
- [Medium Memory]:
  - [Low depth complexity]: Winner(SB)
  - [High depth complexity]: Winner(BUN-LL)
- [High Memory]:
  - [Low coplanarity]: Winner(F2B-FAB, DUAL-FAB)
  - [High coplanarity]: Winner(F2B-LL, DUAL-LL)

Discussion

G: modern hardware

- [Low Memory]: Winner(FAB)
- [Medium Memory]:
  - [Low depth complexity]: Winner(SB)
  - [High depth complexity]: Winner(BUN-LL)
- [High Memory]:
  - [Low coplanarity]: Winner(F2B-FAB, DUAL-FAB)
  - [High coplanarity]: Winner(F2B-LL, DUAL-LL)

G: commodity hardware

- [Low coplanarity]: Winner(F2B-3P, DUAL-2P)
- [High coplanarity]: Winner(F2B-KB, DUAL-KB)

Discussion

G: modern hardware

- [Low Memory]: Winner(FAB)
- [Medium Memory]:
  - [Low depth complexity]: Winner(SB)
  - [High depth complexity]: Winner(BUN-LL)
- [High Memory]:
  - [Low coplanarity]: Winner(F2B-FAB, DUAL-FAB)
  - [High coplanarity]: Winner(F2B-LL, DUAL-LL)

G: commodity hardware

- [Low coplanarity]: Winner(F2B-3P, DUAL-2P)
- [High coplanarity]: Winner(F2B-KB, DUAL-KB)

F2B versus DUAL extensions

- [Tessellation]: low/high - [Resolution]: low/high

Outline

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments

[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs

[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]

[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]

[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

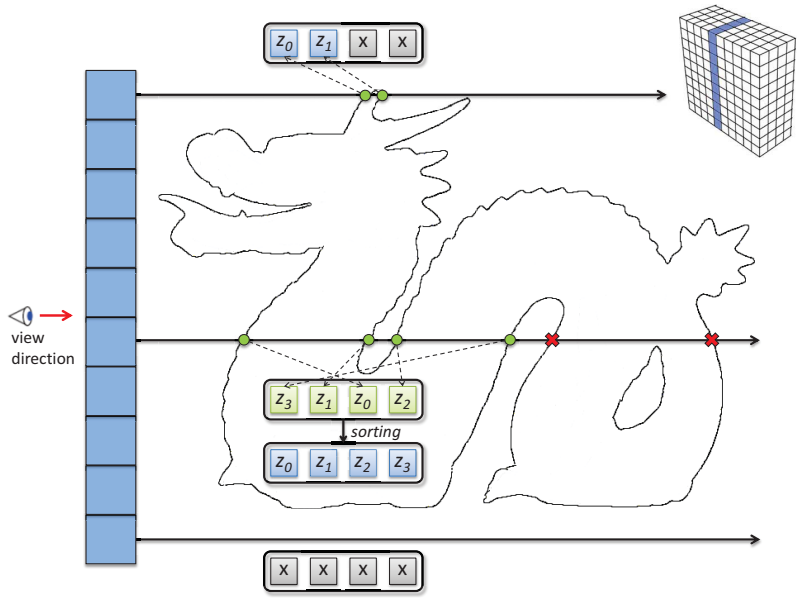
- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]
  4. unbounded memory[4]

[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]
  4. unbounded memory[4]

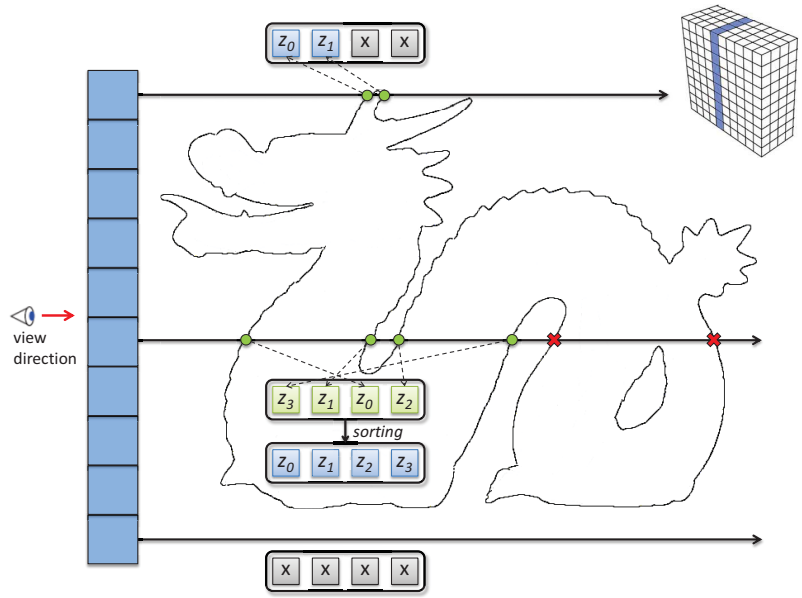


[1] [Bavoil et al, I3D'07], [2] [Mayers et al, SIGGRAPH'07], [3] [Maule et al, I3D'13], [4] [Yu et al, I3D'12]

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]
  4. unbounded memory[4]



Our contribution - k+-buffer†

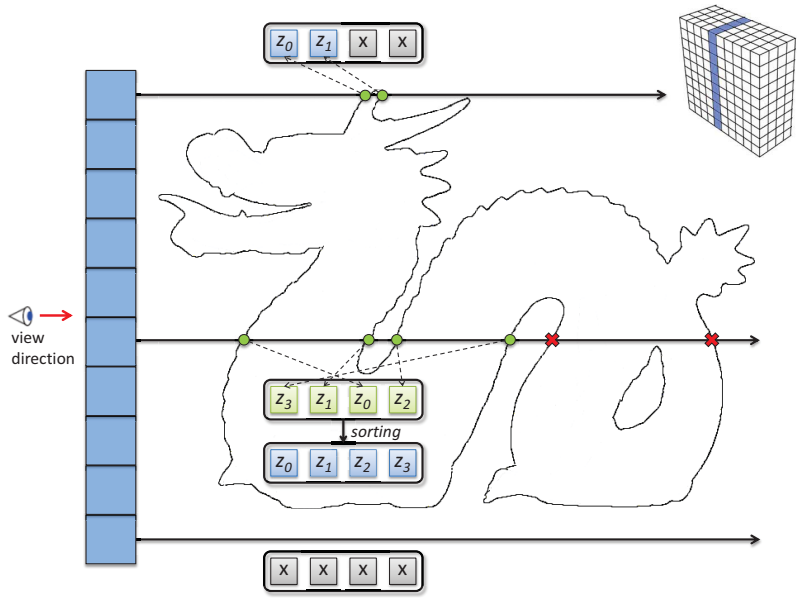
1. overcomes all limitations of existing *k*-buffer alternatives

† **A. Vasilakis** & I. Fudos, *k+-buffer: Fragment synchronized k-buffer*, Symposium on Interactive 3D Graphics and Games (I3D 14), San Francisco, USA, March, 2014.

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]
  4. unbounded memory[4]



Our contribution - k+-buffer†

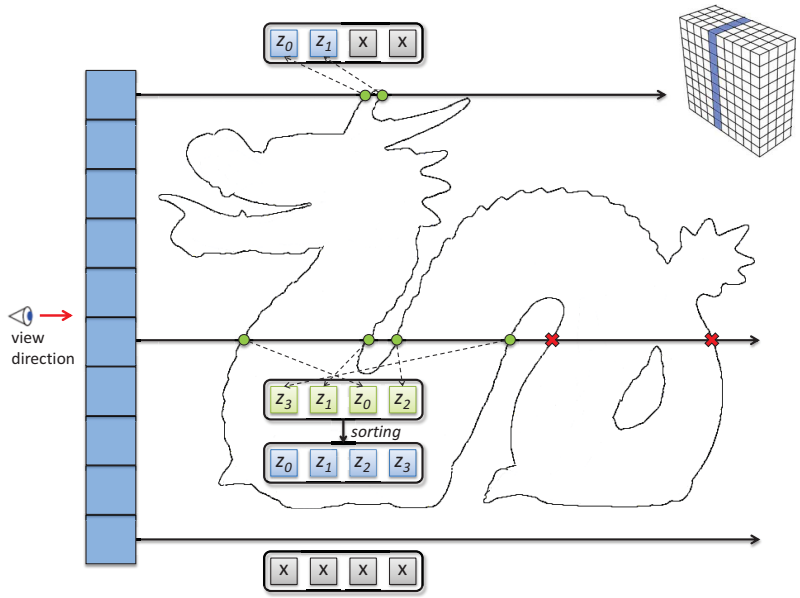
1. overcomes all limitations of existing *k*-buffer alternatives
2. memory-friendly variation (extra pass needed)

† **A. Vasilakis** & I. Fudos, *k+-buffer: Fragment synchronized k-buffer*, Symposium on Interactive 3D Graphics and Games (I3D 14), San Francisco, USA, March, 2014.

Revisiting *k*-buffer

k-buffer[1] & variants[2,3,4]

- capture the k-closest fragments
- reduce memory & sorting costs
- suffer from
  1. RMW hazards[1]
  2. geometry pre-sorting[1,2]
  3. depth precision conversion[3]
  4. unbounded memory[4]



Our contribution - k+-buffer†

1. overcomes all limitations of existing *k*-buffer alternatives
2. memory-friendly variation (extra pass needed)
3. supports Z-buffer and A-buffer functionalities

† **A. Vasilakis** & I. Fudos, *k+-buffer: Fragment synchronized k-buffer*, Symposium on Interactive 3D Graphics and Games (I3D 14), San Francisco, USA, March, 2014.

k^+-buffer Pipeline

Store & Sort solution:

1. captures fragments in an unsorted sequence

k^+-buffer Pipeline

Store & Sort solution:

1. captures fragments in an unsorted sequence
2. reorders stored fragments by their depth

Store Pass:

example

1. spin-lock strategy ([binary semaphores][1] or [pixel sync][2])
   - avoids 32-bit atomic operations

[1] [Crassin, blog'10], [2] [Salvi, SIGGRAPH'13]

k^+-buffer Pipeline

Store & Sort solution:

1. captures fragments in an unsorted sequence
2. reorders stored fragments by their depth

Store Pass:

example

1. spin-lock strategy ([binary semaphores][1] or [pixel sync][2])
   - avoids 32-bit atomic operations
2. two bounded array-based structures:
   - if $k < 16$: [max-array]-(K^+B-Array), else [max-heap]-(K^+B-Heap)

[1] [Crassin, blog'10], [2] [Salvi, SIGGRAPH'13]

k^+-buffer Pipeline

Store & Sort solution:

1. captures fragments in an unsorted sequence
2. reorders stored fragments by their depth

Store Pass:

example

1. spin-lock strategy ([binary semaphores][1] or [pixel sync][2])
   - avoids 32-bit atomic operations
2. two bounded array-based structures:
   - if $k < 16$: [max-array]-(K^+B-Array), else [max-heap]-(K^+B-Heap)
   - early-fragment culling

[1] [Crassin, blog'10], [2] [Salvi, SIGGRAPH'13]

k^+-buffer Pipeline

Store & Sort solution:

1. captures fragments in an unsorted sequence
2. reorders stored fragments by their depth

Store Pass:

example

1. spin-lock strategy ([binary semaphores][1] or [pixel sync][2])
   - avoids 32-bit atomic operations
2. two bounded array-based structures:
   - if $k < 16$: [max-array]-(K^+B-Array), else [max-heap]-(K^+B-Heap)
   - early-fragment culling

Sorting Pass:

- if $k < 16$: [insertion-sort], else [shell-sort]

[1] [Crassin, blog'10], [2] [Salvi, SIGGRAPH'13]

Extending k^+-buffer pipeline

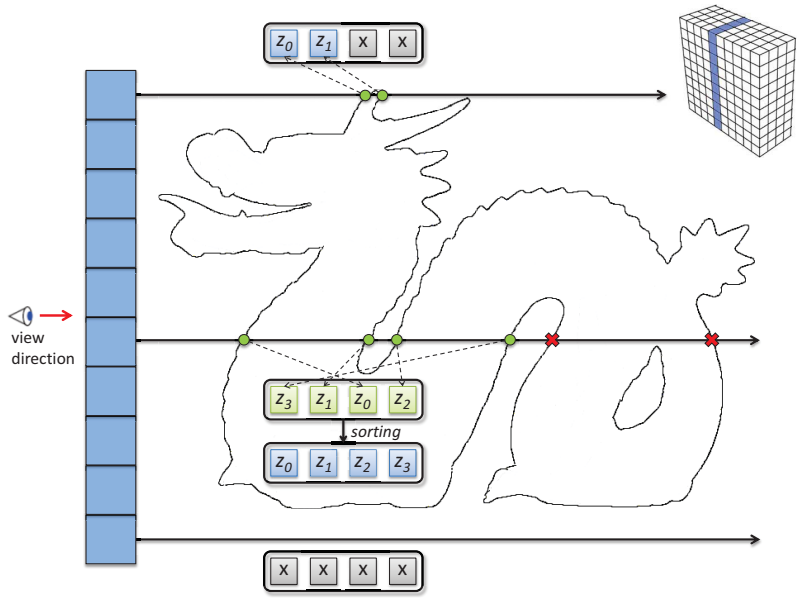
Precise memory allocation

- k is the same for all pixels

Extending k^+-buffer pipeline

Precise memory allocation

- k is the same for all pixels
- [Idea]: S-buffer[1] - (K^+B-SB)
  - counting $\longmapsto$ [hybrid scheme]
  - [condition] if k reaches n stop
  - extra pass & shared memory

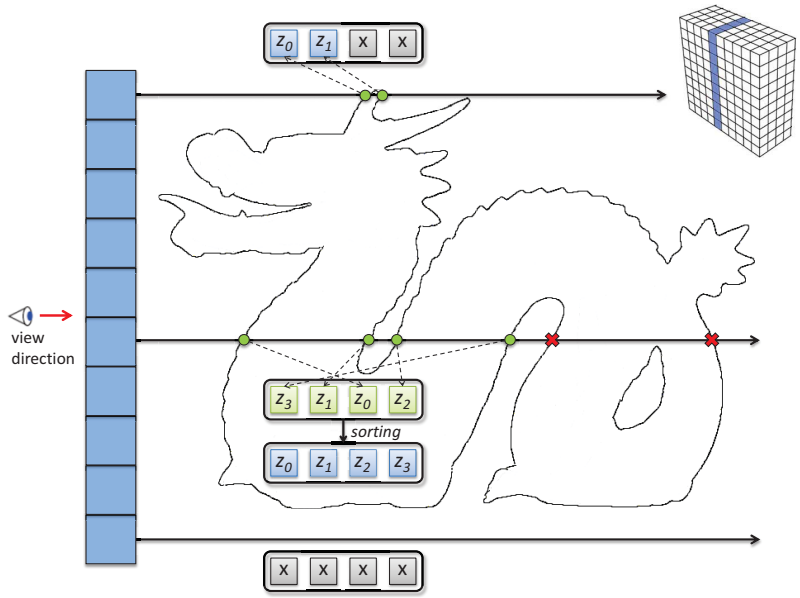


[1] [Vasilakis et al, EG'12], [2] [Liu et al, I3D'10]

Extending k^+-buffer pipeline

Precise memory allocation

- k is the same for all pixels
- [Idea]: S-buffer[1] - (K^+B-SB)
  - counting $\longmapsto$ [hybrid scheme]
  - [condition] if k reaches n stop
  - extra pass & shared memory



Unified framework

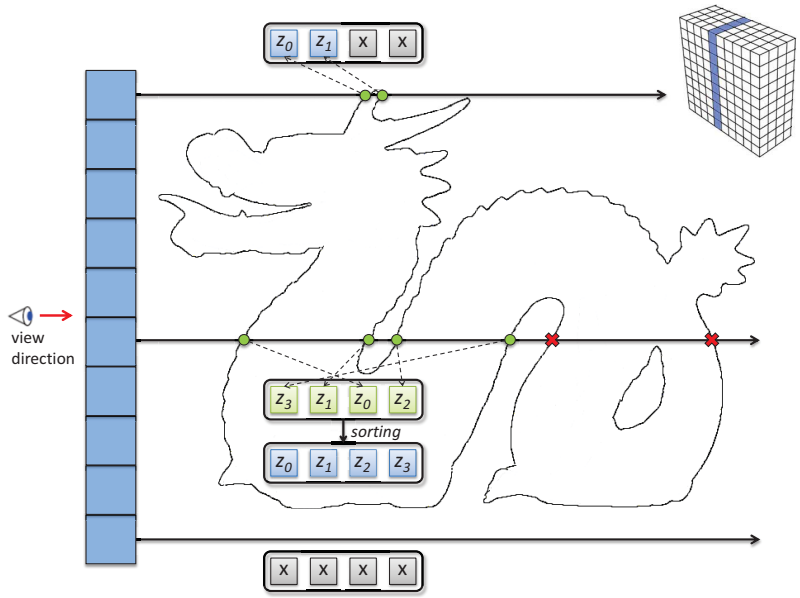
- adjust the value of k:
  - $k = 1$: [Z-buffer] behavior

[1] [Vasilakis et al, EG'12], [2] [Liu et al, I3D'10]

Extending k^+-buffer pipeline

Precise memory allocation

- k is the same for all pixels
- [Idea]: S-buffer[1] - (K^+B-SB)
  - counting $\longmapsto$ [hybrid scheme]
  - [condition] if k reaches n stop
  - extra pass & shared memory



Unified framework

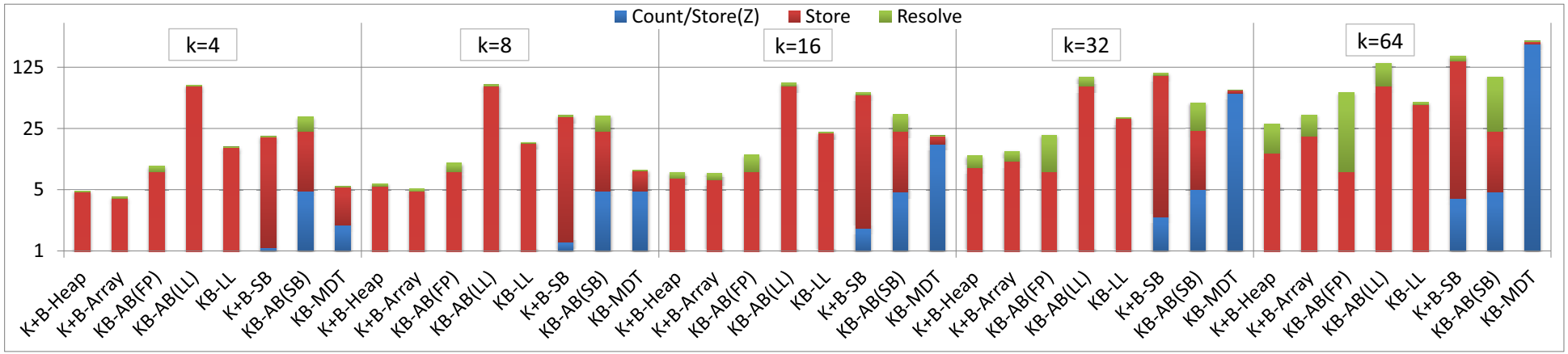
- adjust the value of k:
  - $k = 1$: [Z-buffer] behavior
  - $k = \max_p\{f(p)\}$: [A-buffer] behavior:
    - K^+B $\longmapsto$ AB_{FP}[2]
    - K^+B-SB $\longmapsto$ S-buffer[1]

[1] [Vasilakis et al, EG'12], [2] [Liu et al, I3D'10]

Performance Analysis - k-buffer

Impact of k

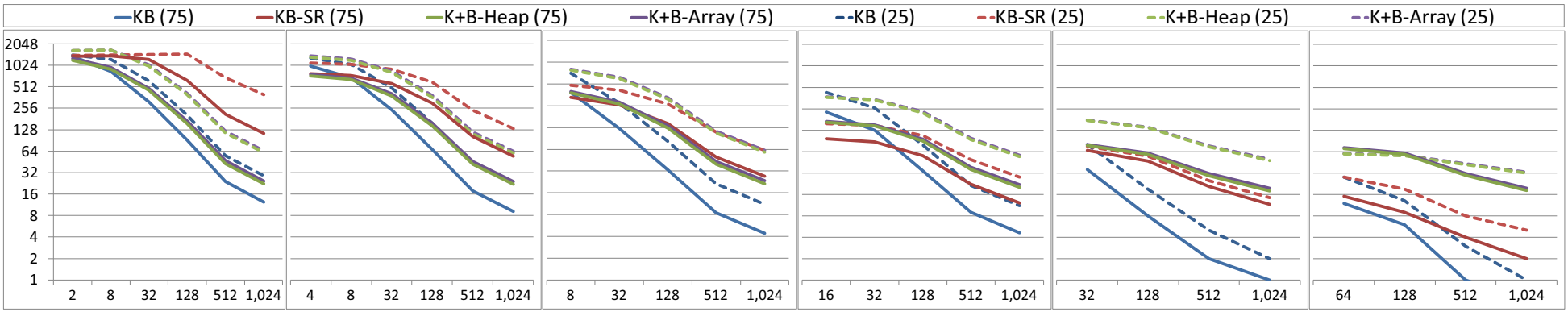
- [Scene]: $n = 128,\ k = \{4, \ldots, 64\}$
- K+B-Array [$k \downarrow$] - K+B-Heap [$k \uparrow$]
- KB-MDT (two-pass method): future 64-bits?
- KB-AB$_{FP}$: storing [$k \uparrow$] - sorting [$k \downarrow$]



Performance Analysis - k-buffer

Impact of Sorting

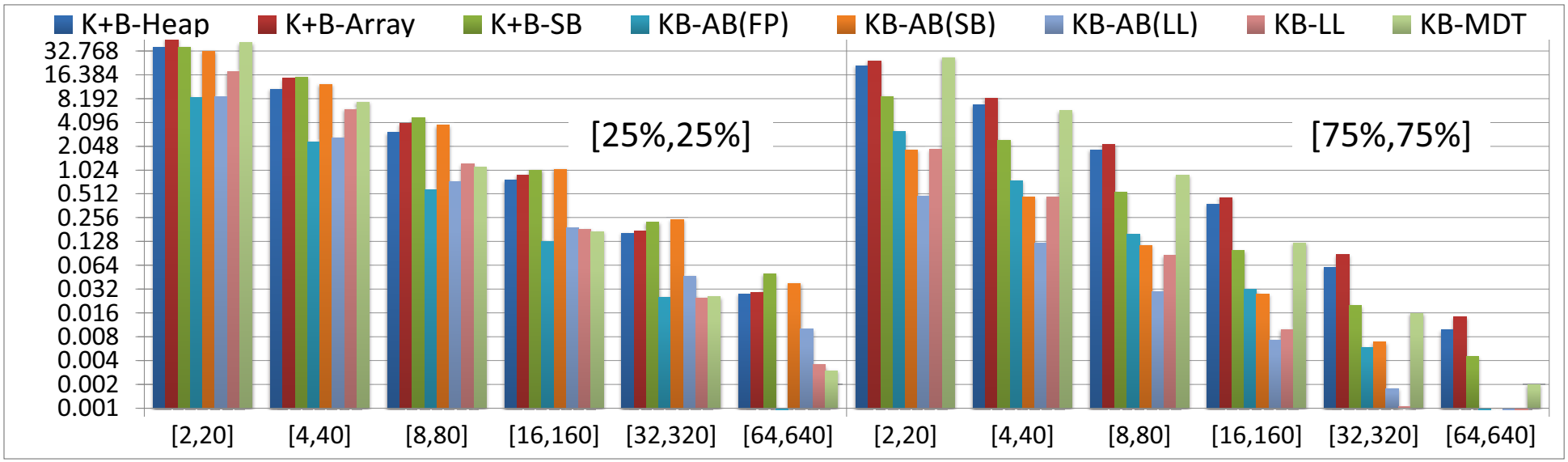
- [Scene]: $n = \{k, \ldots, 1024\}, p_d = \{25\%, 75\%\}$, (in depth order)
- K$^+$B-Array $O(1)$ > K$^+$B-Heap $O(\log_2 k)$ - (linear behavior p_d)
- $[k \uparrow]$ + multi-pass: K$^+$B > KB-SR > KB



Performance Analysis - k-buffer

Impact of Memory

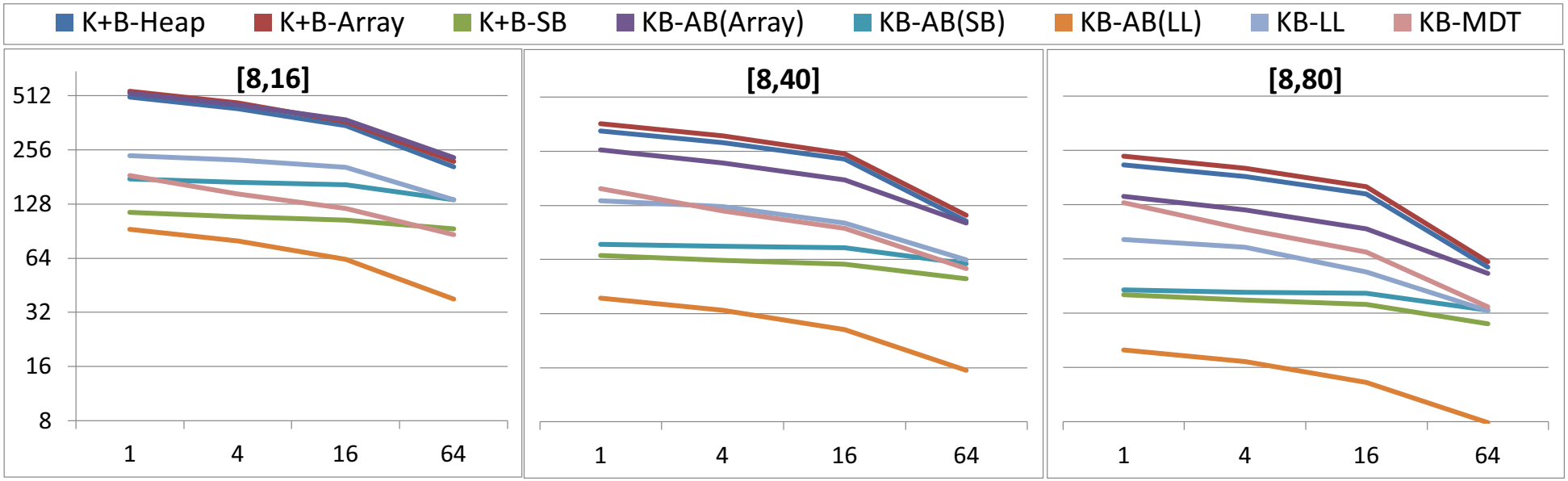
- [Scene]: $n = \{k, \ldots, 10 \cdot k\}, [p_d, f_p]$ - speed: fps/MB
- $[p_d \downarrow, f_p \downarrow]$: K$^+$B-SB $>$ K$^+$B
- $[p_d \uparrow, f_p \uparrow]$: K$^+$B-SB $<$ K$^+$B
- A-buffer-based solutions fail (overflow) when $k = 64$



Performance Analysis - k-buffer

Impact of Tessellation

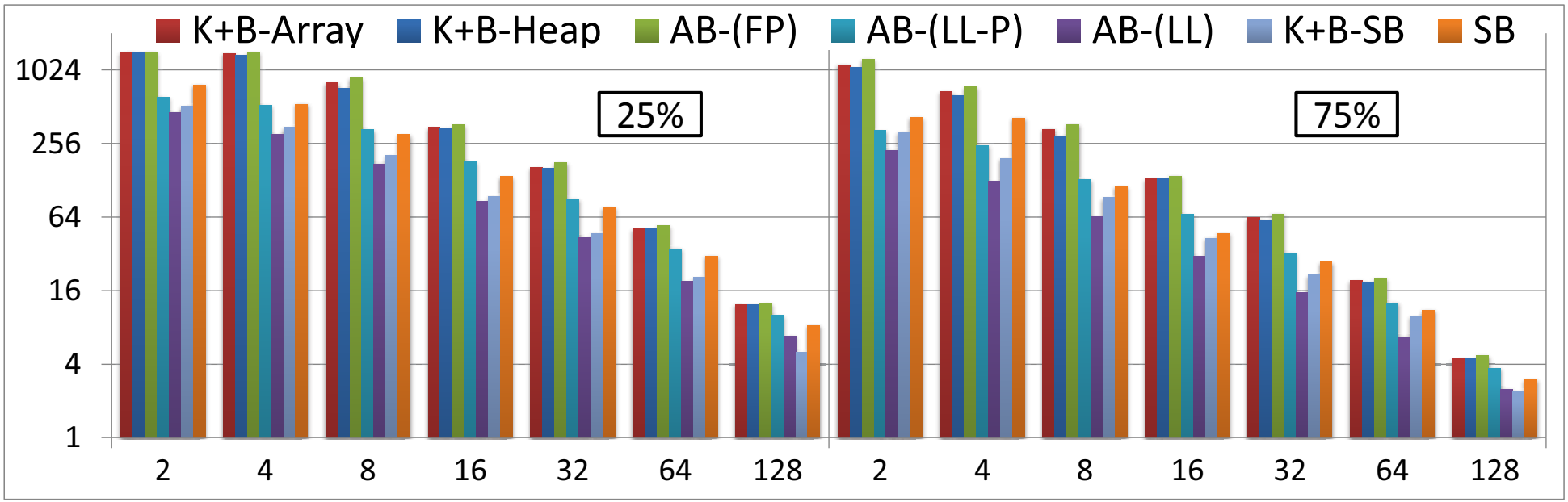
- [Scene]: level$= \{1, \ldots, 64\}, k = 8, n = \{16, 40, 80\}$
- linear behavior of K$^+$B methods
- counting geometry pass is not-tessellation dependent



Performance Analysis - A-buffer

Impact of k

- [Scene]: $k = n, p_d = \{25\%, 75\%\}$
- $AB_{FP} >$ K$^+$B-Array $>$ K$^+$B-Heap $>$ rest methods (culling mechanism)
- S-buffer $>$ K$^+$B-SB $> AB_{LL}$ (if condition at counting pass)



Memory Allocation Analysis

table

Comparison - bounded-buffers

- KB-AB$_{FP}$ needs huge resources
- K$^+$B methods require more storage (8-byte/pixel) than KB,KB-MHA
- [pixel sync] avoids semaphore allocation (4-byte)
- [data packing] employed: $\forall k > 1 : 4k > 2k + 2$
- K$^+$B vs KB-SR: $\forall k > 2 : 3k > 2k + 2$

Comparison - unbounded-buffers

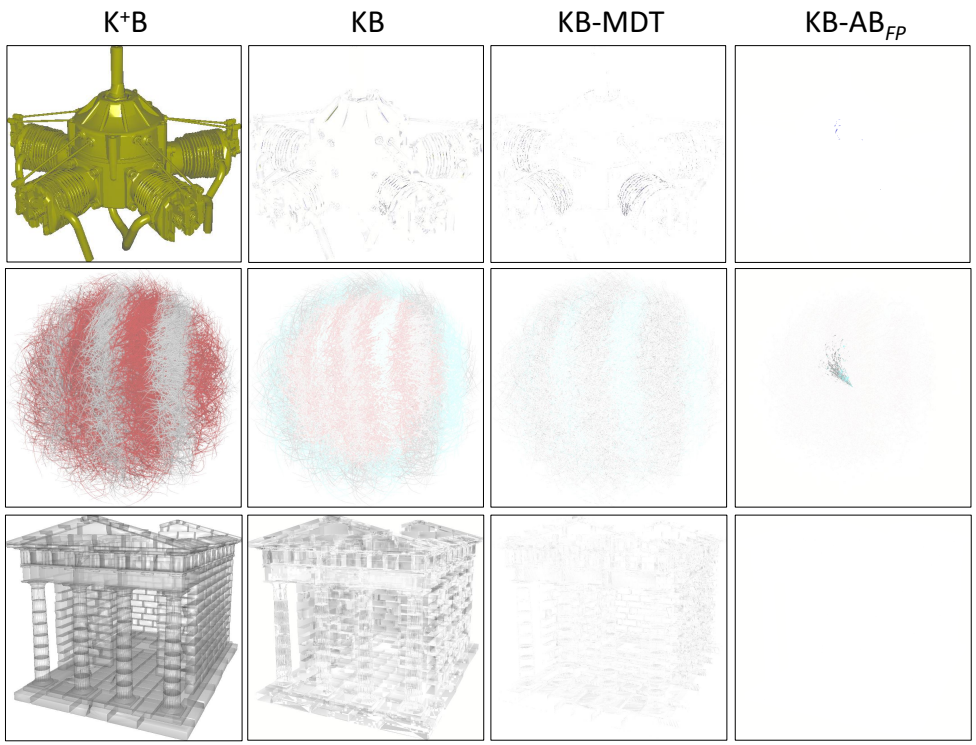
- K$^+$B-SB requires:
  1. [equal]: $(f(p) \leq k)$
  2. [less]: $(f(p) > k)$

  } KB-AB$_{LL}$,KB-LL,KB-AB$_{SB}$
- A-buffer simulation: K$^+$B = AB$_{FP}$ & K$^+$B-SB = S-buffer

Image Quality Analysis

Noticeable image differences

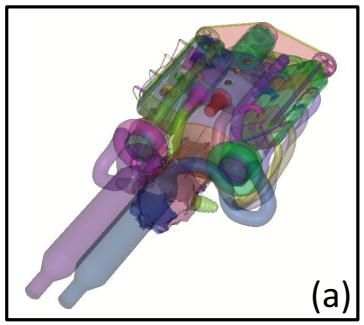
- (a) Z-buffer, (b) k-buffer, (c) A-buffer
- (left) KB: [RMW hazards], (middle) KB-MDT: [depth conversion], (right) KB-AB$_{FP}$: [avoid overflow]

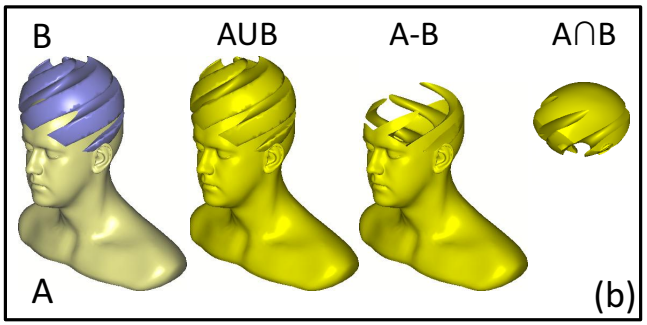


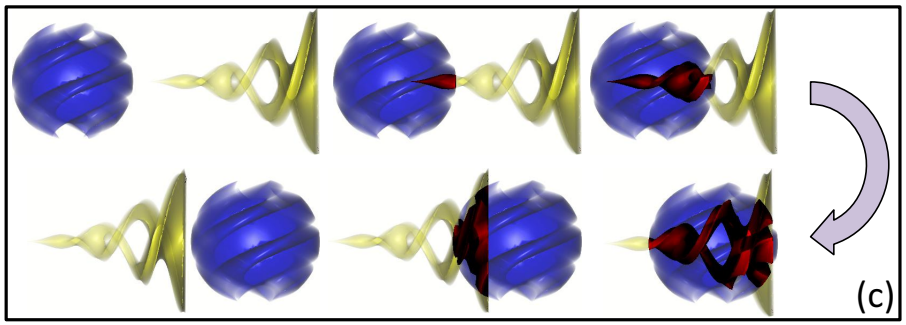
Conclusions

Bounded multi-fragment storage using k^+-buffer

- alleviates prior *k*-buffer limitations and bottlenecks by exploiting fragment culling and pixel synchronization
- introduces an extension to avoid wasteful memory consumption.
- can also simulate the behavior of Z-buffer or A-buffer







Outline

Direct Rendering of Self-Trimmed Surfaces

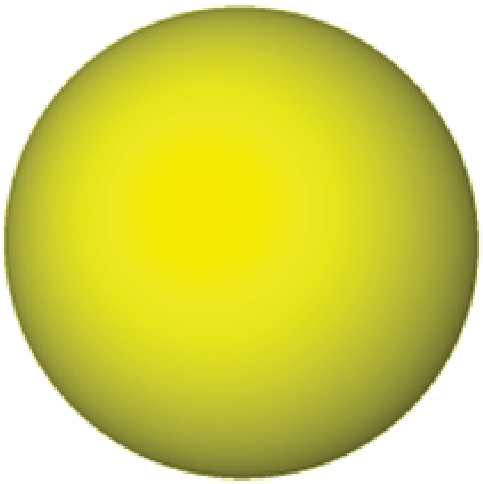
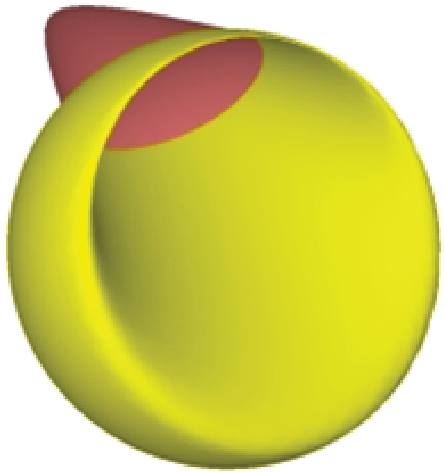
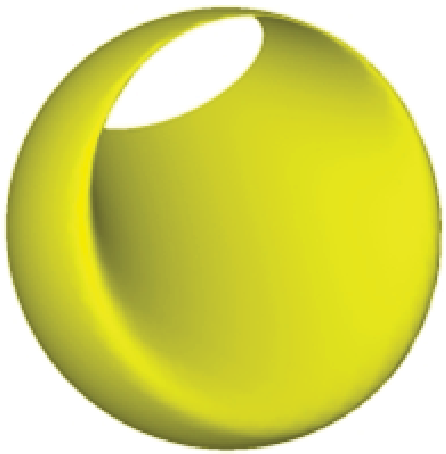
Rendering Self-Trimmed Surface (STS)

- provide interior definition (**I**) for self-crossing surface (SCS)

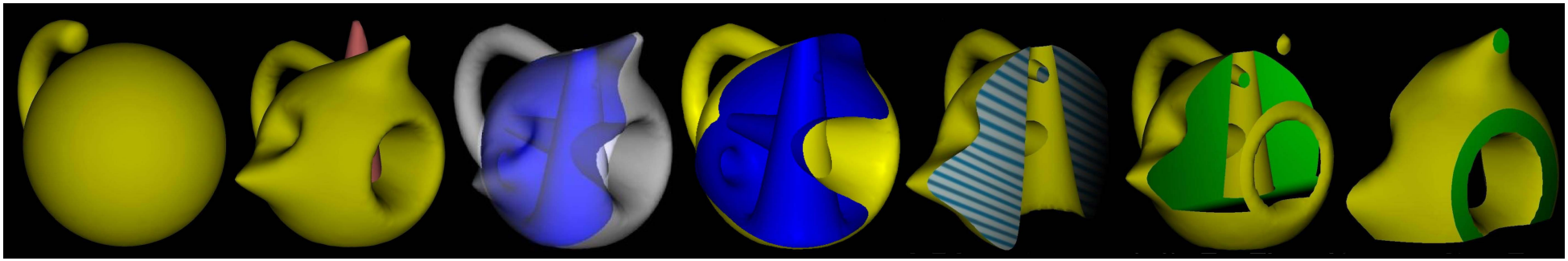
Direct Rendering of Self-Trimmed Surfaces

Rendering Self-Trimmed Surface (STS)

- provide interior definition (**I**) for self-crossing surface (SCS)
- render *trim* (boundary of **I**) of (a) [static] SCS & (b) [dynamic] SCS

Direct Rendering of Self-Trimmed Surfaces

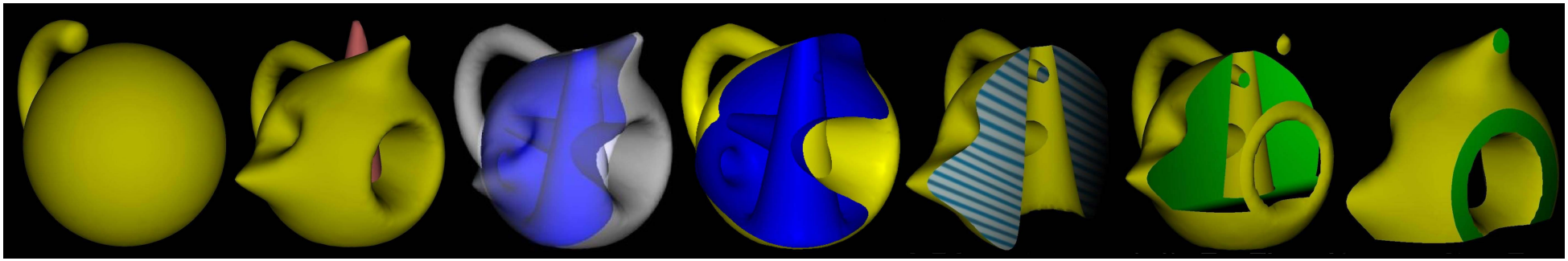


Our contribution†

1. eliminate the need for computing self-crossing curves
2. support intuitive animations & interactive editing results

† J. Rossignac & I. Fudos & **A. Vasilakis**, *Direct rendering of boolean combinations of self-trimmed surfaces*, Computer Aided Design, 45(2):288-300, 2013.

Direct Rendering of Self-Trimmed Surfaces

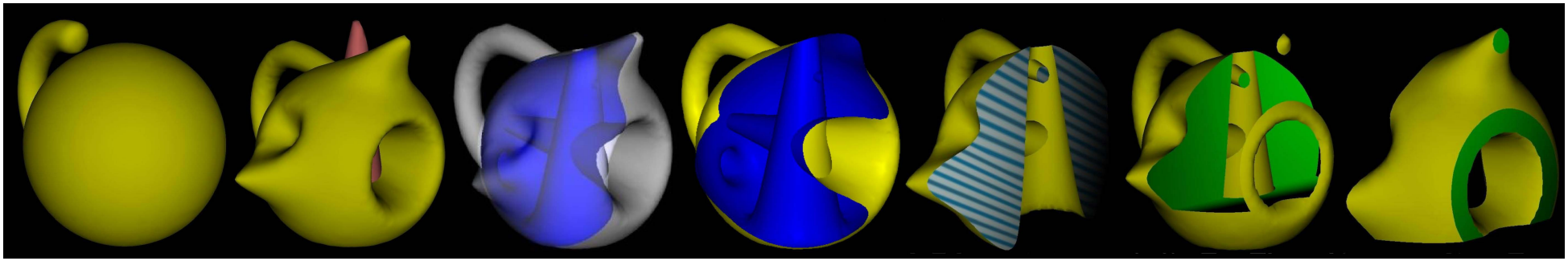


Our contribution†

1. eliminate the need for computing self-crossing curves
2. support intuitive animations & interactive editing results
3. provide real-time trimming & inspection tools for STS

† J. Rossignac & I. Fudos & **A. Vasilakis**, *Direct rendering of boolean combinations of self-trimmed surfaces*, Computer Aided Design, 45(2):288-300, 2013.

Direct Rendering of Self-Trimmed Surfaces



Our contribution†

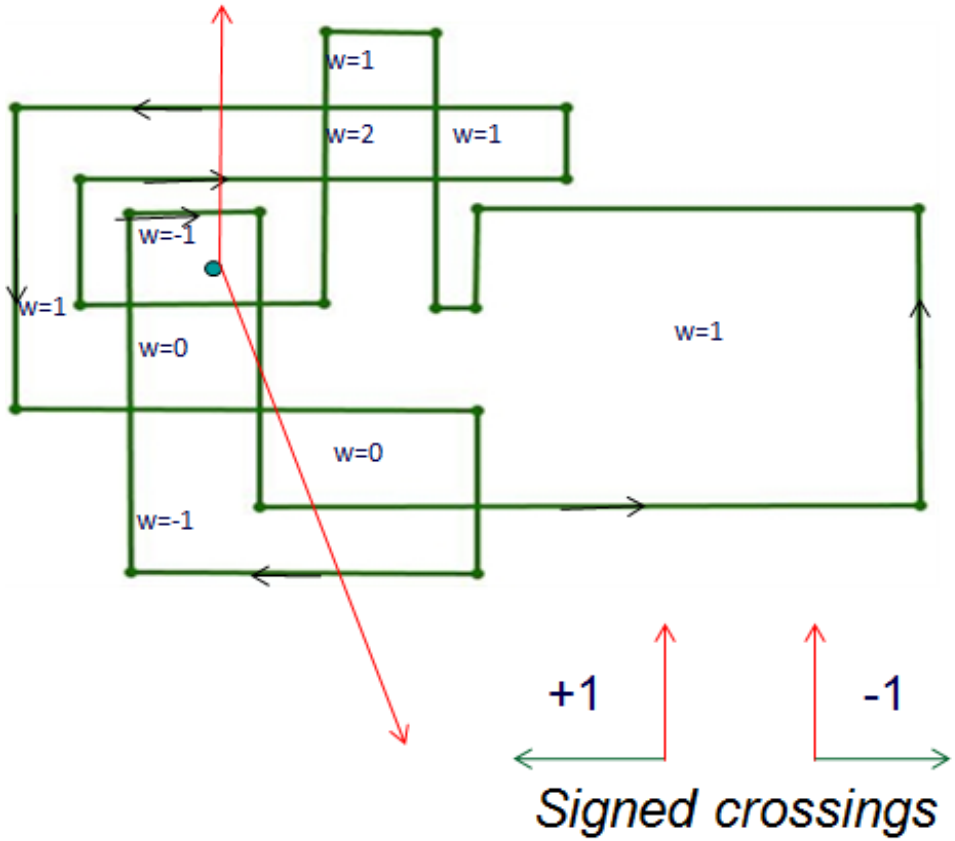
1. eliminate the need for computing self-crossing curves
2. support intuitive animations & interactive editing results
3. provide real-time trimming & inspection tools for STS
4. treat STS as valid primitives in CAD, including CSG

† J. Rossignac & I. Fudos & **A. Vasilakis**, *Direct rendering of boolean combinations of self-trimmed surfaces*, Computer Aided Design, 45(2):288-300, 2013.

Winding Number - 2D

Computation:

- count of signed crossings ($+1$ or -1) of ray to point outside
- independent of the choice ray

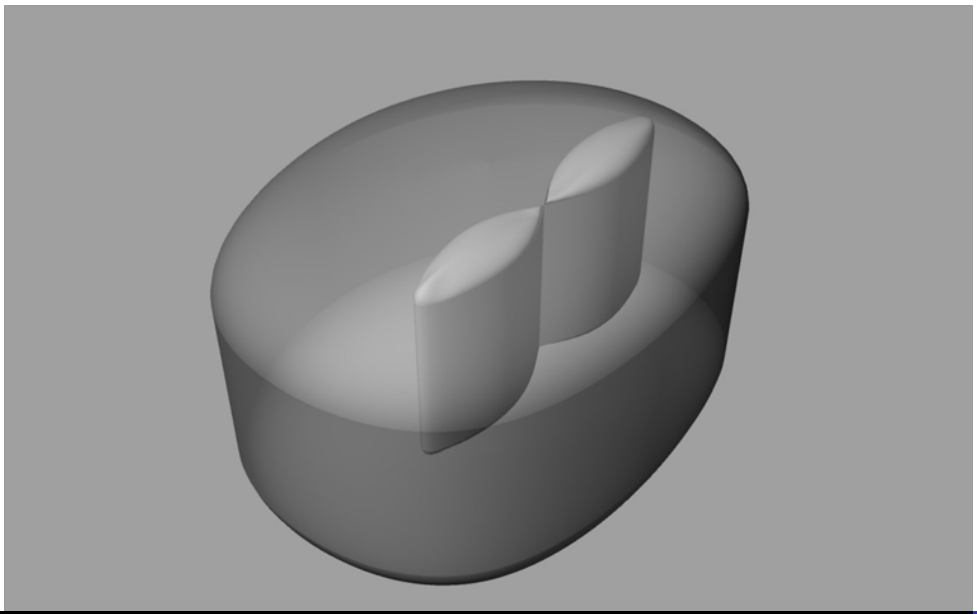


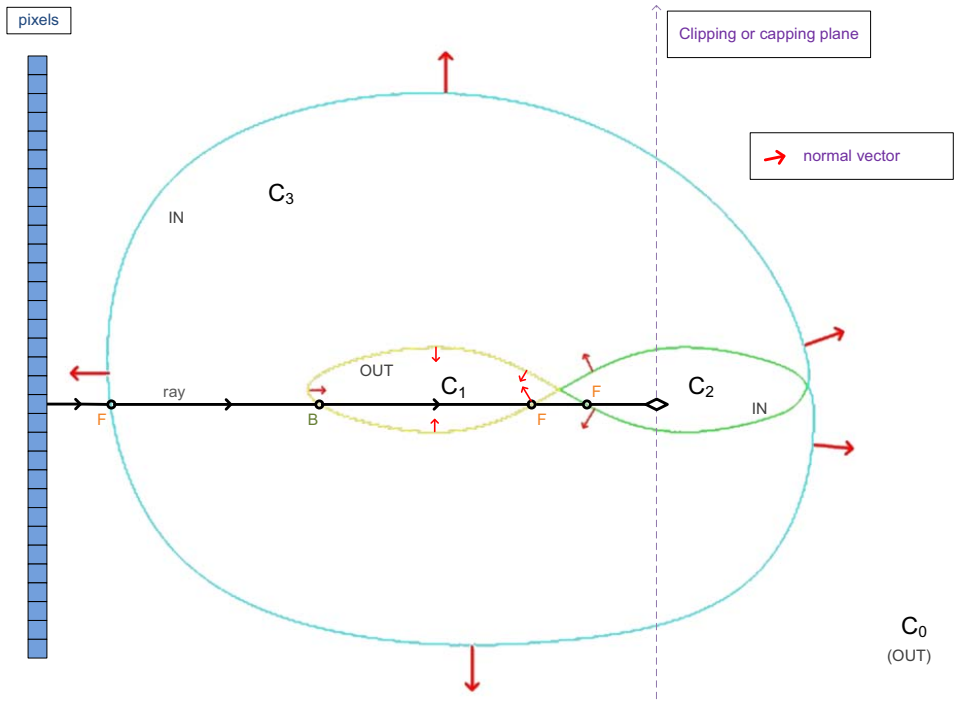
Winding Number - 3D

Computation:

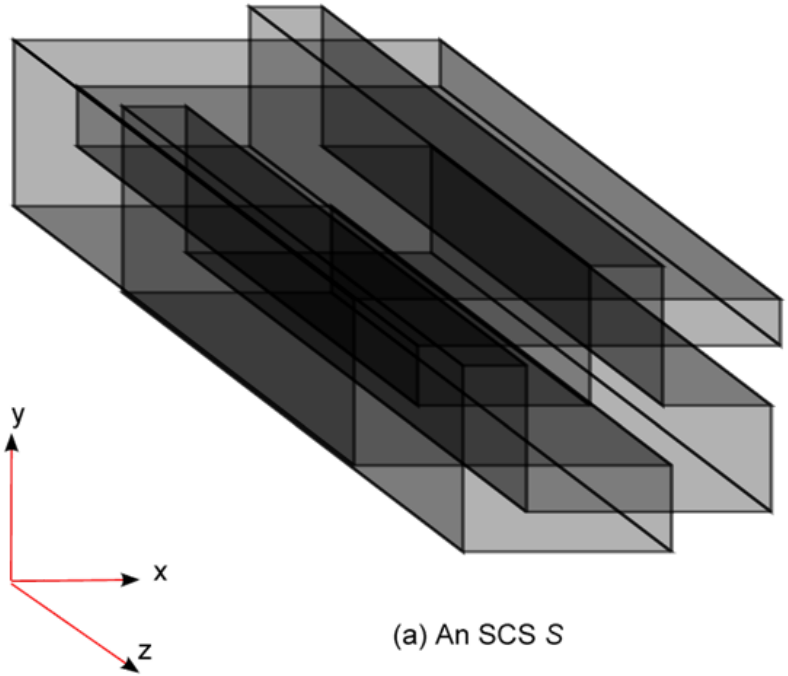
- Surface S, Pixel P, Fragment f, point p
  1. $k_i = \#$ P enters S (f.normal [↑])
  2. $k_o = \#$ P exits S (f.normal [↓])

$$\left.\begin{array}{l} \\ \\ \end{array}\right\} w(p,S) = k_i - k_o$$

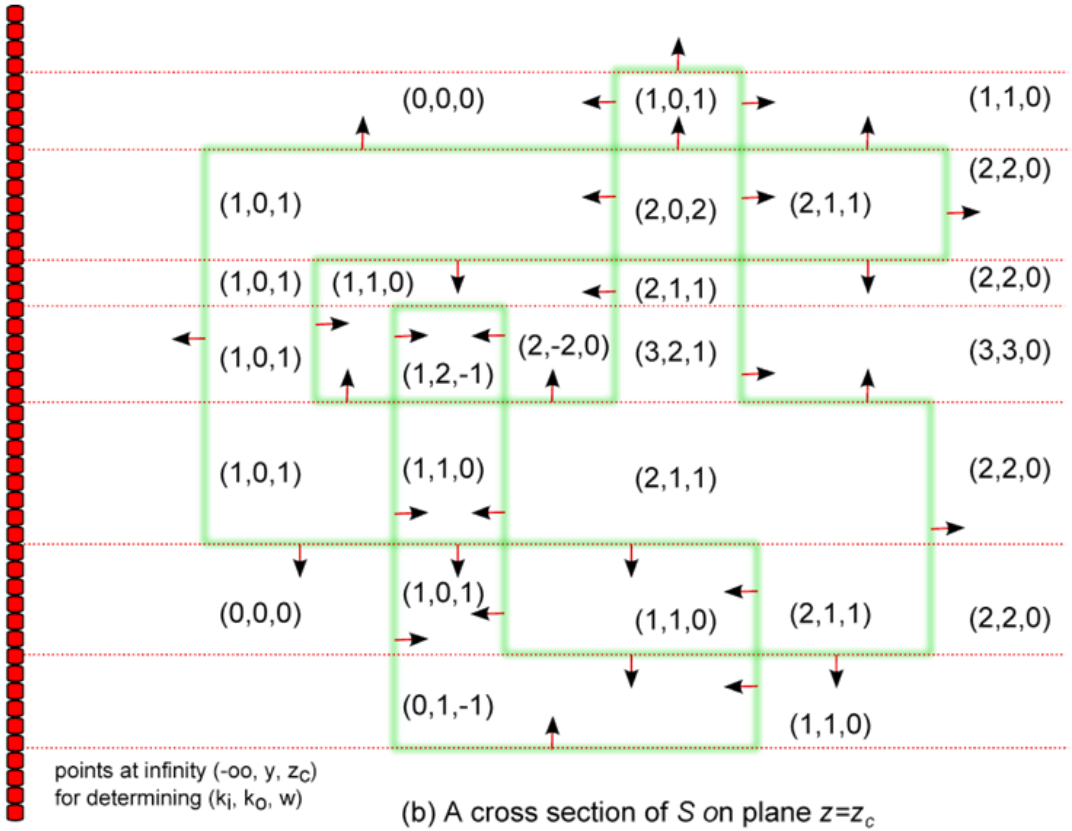




Winding Number - 3D Example



(a) An SCS S



(b) A cross section of S on plane $z=z_c$

Static Rules

Given the winding number w:

Static Rules

Given the winding number w:

1. [positive index]: $w > 0$
   - not orientation invariant: fail if we reverse orientation

Static Rules

Given the winding number w:

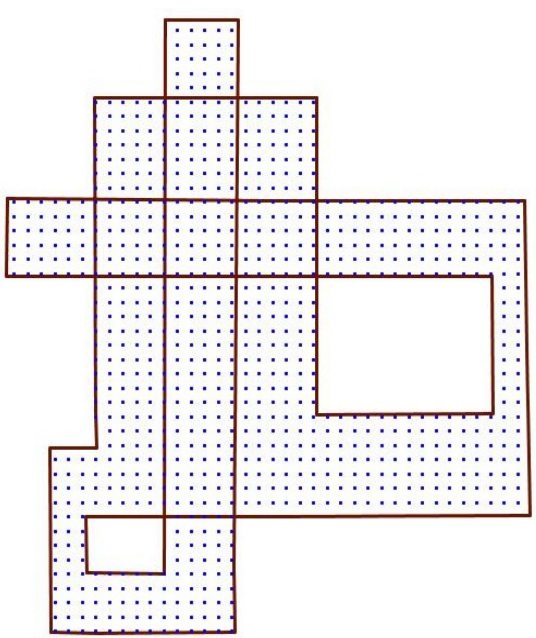
1. [positive index]: $w > 0$
   - not orientation invariant: fail if we reverse orientation
2. [parity] or [alternating component]: $w\%2 = 1$ (or w is odd)
   - $T(S) = S$: does not trim the surface at all
   - produces non-manifolds solids

Static Rules

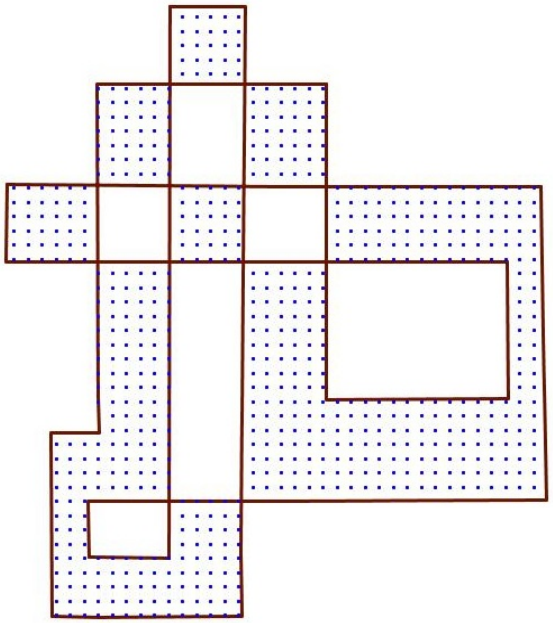
Given the winding number w:

1. [positive index]: $w > 0$
   - not orientation invariant: fail if we reverse orientation
2. [parity] or [alternating component]: $w\%2 = 1$ (or w is odd)
   - $T(S) = S$: does not trim the surface at all
   - produces non-manifolds solids
3. [alternating border]: $\lceil \frac{w}{2} \rceil \%2 = 1$
   - does not produce non-manifold solids
   - obtain complement of solid $\longmapsto$ add two surrounding boxes $(w + 2)$
   - reverse the surface orientation we obtain the complement of the trim

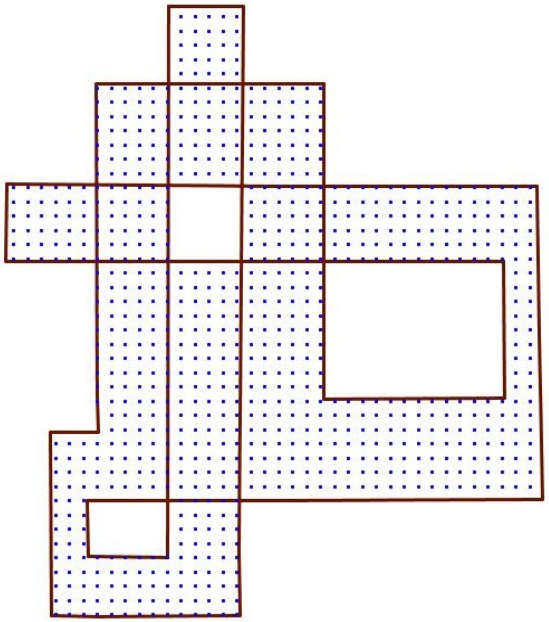
Static rules fail to capture well deformations...



(a)

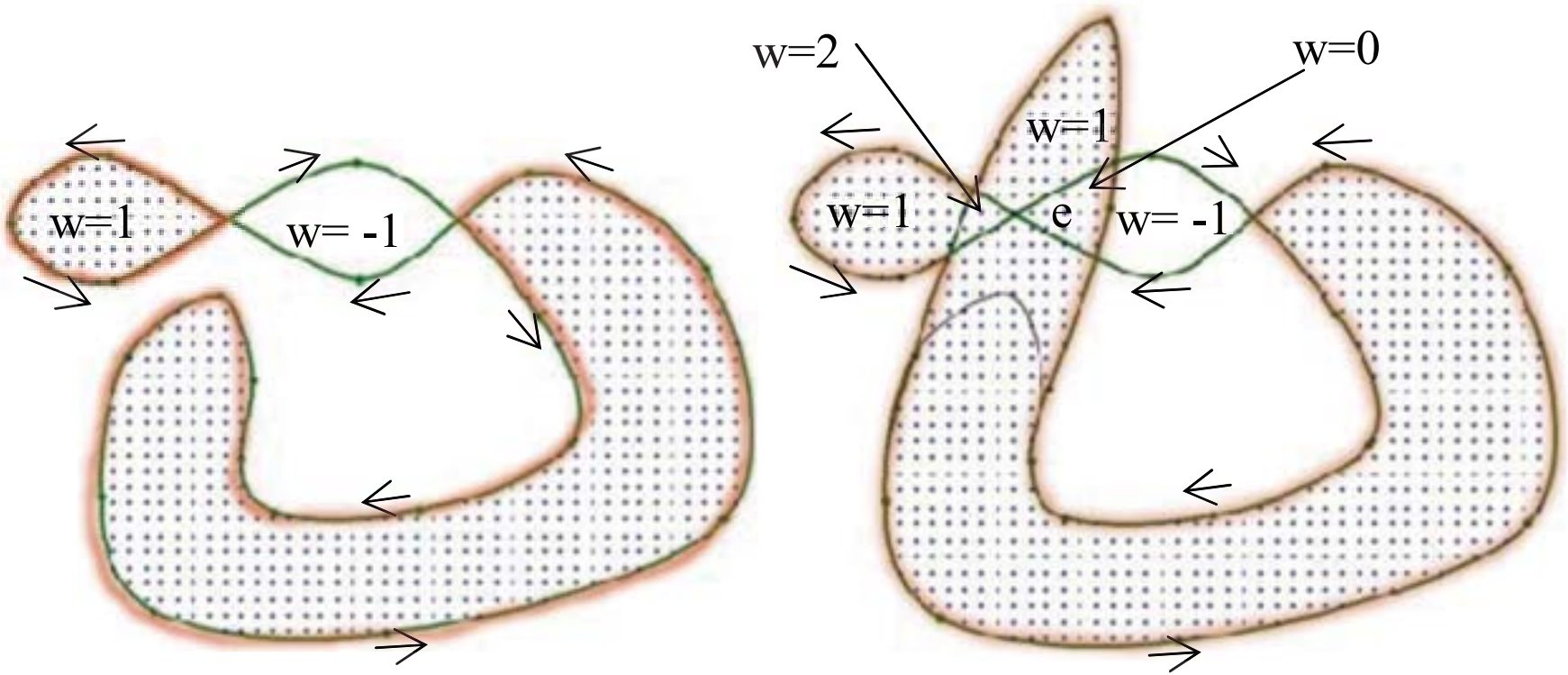


(b)

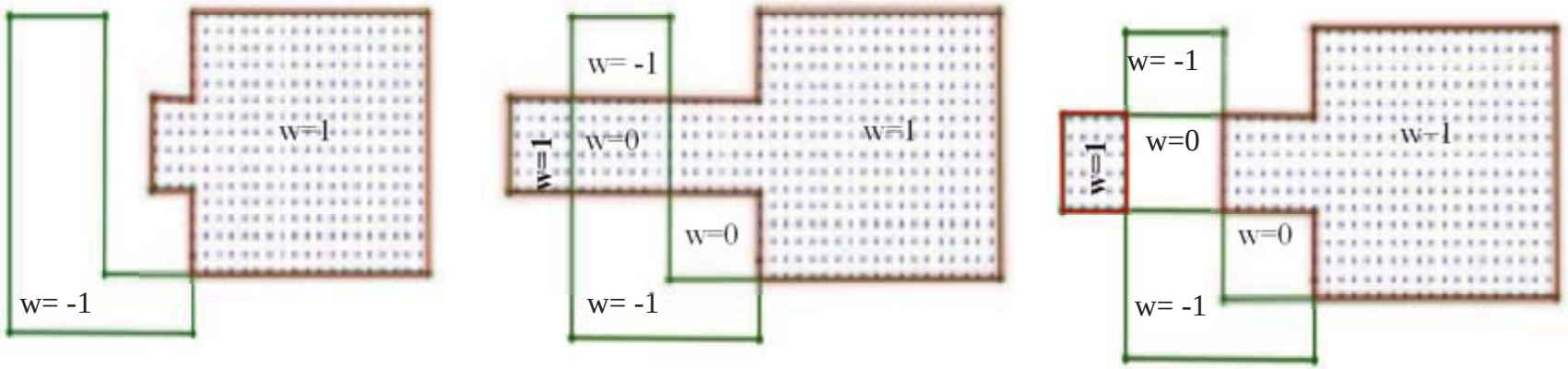


(c)

Static rules fail to capture well deformations...



Static rules fail to capture well deformations...



Dynamic Rules

Modeling Deformations for Dynamic Rules

- need to maintain history from reference S to deformed pose S'
- k disjoint deformations: a boundary surface may cross a point only once

Dynamic Rules

Modeling Deformations for Dynamic Rules

- need to maintain history from reference S to deformed pose S'
- k disjoint deformations: a boundary surface may cross a point only once

1. [constructive]: $i(p, S') = i(p, S)$ **op** $(w(p, S) \neq w(p, S'))$
   - initial classification of S_0: any static rule
   - design intent & has a constructive nature
   - ill-defined results: trim is not part of initial surface

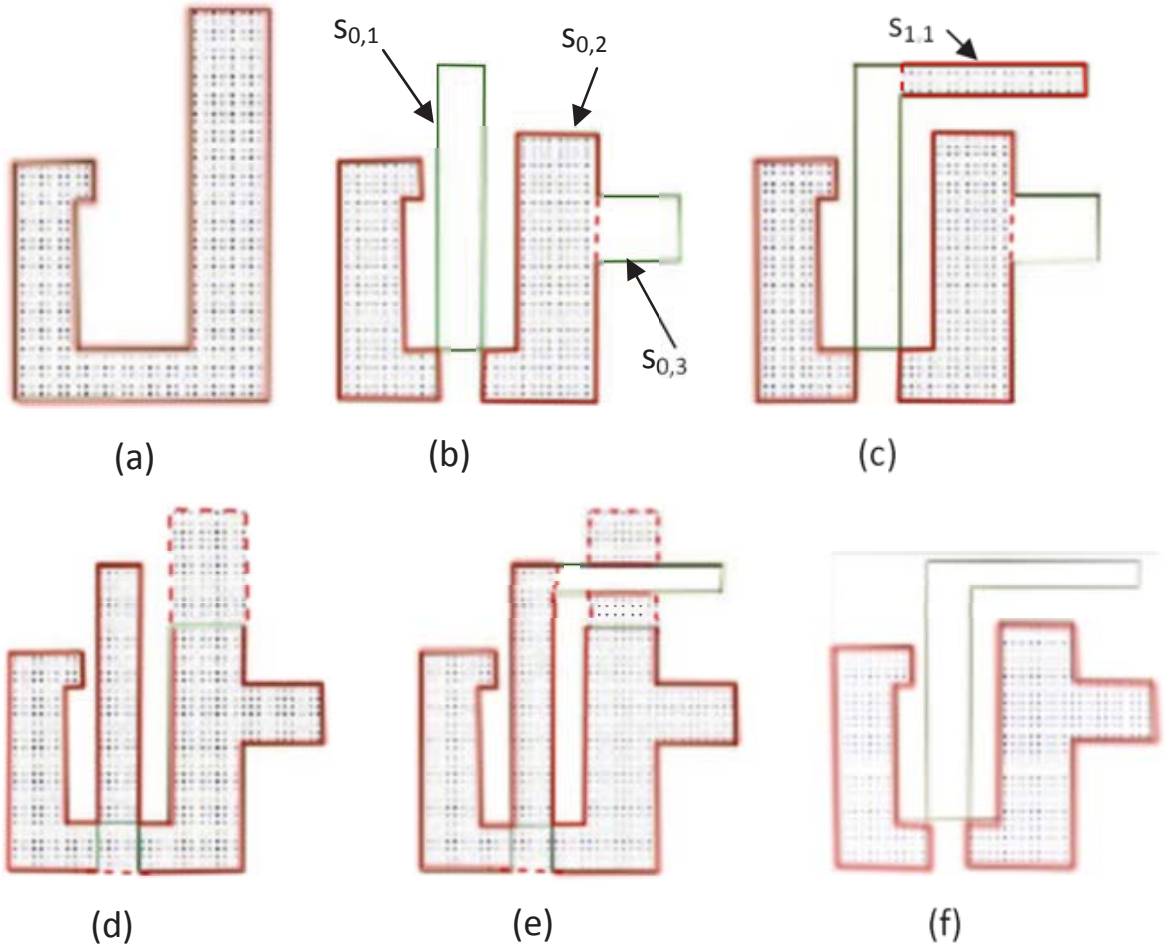
Dynamic Rules

Modeling Deformations for Dynamic Rules

- need to maintain history from reference S to deformed pose S'
- k disjoint deformations: a boundary surface may cross a point only once

1. [constructive]: $i(p, S') = i(p, S)$ **op** $(w(p, S) \neq w(p, S'))$
   - initial classification of S_0: any static rule
   - design intent & has a constructive nature
   - ill-defined results: trim is not part of initial surface
2. [confluent deformation]: $i(p, S') = \begin{cases} i(p, S), \; w(p, S) = w(p, S') \\ \left\lceil \frac{w(p,S')-w(p,S)}{2} \right\rceil \%2, \; otherwise \end{cases}$
   - prohibit deformations of the trimmed surface parts

Dynamic Rules - Example



Rendering & Trimming Algorithms for Static Rules

Pipeline

1. [classification]: compute p.index $\longmapsto$ p.class

Rendering & Trimming Algorithms for Static Rules

Pipeline

1. [classification]: compute p.index $\longmapsto$ p.class
2. [trimming]: find f with different in/out classifications

Rendering & Trimming Algorithms for Static Rules

Pipeline

1. [classification]: compute p.index $\longmapsto$ p.class
2. [trimming]: find f with different in/out classifications
3. [clipping]: start count/render after C.depth

Rendering & Trimming Algorithms for Static Rules

Pipeline

1. [classification]: compute p.index $\longmapsto$ p.class
2. [trimming]: find f with different in/out classifications
3. [clipping]: start count/render after C.depth

Multi-fragment Rendering

1. multi-pass depth peeling
2. buffer-based methods

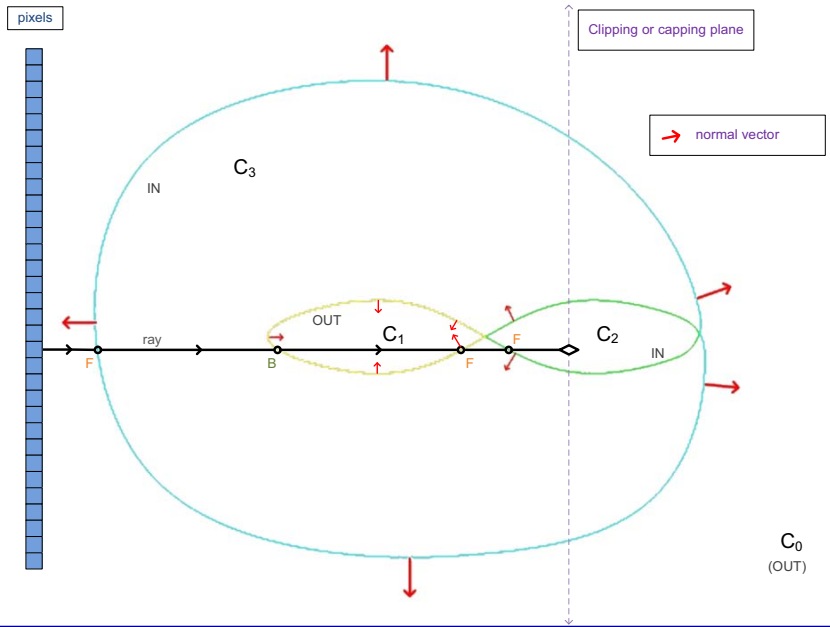
Rendering & Trimming Algorithms for Static Rules

Pipeline

1. [classification]: compute p.index $\longmapsto$ p.class
2. [trimming]: find f with different in/out classifications
3. [clipping]: start count/render after C.depth

Multi-fragment Rendering

1. multi-pass depth peeling
2. buffer-based methods
3. sort-independent methods[1]



[1] [Meshkin, GDC'07]

Rendering & Trimming Algorithms for Dynamic Rules

Pipeline

1. [classification]: dynamic p.class computation requires:
   - [reference] p^r: p^r.index, p^r.class (64-bit vector BC^r)
   - [current] p_c: p^c.index
2. [trimming]: find f with different in/out classifications

shader

Rendering & Trimming Algorithms for Dynamic Rules

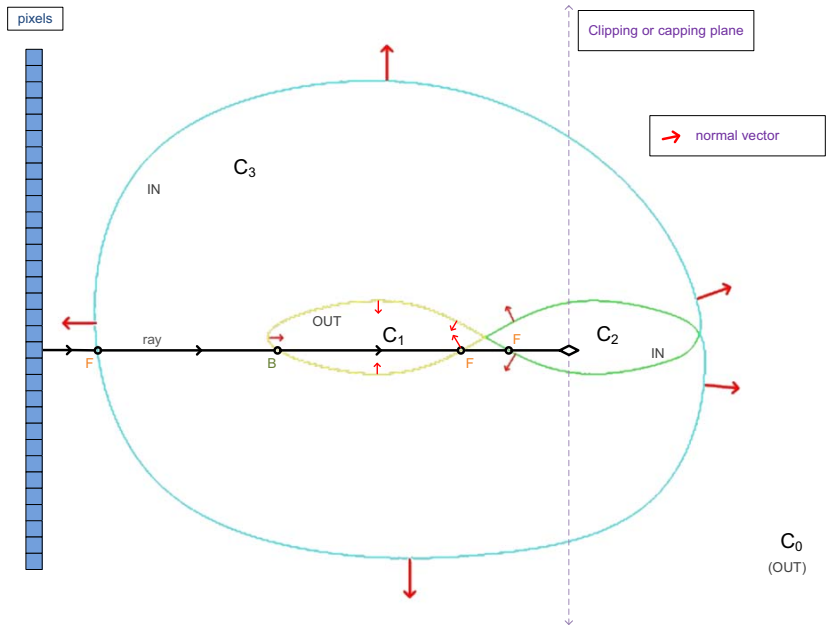
Pipeline

1. [classification]: dynamic p.class computation requires:
   - [reference] p^r: p^r.index, p^r.class (64-bit vector BC^r)
   - [current] p_c: p^c.index
2. [trimming]: find f with different in/out classifications

shader

Frame characterization

- [reference]: *frame*.class $= 0$
- [current]: *frame*.class $= 1$
- [both]: *frame*.class $= 2$



Extend Dynamic Trimming for CSG/Capping

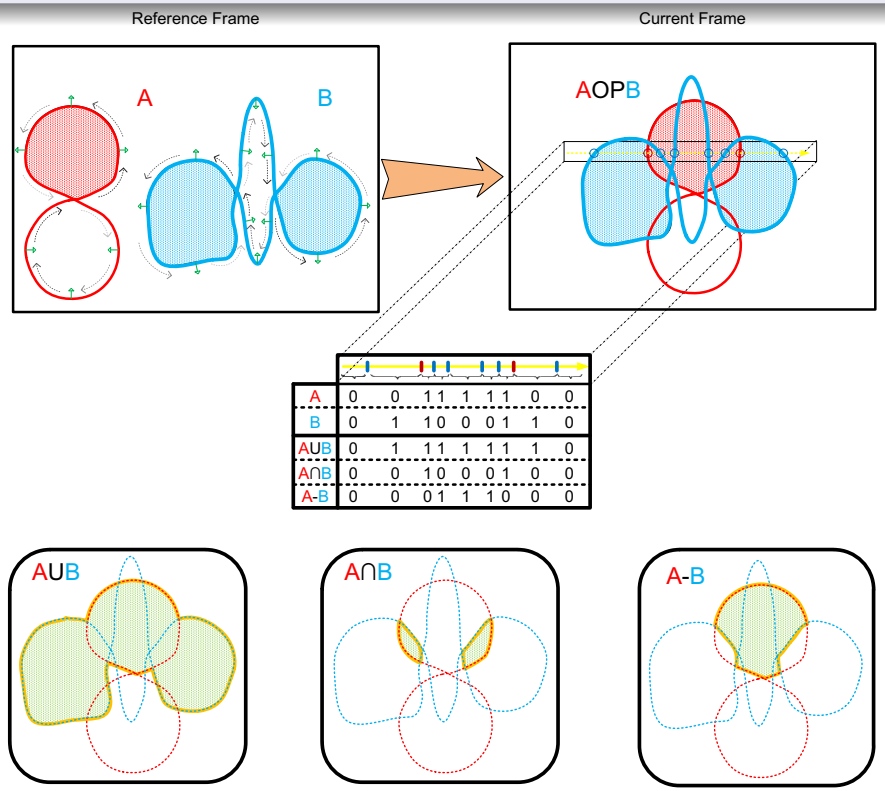
shader

- [solid A]: frame.*class* $= 1$ - [solid B]: frame.*class* $= 2$
- store/csg operate two classification vectors: $\mathrm{BC}^r_A[\mathrm{i}]$ *op* $\mathrm{BC}^r_B[\mathrm{j}]$

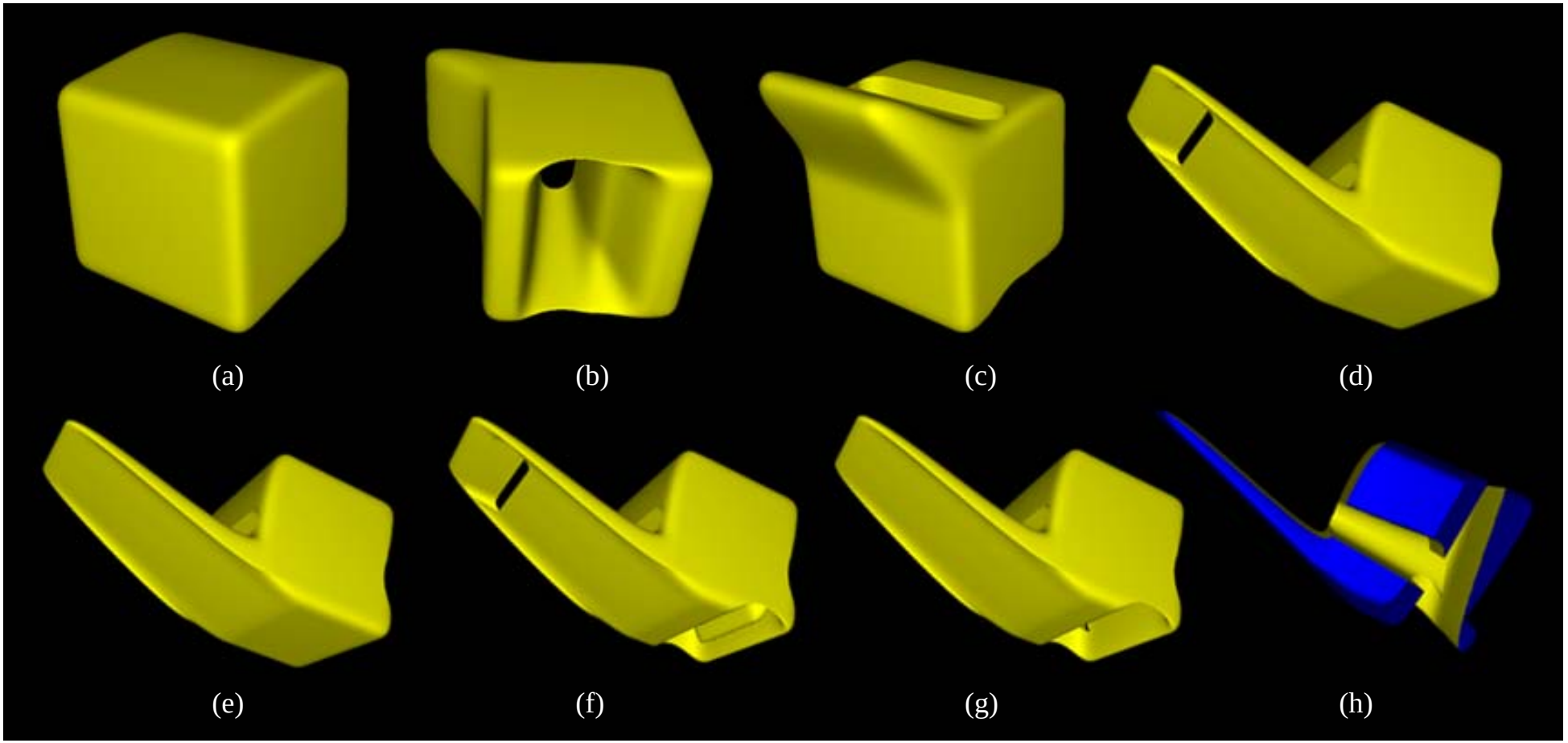
Extend Dynamic Trimming for CSG/Capping

shader

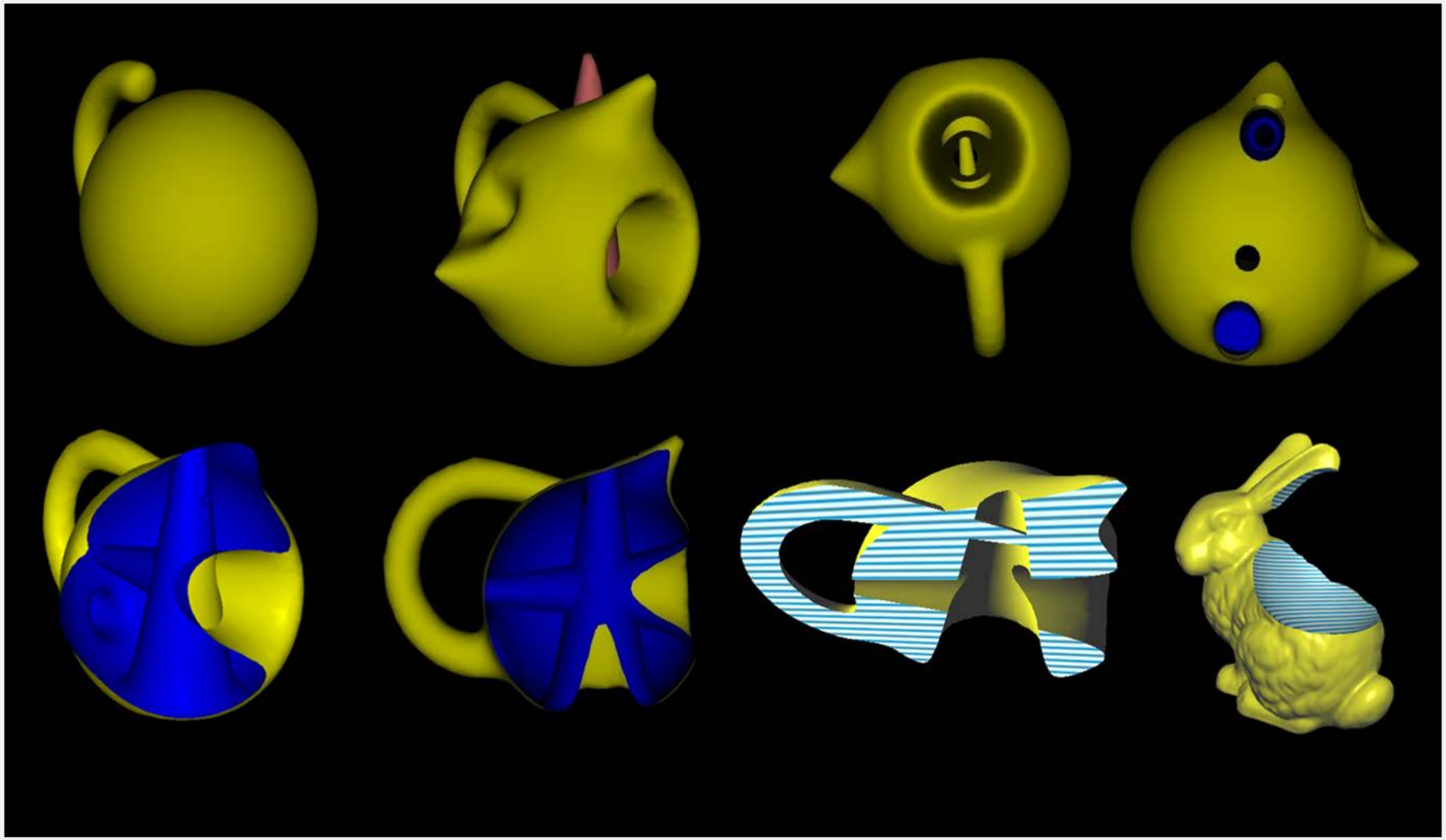
- [solid A]: frame.*class* = 1 - [solid B]: frame.*class* = 2
- store/csg operate two classification vectors: $BC^r_A[i]$ *op* $BC^r_B[j]$
- [capping]: subtracting a capping box from our surface



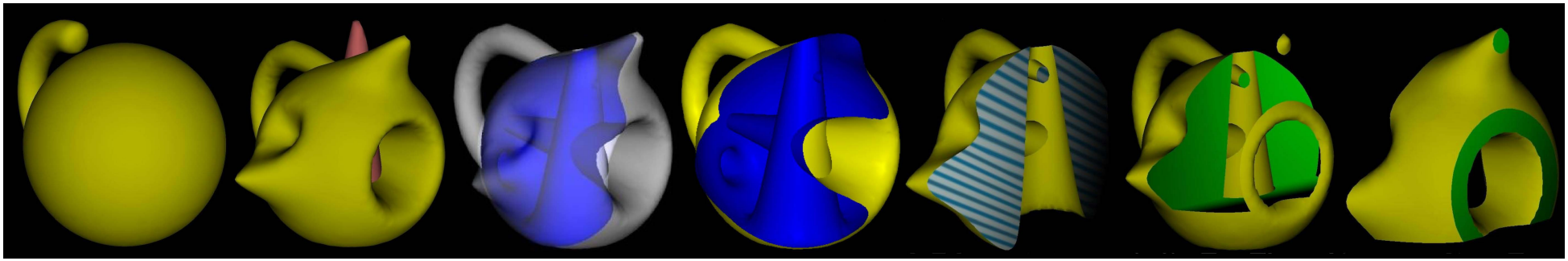
Static & Dynamic Trimming



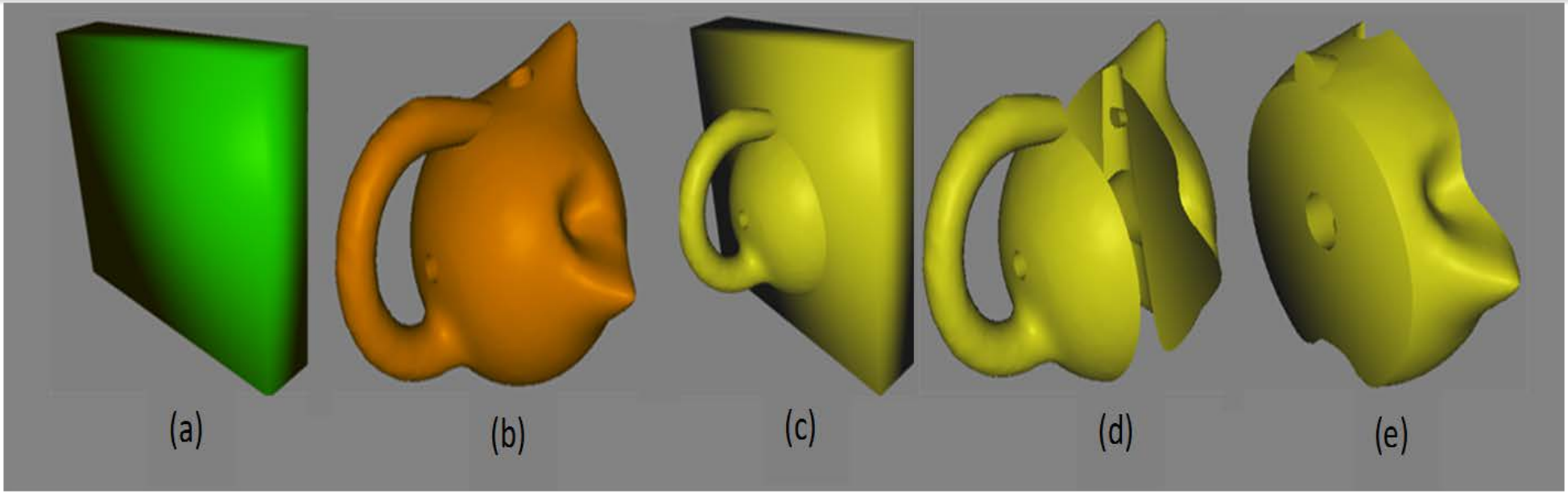
Clipping & Capping



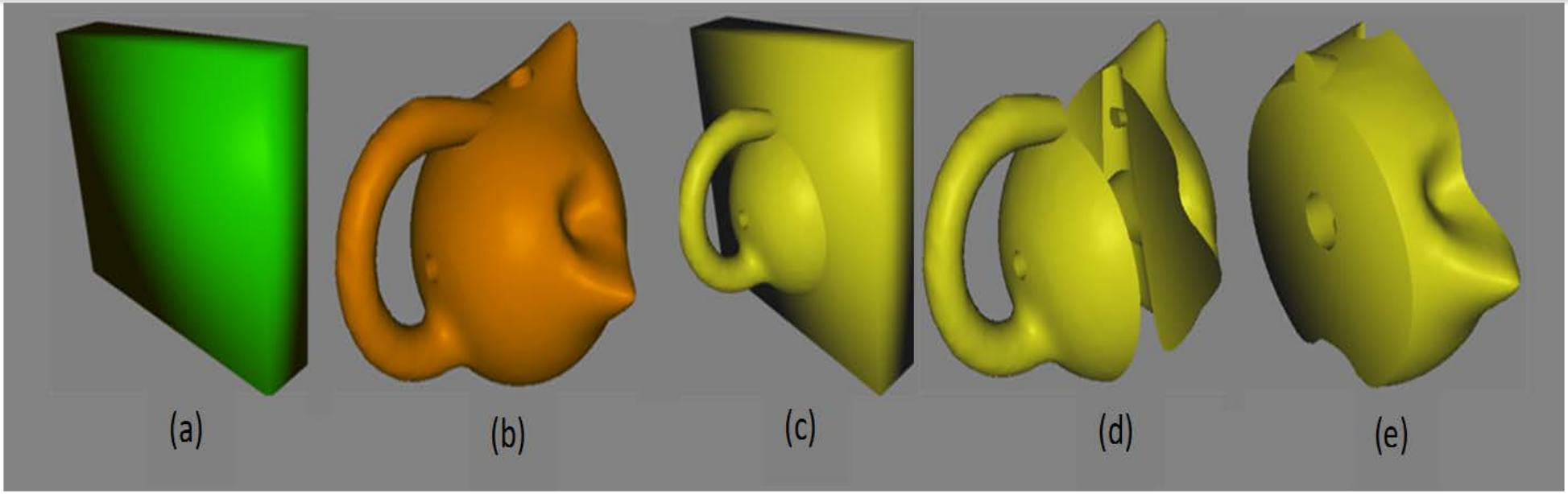
Rendering & Transparency & Clipping & Capping & CSG

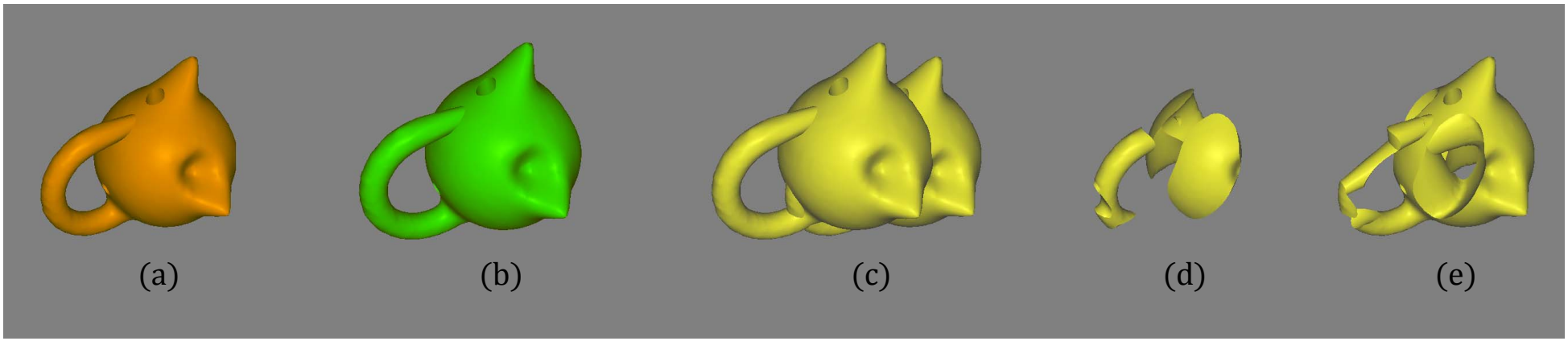


Constructive Solid Geometry

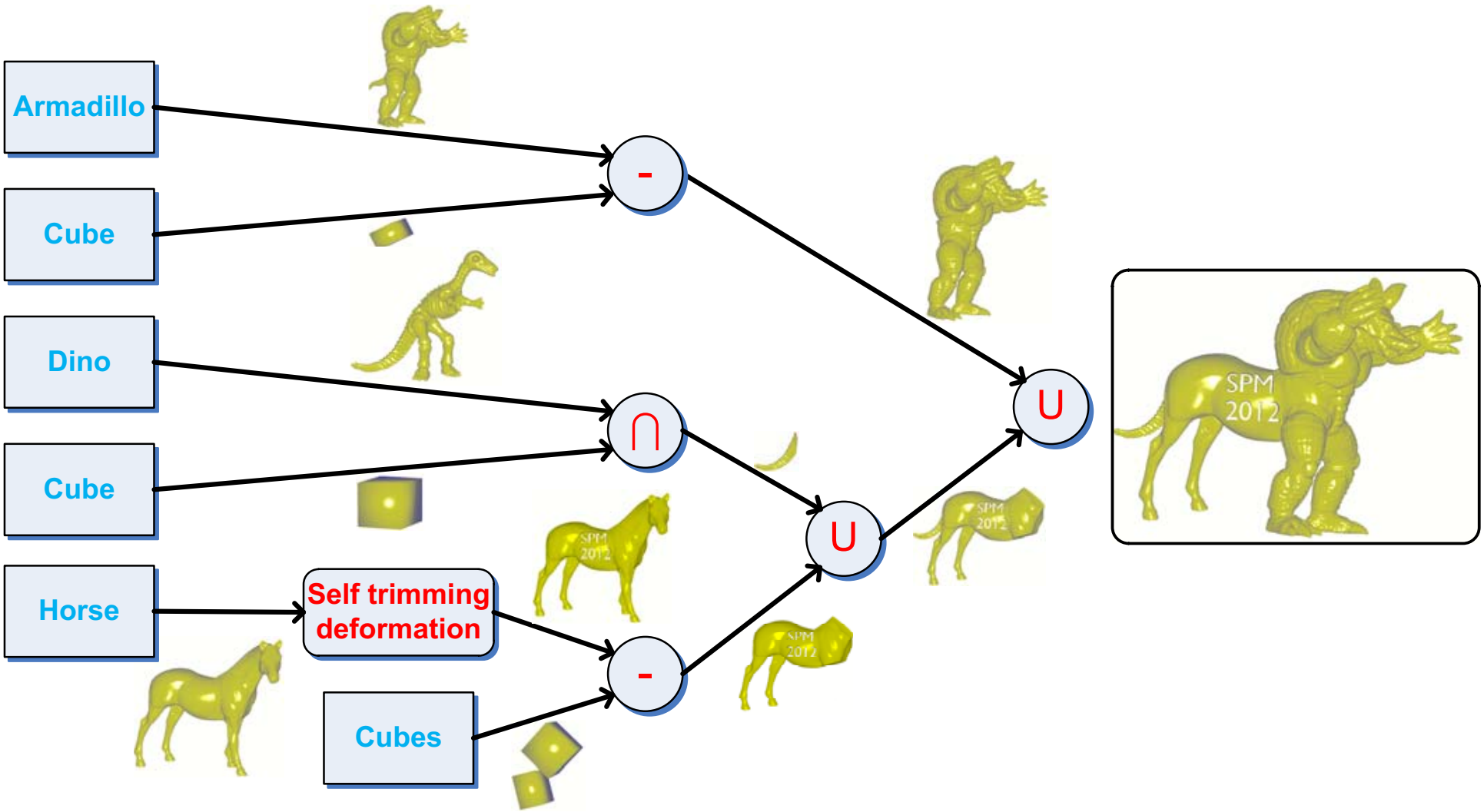


Constructive Solid Geometry

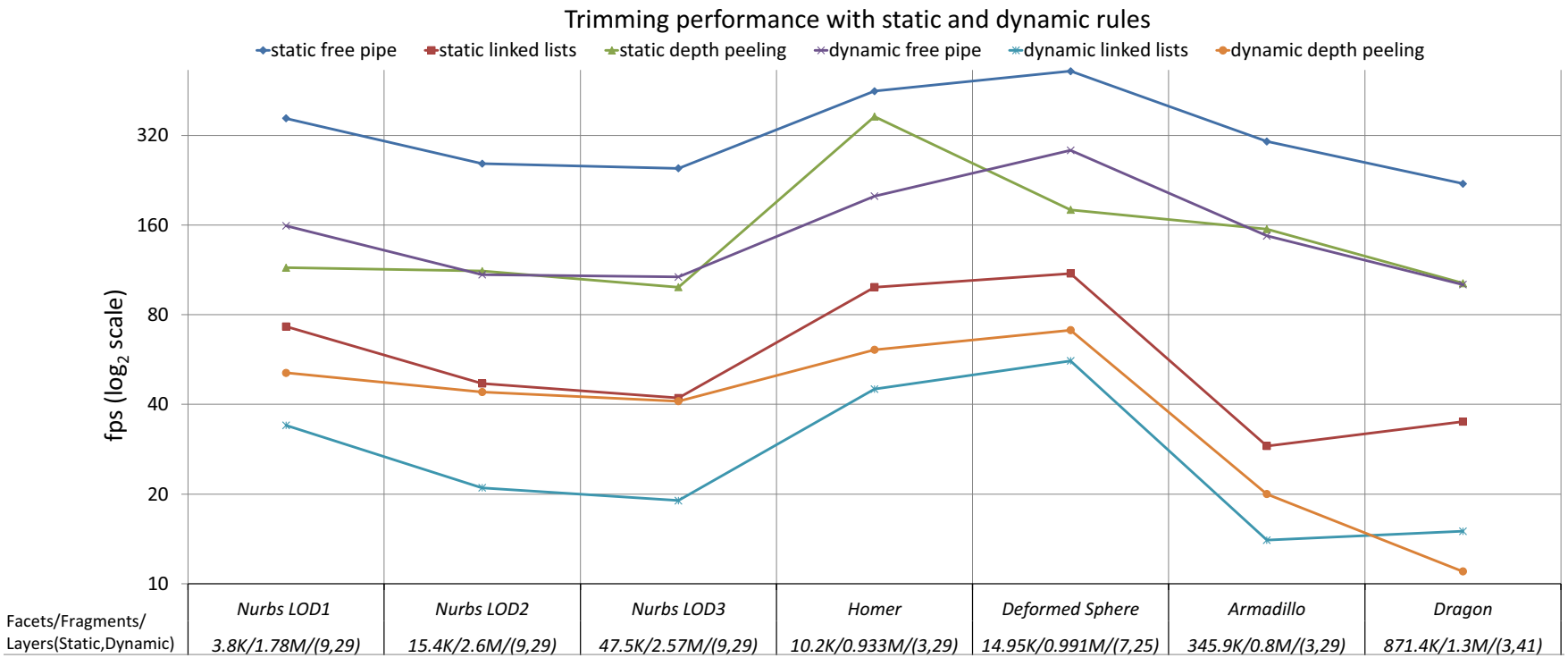




CSG Tree


Armadillo
Cube
Dino
Cube
Horse
Self trimming deformation
Cubes
-
∩
-
U
U
SPM 2012

Performance



Video - Sphere

Video - Nurbs

Outline

Conclusions

Segmentation by combining pose partitionings

- handles off-line, real-time and editable MAs
- supports rigid, highly-deformable and hybrid MAs
- enables multi-resolution & variable segmentations
- introduces smooth visualization of segment transitions

Conclusions

Segmentation by combining pose partitionings

- handles off-line, real-time and editable MAs
- supports rigid, highly-deformable and hybrid MAs
- enables multi-resolution & variable segmentations
- introduces smooth visualization of segment transitions

Approximating MAs with pose-to-pose skinning

- handles both linear and non-linear fitting
- supports rigid, highly-deformable and hybrid MAs
- enables pose editing of arbitrary animation frames
- increases skinning quality via novel vertex and weight corrections

Conclusions

Multi-fragment rendering via S-buffer

- two-geometry-pass GPU-accelerated A-buffer that take advantage of
  - the fragment distribution (exact memory allocation)
  - the sparsity of the pixel-space (improved performance)
- organizes storage into variable contiguous regions for each pixel
- integrates into the standard graphics pipeline

Conclusions

Multi-fragment rendering via S-buffer

- two-geometry-pass GPU-accelerated A-buffer that take advantage of
  - the fragment distribution (exact memory allocation)
  - the sparsity of the pixel-space (improved performance)
- organizes storage into variable contiguous regions for each pixel
- integrates into the standard graphics pipeline

Z-fighting free multi-fragment rendering

- introduces robust and approximate algorithms by extending conventional multi-pass rendering methods
- increases multi-pass rendering performance via geometry culling
- integrates into the standard graphics pipeline

Conclusions

Bounded multi-fragment storage using k^+-buffer

- alleviates prior *k*-buffer limitations and bottlenecks by exploiting fragment culling and pixel synchronization
- introduces an extension to avoid wasteful memory consumption.
- can also simulate the behavior of Z-buffer or A-buffer

Conclusions

Bounded multi-fragment storage using k^+-buffer

- alleviates prior *k*-buffer limitations and bottlenecks by exploiting fragment culling and pixel synchronization
- introduces an extension to avoid wasteful memory consumption.
- can also simulate the behavior of Z-buffer or A-buffer

Real-time (interior) visualization of self-intersecting static/dynamic meshes

- eliminates the cost of computing the self-intersection geometry curves
- introduces static/dynamic rules for the in/out classification
- proposes GPU-based rendering & trimming algorithms
- supports clipping, capping and boolean operations

Directions for future work

Segmentation by combining pose partitionings

- support of time-varying meshes by exploiting vertex mapping
- increased resolution of over-segmentation (cluster similar poses)
- increase the segmentation quality:
  1. use of sophisticated mesh clustering algorithm
  2. rectifying the generated segment boundaries
  3. retain important per pose partitions (confidence criterion)
- retain visualization coherency between "far-away" poses

Directions for future work

Segmentation by combining pose partitionings

- support of time-varying meshes by exploiting vertex mapping
- increased resolution of over-segmentation (cluster similar poses)
- increase the segmentation quality:
  1. use of sophisticated mesh clustering algorithm
  2. rectifying the generated segment boundaries
  3. retain important per pose partitions (confidence criterion)
- retain visualization coherency between "far-away" poses

Approximating MAs with pose-to-pose skinning

- reduce the dimensions of the fitting optimization by:
  1. extracting key-frames
  2. simplifying the rest-pose
  3. performing computations in GPU
- automatically discover the smallest number of suitable joints

Directions for future work

Multi-fragment rendering

- develop a more sophisticated hash function that uniformly divide the non-empty pixels (S-buffer)
- reduce cost of additional accumulation step (S-buffer,k^+-buffer):
  1. lower-detailed subdivision of the initial 3D scene
  2. exploit temporal coherence solutions
- investigate a hybrid depth peeling technique (z-fighting):
  1. captures a sequence of coplanarity-free layers followed by
  2. on demand peeling of overlapping fragments
- explore dynamic k^+-buffer: k value is not the same for all pixels

Directions for future work

Multi-fragment rendering

- develop a more sophisticated hash function that uniformly divide the non-empty pixels (S-buffer)
- reduce cost of additional accumulation step (S-buffer,k^+-buffer):
  1. lower-detailed subdivision of the initial 3D scene
  2. exploit temporal coherence solutions
- investigate a hybrid depth peeling technique (z-fighting):
  1. captures a sequence of coplanarity-free layers followed by
  2. on demand peeling of overlapping fragments
- explore dynamic k^+-buffer: k value is not the same for all pixels

Rendering and trimming of self-crossing solids

- support of fast collision detection
- voxelize the interior and produce actual self-trimmed mesh

My Publications

A. Vasilakis and I. Fudos, *Pose Partitioning for Multi-resolution Segmentation of Arbitrary Mesh Animations*, Computer Graphics Forum (Proceedings of Eurographics 2014), vol. 33(2):?-?, 04-2014.

A. Vasilakis and I. Fudos, *k^+-buffer: Fragment Synchronized k-buffer*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14), San Francisco, USA, 03-2014.

A. Vasilakis and I. Fudos, *Depth-fighting Aware Methods for Multifragment Rendering*, IEEE Transactions on Visualization and Computer Graphics, vol. 19(6):967-977, 06-2013. [also presented at I3D '13]

J. Rossignac, I. Fudos and **A. Vasilakis**, *Direct Rendering of Boolean Combinations of Self-Trimmed Surfaces*, Computer-Aided Design (Proceedings of Solid and Physical Modeling 2012), Volume 45(2):288-300, 02-2013.

A. Vasilakis and I. Fudos, *S-buffer: Sparsity-aware Multi-fragment Rendering*, Proceedings of Eurographics 2012 (Short Papers), 101-104, Cagliari, Italy, 05-2012.

A. Vasilakis and I. Fudos, *Z-fighiting aware Depth Peeling*, SIGGRAPH 2011 (Posters), Vancouver, Canada, 08-2011.

A. Vasilakis, G. Antonopoulos and I. Fudos, *Pose-to-Pose Skinning of Animated Meshes*, Proceedings of Symposium of Computer Animation 2011 (Posters), Vancouver, Canada, 08-2011.

A. Vasilakis and I. Fudos, *GPU Rigid Skinning based on a Refined Skeletonization Method*, Computer Animation and Virtual Worlds, vol. 22: 27-46, 01-2011.

A. Vasilakis and I. Fudos, *Skeleton-based Rigid Skinning for Character Animation*, Proceedings of the Fourth International Conference on Computer Graphics Theory and Applications (GRAPP '09), 302-308, Lisboa, Portugal, 02-2009.

Acknowledgements

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

European Union
European Social Fund

OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS
MANAGING AUTHORITY

NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND

Co-financed by Greece and the European Union

Questions?

Thank you!

Z-fighting (more)

go back

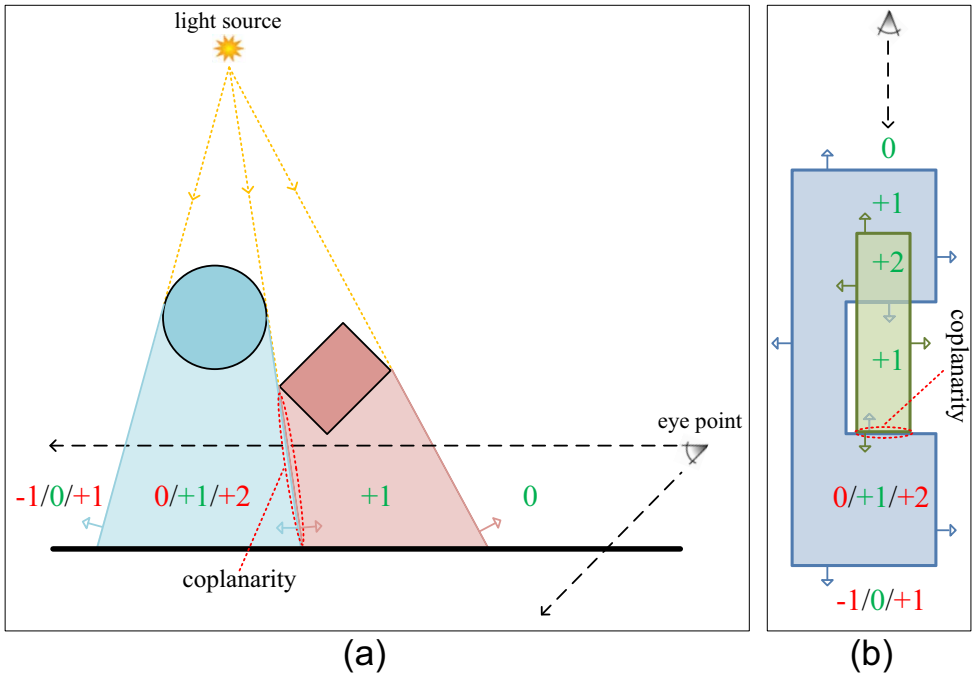


Figure: Illustrating the values of the popular winding number (used for in/out classifications) when ray casting for (a) shadow volume computations and (b) CSG modeling. Red-painted values highlight erroneous computations (in cases where only one of the two coplanar fragments is successfully captured).

Over-segmentation (more)

go back

Algorithm 1 Over-segmentation(A)

1: $OS(A) \leftarrow \oslash$; $VL \leftarrow$ list of all vertices in V;
2: $k_{max} \leftarrow 0$; $\forall v \in V : v.segment \leftarrow -1$;
3: Mark all edges as not visited;
4: **while** $VL \neq \oslash$ **do**
5: $\quad v \leftarrow VL.next$;
6: $\quad$ **if** $v.segment == -1$ **then**
7: $\quad\quad S_{k_{max}} \leftarrow \{v\}$; $v.segment \leftarrow k_{max}$;
8: $\quad\quad k \leftarrow k_{max}$; $k_{max} \leftarrow k_{max} + 1$;
9: $\quad$ **else**
10: $\quad\quad k \leftarrow v.segment$;
11: $\quad$ **end if**
12: $\quad$ **for each** not visited edge $e_i \leftarrow (v, v_i)$ **do**
13: $\quad\quad$ Mark edge e_i as visited;
14: $\quad\quad$ **if** $cav(v) == cav(v_i)$ **then**
15: $\quad\quad\quad S_k \leftarrow S_k \cup \{v_i\}$;
16: $\quad\quad\quad v_i.segment \leftarrow k$;
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad$ Remove vertex v from list VL;
20: **end while**
21: $OS(A) = \{S_0, S_1, \ldots, S_{k_{max}-1}\}$;

Pose-to-pose cleaning (more)

go back

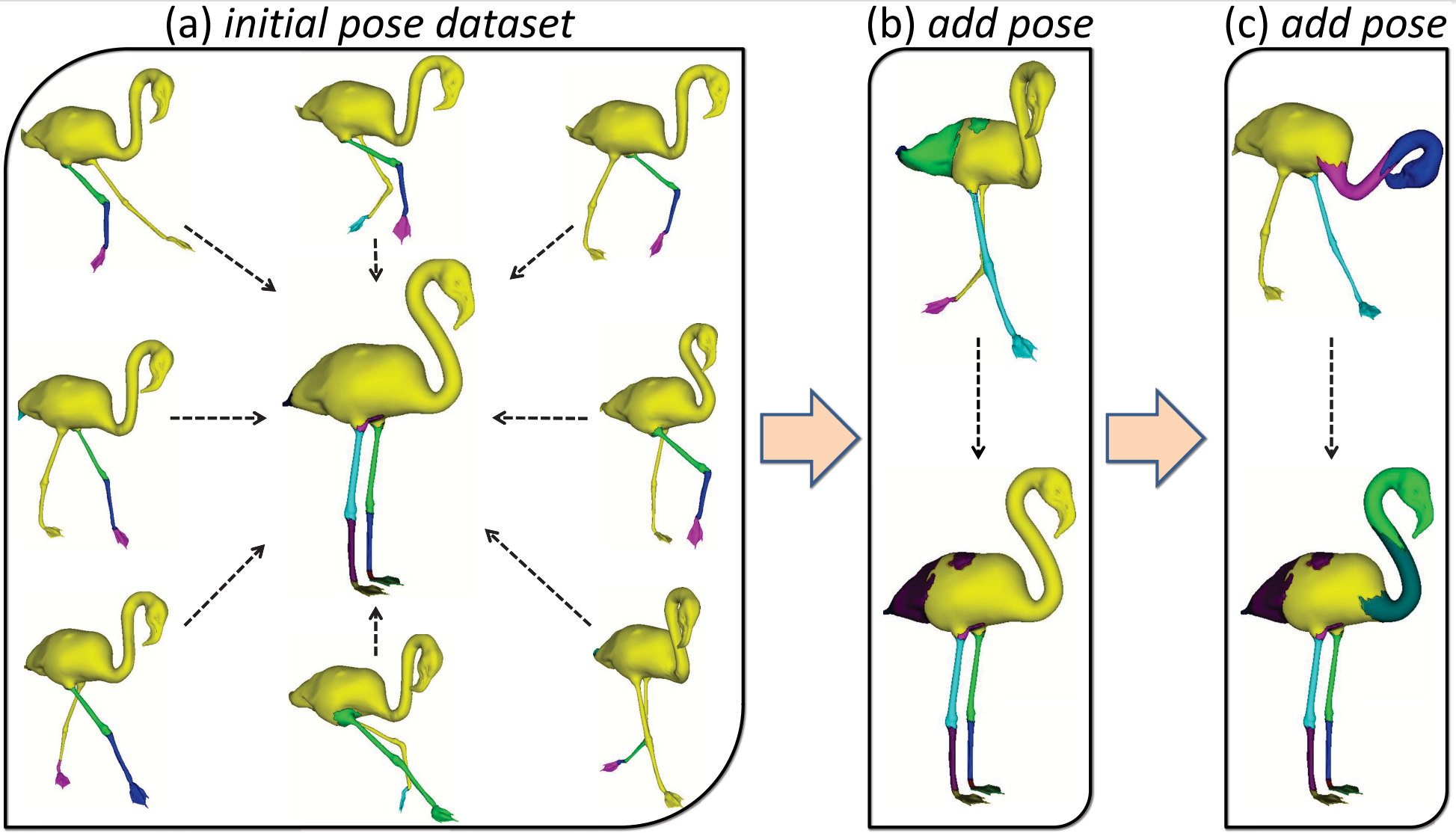
Algorithm 2 p2p-Cleaning($OS(A), h$)

for $p \leftarrow k, 1$ **do**
  OS(A).computeArea();
  for each $S_A \in OS(A)$ **do**
    if $Clean(S_A, C_p^{cav(S_A)[p]}, h, p)$ **then**
      $maxArea \leftarrow a(S_A)$;
      for $S_B \in N(S_A)$ **do**
        if $maxArea < a(S_B)$ **and** $cav(S_A)[1, p-1] == cav(S_B)[1, p-1]$ **and** $cav(S_A)[p] \neq cav(S_B)[p]$ **then**
          $maxArea \leftarrow a(S_B)$;
        end if
      end for
      if $maxArea > a(S_A)$ **then**
        $S_B.Copy(S_A)$; ▷ neighbors,vertices,faces
        comment: Leave $a(S_B)$ unchanged
        $OS(A).Remove(S_A)$;
      end if
    end if
  end for
end for

function CLEAN(S_A, S_P, h, p)
  if $a(S_A) > h \cdot a(S_P)$ **then**
    return false;
  end if
  S_L = new List<Vertex>();
  comment: The following is realized with a breadth
  first search from S_B (similar to Algorithm 1)
  for each $v \in segment(S_A, p-1)$ **do**
    $S_L.Add(v)$;
  end for
  return $a(S_A) \leq h \cdot a(S_L)$;
end function

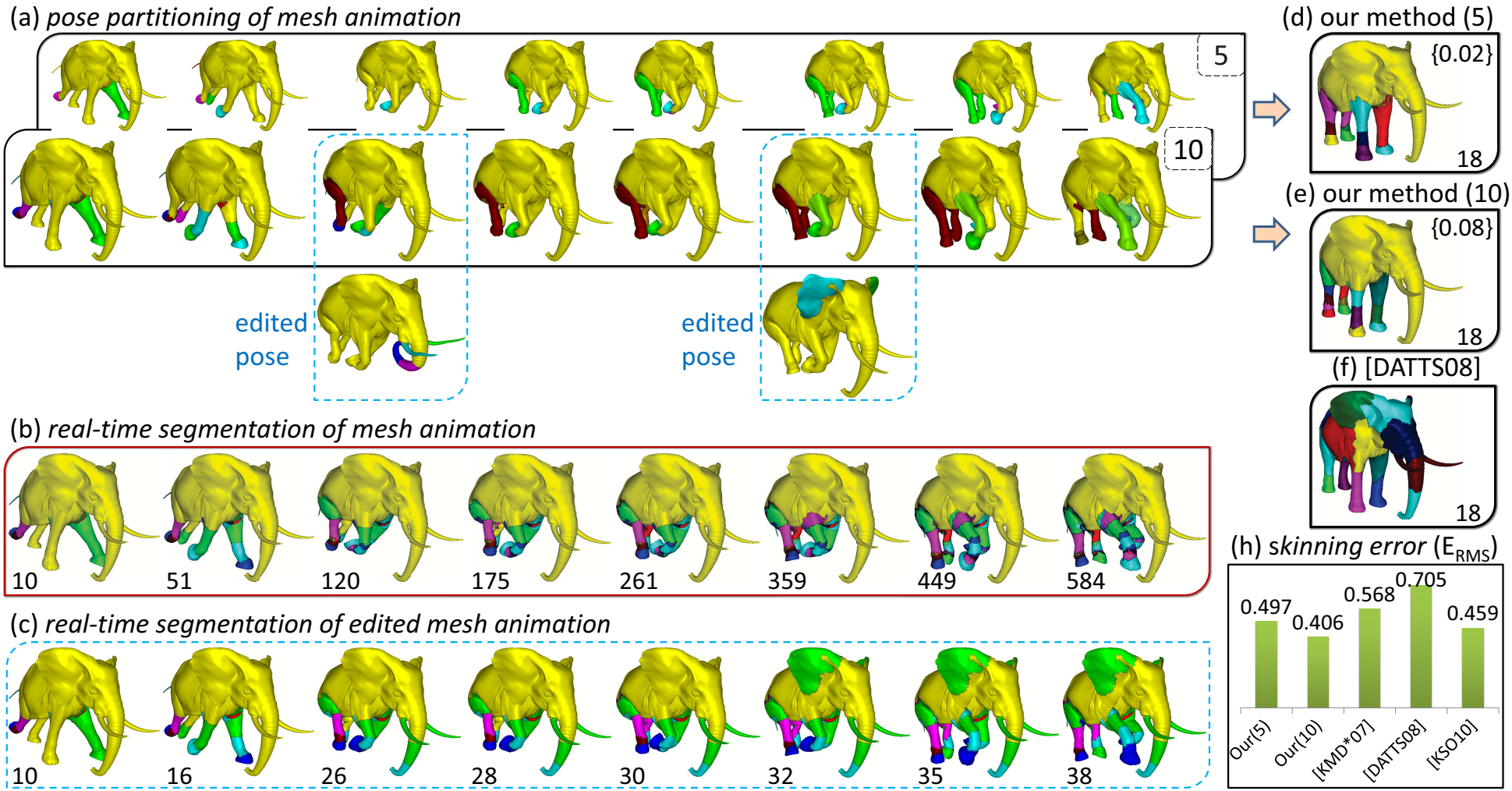
Segmentation (add new poses)

go back



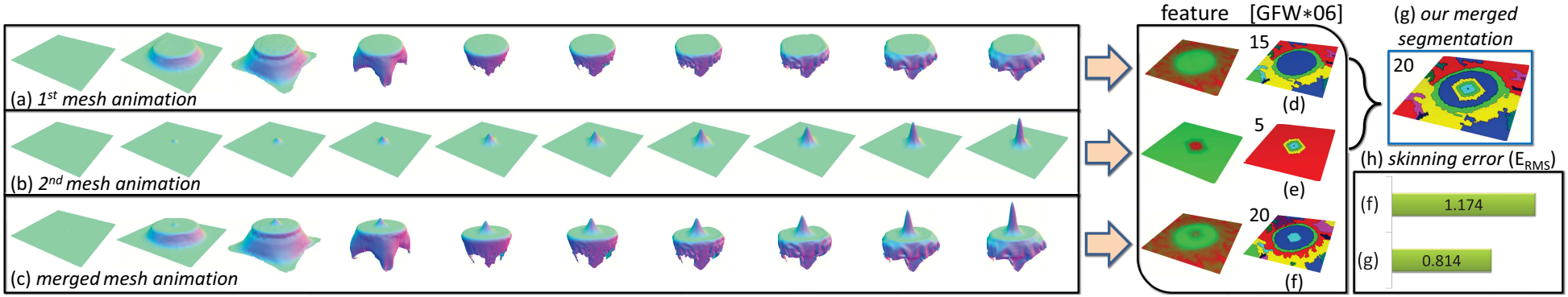
Segmentation (edit poses)

go back



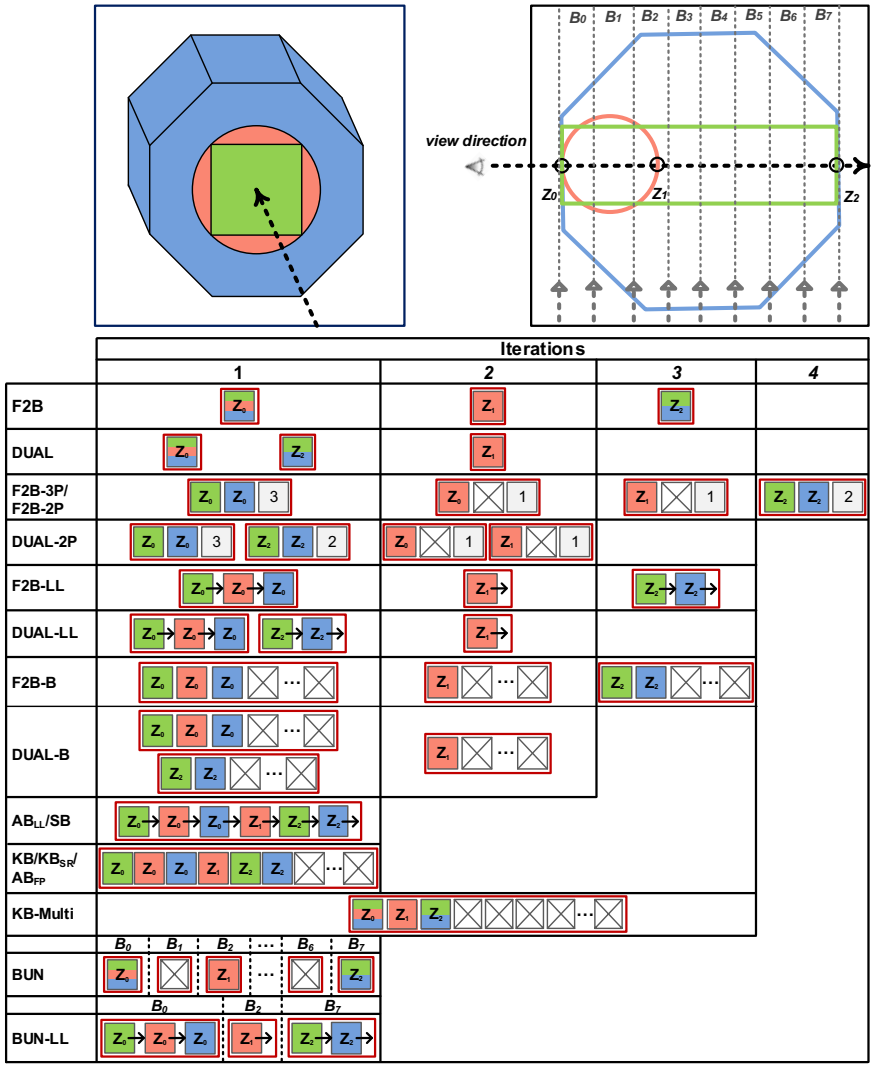
Segmentation transfer

go back



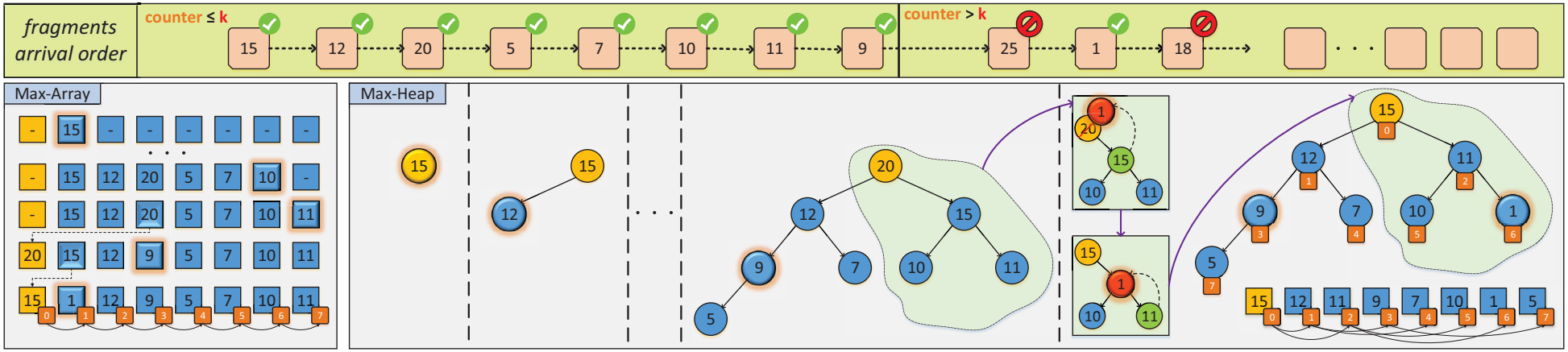
Depth peeling example

go back



k-buffer store example

go back



k-buffer methods details

go back

Algorithm		Performance	Sorting need		Peeling Accuracy		Memory	
Acronym	**Description**	**Rendering Passes**	**on primitives**	**on fragments**	**Max *k***	**32bit Float Precision**	**Per Pixel Allocation**	**Fixed**
KB	Initial *k* -buffer implementation [Callahan2005,Bavoil2007]	1	√	√	8; 16	√	2k; 4k	√
KB-Multi	Multi-pass *k* -buffer [Liu2009a]	1 to k	√	√			2k; 4k	
KB-SR	Stencil routed *k* -buffer [Bavoil2008]	1	√	√	32		3k	
KB-PS	*k* -buffer using pixel synchronization extension [Salvi2013]	1	x	√	-		2k	
K^+B-Array	k^+ -buffer using max-array	1	x	√	-		2k + 2	
K^+B-Heap	k^+ -buffer using max-heap	1	x	√	-		2k + 2	
KB-MDT	Multi depth test scheme [Liu2010,Maule2013]	2	x	x	-	x	2k	
KB-MHA	Memory-hazard-aware *k* -buffer [Zhang2013]	1	√	√	8; 16		2k; 4k	
KB-AB_{Array}	*k* -buffer based on A-buffer (fixed-size arrays)	1	x	√	-	√	2n + 1	x
KB-AB_{LL}	*k* -buffer based on A-buffer (dynamic linked lists) [Yu2012]	1	x	√	-		3f + 1	
KB-LL	*k* -buffer based on linked lists [Yu2012]	1	x	x	-		3f + 6	
KB-AB_{SB}	*k* -buffer based on S-buffer (variable-contigious regions)	2	x	√	-		2f + 2	
K^+B-SB	Memory-friendly variation of k^+-buffer	2	x	√	-		$2f_k$ + 3	
f(p) = # *fragments at pixel* p[x,y]		n = $\max_{x,y}\{f(p)\}$			In A ; B, A denotes the layers/memory for the basic method and B for the variation using attribute packing			
$f_k(p) = (f(p) < k)\ ?\ f(p) : k$		$f_k(p) \leq k$						

Dynamic Trimming Algorithm

go back

Algorithm 13 TrimRenderDynamic(Pixel p, Fragment f)

/* continue until we find the first non trimmed boundary fragment */

1: p^c.depth := C.z;
2: **while** p.color$\neq 0$ **do**
3: **obtain the next fragment** f: f.z$> p^c$.depth
4: p^c.depth := f.z;
5: w_f := (f is *front facing*) ? 1 : -1
6: **if** $frame$.class $\neq 1$ **then**
7: p^r.class := $BC^r[++p^r.\text{count}]$;
8: p^r.index := p^r.index+w_f;
9: **end if**
10: **if** $frame$.class > 0 **then**
11: p^c.index := p^c.index $+w_f$;
12: $CL_b := cCharP$;
13: $CL_a := dynamic\_rule(p^c.\text{index}, p^r.\text{index}, p^c.\text{class})$;
14: p^c.class := CL_a;
15: **if** $CL_a \neq CL_b$ **then**
16: p.color := f.color;
17: **end if**
18: **end if**

CSG Rendering Algorithm

go back

Algorithm 14 RenderCSG(Pixel p, Fragment f)

/* continue until we find the first non trimmed boundary fragment */

1: p^c.depth:= C.z;
2: **while** p.color$\neq 0$ **do**
3: **obtain the next fragment** f: $f.z > p^c.depth$
4: p^c.depth:= f.z;
5: **if** $frame$.class $== 1$ **then**
6: p^r_B.class := $BC^r_B[++Count^r_B]$;
7: **else if** $frame$.class $== 2$ **then**
8: p^r_A.class := $BC^r_A[++Count^r_A]$;
9: **end if**
10: $CL_b := BC^r_{AopB}[Count^c_{AopB}++]$;
11: $CL_a := p^r_A$.class $LOGIC\_OP$ p^r_B.class;
12: $BC^r_{AopB}[Count^c_{AopB}] := CL_a$;
13: **if** $CL_a \neq CL_b$ **then**
14: p.color:= f.color;
15: **end if**
16: **end while**