Direct Rendering of Feature-based Skinning Deformations

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα απο την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Ανδρέα-Αλέξανδρο Βασιλάκη

ως μέρος των Υποχρεώσεων για την λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Ιανουάριος 2014

This work 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.

Τριμελής Συμβουλευτική Επιτροπή

- Φούντος Ιωάννης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Παν/μίου Ιωαννίνων
- Θεοχάρης Θεοχάρης, Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του ΕΚΠΑ - Department of Computer and Information Science, Trondheim Norwegian University of Science and Technology, Norway
- Μεταξάς Δημήτριος, Professor, Department of Computer Science, Rutgers University, USA

Επταμελής Εξεταστική Επιτροπή

- Φούντος Ιωάννης, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Παν/μίου Ιωαννίνων
- Θεοχάρης Θεοχάρης, Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του ΕΚΠΑ - Department of Computer and Information Science, Trondheim Norwegian University of Science and Technology, Norway
- Μεταξάς Δημήτριος, Professor, Department of Computer Science, Rutgers University, USA
- Δημακόπουλος Βασίλειος, Αναπληρωτής Καθηγητής του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Παν/μίου Ιωαννίνων
- Ευθυμίου Αριστείδης, Επίκουρος Καθηγητής του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Παν/μίου Ιωαννίνων
- Παπαγιαννάκης Γεώργιος, Επίκουρος Καθηγητής του Τμήματος Επιστήμης Υπολογιστών του Παν/μίου Κρήτης
- Παπαϊωάννου Γεώργιος, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Ο.Π.Α.

Dedication

To morfeas

Acknowledgements

This work could not be accomplished without support from many people.

First of all, I am extremely thankful to my supervisor Ioannis Fudos for his valuable help, his leading advices and his inexhaustible patience that he has shown during the elaboration of this thesis. Throughout this work he encouraged me to pursue my own research interests and to develop independent thinking and research skills.

I would also like to thank the members of my committee Prof. Theoharis and Prof. Metaxas for their suggestions and insight. Also, many thanks to Prof. Papaioannou, Prof. Papagiannakis, Prof. Dimakopoulos and Prof. Efthymiou for their suggestions and remarks.

Furthermore, I need to thank my colleagues and best friends G. Margaritis and A. Kalogeratos for their honest and moral support through this challenging journey. I have been fortunate to carry out the research for this dissertation in a very friendly environment of my colleagues G. Antonopoulos, K. Tziomakis and A. Lazos. I would not forget my friends and colleagues S. Agathos and N. Papanikos, and A. Hatzieleftheriou, for the pleasant and useful conversations we had for the past five years.

Finally, I would like to thank my parents Theofilos and Sofia, and my sister Katerina, for always believing and unconditionally supporting me. I consider myself very lucky to have them in my life.

This work 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.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Animation

DQS	Dual Quaternion Skinning
FESAM	Fast and Efficient Skinning of Animated Meshes
LBS	Linear Blend Skinning
MA	Mesh Animation
SAD	Skinning Arbitrary Deformations
SCS	Self-Crossing Surface
SMA	Skinning Mesh Animation
STS	Self-Trimmed Surface
TCMA	Temporal-Coherent Mesh Animation
TIMA	Temporal-Incoherent Mesh Animation

Other

CSG	Constructive Solid Geometry
FPS	Frame Per Second
MB	MegaBytes
ms	MilliSeconds
GPU	Graphics Process Unit
PCA	Principal Component Analysis

Multi-fragment Rendering

AB	A-buffer
AB$_{FP}$	A-buffer using FreePipe
AB$_{LL}$	A-buffer using Linked Lists
AB$_{LL-P}$	A-buffer using Paged Linked Lists
BAD	Bucket Adaptive Depth Peeling
BUN	Bucket Uniform Depth Peeling
BUN-LL	BUN combined with Linked Lists
DUAL	Dual Depth Peeling
DUAL-2P	Z-fighting aware DUAL
DUAL-FP	DUAL combined with FreePipe
DUAL-KB	DUAL combined with K-buffer
DUAL-LL	DUAL combined with Linked Lists
F2B	Front to Back Depth Peeling
F2B2	Two-pass F2B
F2B-2P	Two-pass Z-fighting aware F2B
F2B-3P	Three-pass Z-fighting aware F2B
F2B-LL	F2B combined with Linked Lists
F2B-FP	F2B combined with FreePipe
F2B-KB	F2B combined with K-buffer
KB	K-buffer
KB-AB$_{FP}$	K-buffer using FreePipe
KB-AB$_{LL}$	K-buffer using AB$_{LL}$
KB-AB$_{SB}$	K-buffer using S-buffer
KB-LL	K-buffer using Linked Lists
KB-MDT	Multi Depth Test K-buffer
KB-MHA	Memory-Hazard-Aware K-buffer
KB-Multi	Multi-pass K-buffer
KB-PS	Pixel synchronized K-buffer
KB-SR	Stencil Routed K-buffer
MRT	Multi-Render Targets
MSAA	Multi-Sample Anti-Alising
PS	Pixel Synchronization
RMWH	Read-Modify-Write Hazards
SB	S-buffer

Abstract

Vasilakis, Andreas-Alexandros, T., S. PhD,
Computer Science & Engineering Department, University of Ioannina. January, 2014.
Direct Rendering of Feature-based Skinning Deformations.
Thesis Supervisor: Fudos Ioannis.

This thesis studies the problem of direct rendering skinned approximations of arbitrary deformable objects which may also self-intersect on the graphics hardware, which is an important topic in computer animation and visualization. First, we provide efficient methodologies for editable segmentation and skinning representations of arbitrary animated mesh sequences that exploit temporal coherence from a pose-to-pose perspective. Second, we develop rendering algorithms for efficient detection and trimming of (self)-crossing surfaces in the image-space, realized through novel multi-fragment rasterization, without computing any intersections. Since capturing multiple fragments efficiently on the GPU is a challenging task in terms of time, memory and robustness, we study several aspects of the multi-fragment rendering problem from various perspectives and present alternatives for reducing fragment-contention, eliminating z-fighting and avoiding fragment-overflow.

Εκτεταμενη Περιληψη στα Ελληνικα

Βασιλάκης, Ανδρέας-Αλέξανδρος, Θ. Σ. PhD,
Τμήμα Μηχανικών Η/Υ & Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιανουάριος 2014
Παραμορφώσεις Περιβλήματος Βασισμένες σε Χαρακτηριστικά για την Κίνηση Χαρακτήρων
Επιβλέποντας: Φούντος Ιωάννης

Η παρούσα διατριβή μελετά το πρόβλημα της άμεσης απόδοσης των προσεγγιστικών παραμορφώσεων του (πιθανώς αυτο-τεμνούμενου) περιβλήματος αυθαίρετων αντικειμένων, που είναι ένα πολύ σημαντικό θέμα στις περιοχές της προσομοίωσης κίνησης και της οπτικοποίησης. Αρχικά, προσφέρουμε μεθόδους για την αποτελεσματική αναπαράσταση της διαμέρισης και της προσεγγιστικής παραμόρφωσης μιας ακολουθίας στιγμιοτύπων ενός αντικειμένου, στοχεύοντας στη διατήρηση της χρονικής συνάφειας, τη μείωση των σφαλμάτων και την επεξεργασία αυθαίρετων στιγμιοτύπων της κίνησης. Στη συνέχεια, αντιμετωπίζουμε το πρόβλημα απόδοσης αυτο-τεμνόμενων επιφανειών κατά την παραμόρφωση με χρήση σημασιολογικών κανόνων και το πρόβλημα αποδοτικής και ακριβής αποθήκευσης όλων των επιπέδων μιας παραγόμενης σχηνής με χρήση αλγορίθμων που χρησιμοποιούν την ενδιάμεση μνήνη της καρτας γραφικών. Συγκεκριμένα, προτείνονται κανόνες για τον καθορισμό του εσωτερικού και εξωτερικού των χωρικών στοιχείων που σχηματίζονται από μία αυτο-τεμνόμενη επιφάνεια μέσω μίας διαδικασίας συνεχούς παραμόρφωσης μίας απλής κλειστής επιφάνειας. Επίσης, αναπτύσσουμε αποδοτικούς αλγορίθμους πραγματικού χρόνου που αποφεύγουν τα προβλήματα συν-επιπεδότητας, υπερχείλισης και υψηλής συμφόρησης στη μνήμη της κάρτας γραφικών, για την απόδοση του ορίου του στερεού που προκύπτει, αποφεύγοντας τον υπολογισμό των καμπυλών που εκφράζουν τα σημεία αυτο-τομής της επιφάνειας.

Chapter 1

Introduction

This thesis studies the problem of direct rendering skinned approximations of arbitrary deformable objects (including self-intersecting ones) on the graphics hardware, which is an important topic in computer animation and visualization. We consider two broad categories of technical issues: (i) *skinning approximation* and (ii) *interactive rendering* problems with respect to the corresponding programmable graphics rendering stage. Generally, vertex shaders typically transform vertex positions and perform animation, while fragment shaders perform pixel-level lighting effects and customized operations.

Skeletal animation is the standard way to animate virtual characters or mechanical objects for a prolonged period of time. It is commonly used by video games and movie

industry, and can also be applied to mechanical objects or any other object made up of rigid elements and joints. Skinning is a simple yet popular skeletal deformation technique, implemented in almost all modern 3D engines, combining compact storage with efficient hardware accelerated rendering by vertex shaders. The mesh movement of the character is defined as the function of its underlying skeleton. The process where the specification of the character animation skeletal structure is built and then attached to the character surface, allowing skeletal motion data to animate the entire character model is mostly driven by a shape decomposition method. In this dissertation, efficient deformation-driven segmentation methods are studied and developed for guiding high-quality skinning approximations of arbitrary mesh animations.

Self-intersecting or coplanar geometry may accidentally or intentionally occur when the user is dynamically constructing the mesh animation via complex editing and *constructive solid geometry* (**CSG**) operations. For example, when planning heart surgery, the surgeon may wish to deform a 3D model of the patient's anatomy to add a connection between two nearby vessels or to create a hole in the wall that separates the two ventricles. While in a static model, the user could be asked to select which manifold portions of the surface should be removed by clicking on them, if we want to apply these topological changes to an animated model, we cannot expect the user to perform these selections at each frame. Visualizing the interior and hidden portions of solids is useful in many applications, from accessing and visualizing inter-body human anatomy to rendering of interior architectural spaces, however it is not supported for self-trimmed surfaces. To this end, we have focused on the problem of accurate and interactive (interior) rendering, e.g. detecting, trimming or combining via CSG, of self-crossing surfaces realized through novel multi-fragment rasterization techniques.

1.1 Computer Animation

Computer animation is the process used for generating animated images by using computer graphics [71]. The more general term *computer-generated imagery* encompasses both static scenes and dynamic images, while computer animation only refers to moving images. To create the illusion of movement, an image is displayed on the computer monitor and repeatedly replaced by a new image that is similar to it, but advanced slightly in time, usually at a rate of 24 or 30 frames per second (FPS).

Computer animation plays a major role in 3D visualization process by being used in a broad spectrum of applications such as manufacturing, medicine, clothes and fashion, and the entertainment industry (movies and games). More specifically in the context of the latter, rapid realistic animation of articulated characters is a key issue in video games, crowd simulations and computer generated imagery films [100].

1.1.1 Character Animation

Character animation is a specialized area of the animation process, which involves bringing animated characters to life [92]. Even though other approaches exist and are used for certain applications, most character animation is created using key-frame systems. Character animation usually comprises the following processes:

(i) Setting up the motion system: a process that usually involves determining and/or adjusting a set of user-controlled kinematic handles.

(ii) Attaching the manipulation structure to a target surface skin.

(iii) Determining how the skin will behave (*deform*) under motion.

Processes (i) and (ii), often called *rigging*, are in charge of the motion, i.e. the kinematics effects. The final process (iii) determines the mesh deformation and is therefore responsible for the final visual effects.

Deformation techniques that improve visual fidelity, computational efficiency and make intuitive use of the character animation structure have been investigated extensively in the literature [22]. *Surface-based* deformation methods [11, 164] advocate the use of *differential coordinates*, to produce aesthetically pleasing animations. From a user perspective, surface-based techniques provide an intuitive deformation interface allowing direct positioning and manipulation of arbitrary handles. While it leads to detail-preserving deformation output, a variational optimization method is involved that compromises real-time performance, a fact that makes interactive animation editing infeasible. On the other hand, *space-based* deformation techniques [38] indirectly reshape an object by warping the surrounding space. In the context of character animation, a low-poly control mesh (often called *cage* [62, 60, 86]) is defined that encloses the target model. In spite of its modeling speed and simplicity as compared to surface-based techniques (by manipulating the cage points, the deformation is propagated to the influenced encapsulated mesh portion), constructing and controlling such structures is not a straightforward task. Finally, Cohen-Or [24] explored the potential of combining the advantages of surface-based and space-based deformation methods.

Alternatively, *skeleton-based* deformation is one of the most popular techniques in computer animation because of the intuitive use of bones as deformation handles that naturally capture the physical rigidity of the character parts. The appeal of using skeletons is intuitively interpreted from the fact that most articulated creatures in the real world (humans [22], quadrupeds [136] and others) are bound to move guided by their internal endoskeleton kinematics. An animation skeleton has usually much simpler structure than the original object and aims at simplifying the deformation process by avoiding the tedious task of animating each vertex independently. Specifically, a skeletal representation model consists of at least three main layers:

(a) a highly detailed 3D triangular surface mesh of the character (Figure 1.1(left)).

(b) an underlying skeleton (*rig*) attached to the skin, defined as a hierarchical tree structure of *joints* connected with rigid links (*bones*) (Figure 1.1(center)).

(c) an additional layer to improve realism incorporating the physical properties of the character's musculature [81] at the cost of performance is *optionally* provided.

A process of extracting a skeleton from a static pose of the character (*skeletonization*) or adapting a given animation skeleton to the character surface (*skeleton embedding*), is initially employed, allowing direct manipulation, inverse kinematics [15] or skeletal motion data [3] to animate the mesh accordingly.

The skinning framework is the predominant technique for real-time skeleton-driven character animation in spite of its limitations [102]. The skeleton is rigged to the target character mesh by assigning multiple blending weights for each influencing bone (Figure 1.1(right)). Then, each point on the surface of the character is transformed by a weighted combination of the affine transformations of each influencing bone. Recently, Jacobson et al.[54] have enhanced its scope by introducing besides skeletons, point and cage deformation handles that are unified under the skinning setup. *Matrix palette skinning* [93], mostly known as *linear blend skinning* (**LBS**), is the most widely used character skinning technique due to its computational efficiency and straightforward implementation in graphics hardware [9, 82]. Although LBS assumes the existence of some underlying skeletal hierarchy, it can still be applied to *highly deformable* objects [68]. Highly deformable animations are used to describe objects that deform under no skeletal influence (for example animations of cloth and soft body internal organs), independently or in conjunction with skeletal animation (*hybrid animation*).

Skinning methods have been widely criticized for requiring trained artists to perform a tedious and cumbersome process of manual weight painting to obtain satisfactory results [166, 147]. A multitude of research approaches on computing weight influences in an automated manner from a target mesh have appeared at the literature in the past few years. Unfortunately, there is no widely agreed criterion for weight selection which is universally acceptable for all applications.

1.1.2 Mesh Animation

A 3D *mesh animation* (**MA**) consists of a sequence of key-frames (*poses*), representing how a static 3D shape is evolving through time. While the production of such a sequence can be done by advanced scanning machinery [3] or multiple video cameras [153, 137], in most cases it is the result of strenuous labor from the part of artists, who create the animation pose by pose. While specialized computer software provides several automated techniques for the generation of deformations implementing several of the aforementioned algorithms, adding fine details always requires human intervention. Note that since each pose of the MA may be independently reconstructed, sequence of meshes with varying connectivities as well as varying topology may result. These MAs are most commonly called *time-varying* or *temporal-incoherent* (**TIMA**) [4]. On the other hand, if the connectivity

Figure 1.1: Illustrating (left) the original character, (center) the underlying skeleton attached to the skin and (right) the resulting influencing weights.

is constant over the whole sequence, then MA is called *temporal-coherent* (**TCMA**) [4]. A mesh that is animated using skeleton animation is called a *skinning mesh*. From a different perspective, MAs can be roughly divided into two categories:

(i) *off-line* which consists of a fixed number of stored consecutive animated meshes

(ii) *real-time* which is either streamed from a shared distributed virtual environment or dynamically generated from interactive manipulation of a deformable object.

1.1.3 Segmenting Mesh Animations

Segmentation of MAs, despite being a new research field when compared to static mesh partitioning [130], has become a key issue in a number of computer graphics applications (a brief collection is highlighted in Table 1.1). Animation compression [2], deformation transfer [84], skinning mesh animations [55, 69, 80], skeleton extraction [126, 31, 48], are representative applications, highly related to character animation, enabled by partitioning a deforming mesh sequence.

While the output depends on the type of application, the main goal of segmentation is to partition the animated mesh into regions with similar *motion characteristics*. Several motion properties have been proposed for defining the feature space. Significant features, which form a dense region in feature space, can be detected by one of the numerous available clustering techniques. From now on, we denote such approaches as *global segmentation* methods, because they work with average motion measures that represent the degree of deformation during the entire animation sequence.

Regardless of the clustering criteria, current global segmentation methods focus on detecting segments with mostly rigid behavior, failing in partitioning correctly highly-

deformable objects. Moreover, these methods are rather limited in cases where the feature vector space is

- *flat*: resulting at an inability to separate regions with similar feature values (zero-variance)
- *anisotropic*: resulting at one or more features dominating the others due to higher variation.

In addition, the segmentation output is highly dependent on a large set of *parameters* that should be determined a priori. Last but not least, the entire process cannot be carried out when the mesh sequence is *modified* [19] by:

- performing subjected editing operations or
- being augmented with additional poses that did not exist in the original MA.

Furthermore, it would be advantageous to have a tool for an automatic conversion from a high segmentation resolution to a lower one. This would remove the cost of reconstructing the entire segmentation for each resolution. Thus, one might ask whether it is possible to design a segmentation algorithm with reasonable time complexity that is not limited to work only on one type of MAs and avoids most of the aforementioned artifacts.

1.1.4 Skinning Mesh Animations

Although pre-computed animations provide greater flexibility at the time of animation creation than skeleton-based methods, they output more space-consuming dynamic scene representations comprising of independent position data streams. The ramifications of size manifest not only in terms of the space used on the disk but also in terms of time and space required by an application to load the sequence on the graphics process unit (GPU). Note that the amount of space and processing time increases dramatically considering that a scene may contain more than one animation sequence.

To this end, skinning techniques have been explored to approximate arbitrary deformable MAs automatically specifying a relatively small number of virtual [55, 68, 69] or hierarchical skeleton bones [126, 31] to act as control joints to derive the skin from the initial rest-pose. Except of the context of data reduction, the fully-automatically transform of MAs into compact and easily modified skeletal versions enables a full repertoire of already existing skeletal animation tools. Some representative ones are easy post-preprocessing, level of detail selection [113, 125], collision handling [49, 70] and efficient hardware accelerated reproduction of the initial input sequence.

While recent methods perform well for a variety of input sequence classes, they mainly focus on optimizing the skinning approximation. However, bone estimation and weight definition are critical to several applications that need to maintain locality such as deformation-driven compression, pose editing, animation transfer, collision detection and

progressive animation. Current techniques cannot support the whole spectrum of these applications.

1.1.5 Self-intersected Mesh Animations

Most steps in the geometry processing pipeline, like deformation, smoothing, subdivision and decimation, may create self-intersections. The approximated skinned surface may also result in a self-crossing surface. In the context of deformation, the designer may for example use 3D input devices to grab, pull, and twist the 3D model in natural and predictable ways to intendedly create self-intersecting surfaces. Space-based deformation techniques [38] are a popular paradigm for designing 3D shapes. They afford an intuitive direct manipulation and seem most appropriate for editing medical and artistic models or animations. Unfortunately, these methods lack useful semantics of what happens when the designer wishes to create a self-intersecting surface model. One may argue that in a static model, the user could be asked to select which manifold portions of the surface should be removed by clicking on them. More importantly, if we want to apply these topological changes to an animated model, we cannot expect the user to perform these selections at each frame. Figure 1.2 illustrates an initial manifold boundary that is not self-crossing and a continuous process that deforms this surface creating non-manifold self-intersection areas where the solid "passes through itself" while keeping it an immersed sub-manifold.

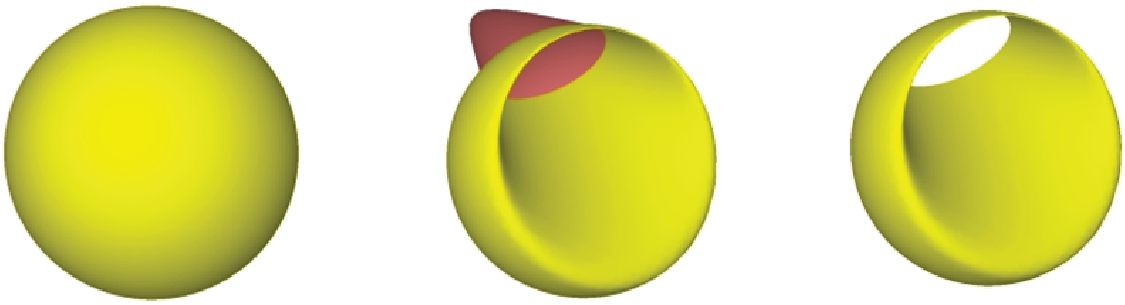


Figure 1.2: Illustrating (left) the original manifold boundary, (center) a pose produced by continuously deforming the first pose (we use pink to illustrate the part of the surface that should be trimmed) and (right) the trimmed result.

We need semantics for defining the watertight solid that represents the boundary of the regularized union of the interior components, which is a subset of the initial surface and is called the trimmed boundary or simply the *trim*. In this context, two generic problems have been identified and several rules have been devised to capture them:

Problem I: Given an SCS (Figure 1.2 (b)) determine the trim (Figure 1.2 (c)).

Problem II: Given an initial manifold boundary (Figure 1.2 (a)) and a continuous process that deforms this surface to an SCS (Figure 1.2 (b)) determine the trim (Figure 1.2 (c)).

Image-based techniques avoid the complexity of both creating and updating the polygonal mesh at each time step. This classification and rendering process is accomplished in real-time through a rasterization process by testing *surfels* (without computing any self-intersection curve), and hence is suited to support animations of self-crossing surfaces. The advantage is that the trimmed solid can be combined with other solids and with half-spaces using boolean operations and hence may be capped (trimmed by a half-space) or used as a primitive in direct CSG rendering. Surfels are represented by *fragments* of the surface that arise from the intersection of the surface with a pixel ray originated at the center of the corresponding pixel of the viewing plane (see Figure 1.3). Surfels are compactly stored in several images (*multiple-layers*) to model the complete geometry seen from one viewpoint.

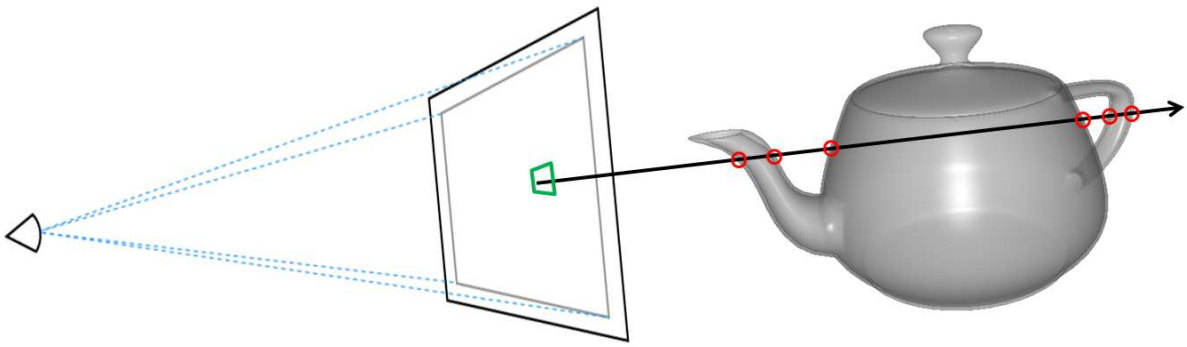


Figure 1.3: Illustrating the fragment construction process (highlighted with red circles) on a selected (green) pixel, when ray casting the teapot model.

1.2 Multi-fragment Rendering

Determining visibility when rasterizing objects in arbitrary order is a challenging task in terms of time and space for a host of algorithms that simulate complex rendering effects in real-time. Many image-based techniques produce visually realistic results at interactive speeds in games (order-independent transparency [97], shadows [167], hair rendering [168]) and other graphics applications (volume rendering [14], collision detection [56], CSG [74, 170], trimming [44]) enabled by a family of GPU-accelerated methods that capture the surface-intersections (fragments) when ray-casting from the viewer position through each screen pixel.

Z-fighting is a phenomenon in three-dimensional rendering that occurs when two or more primitives have the same or similar values in the Z-buffer [20] (see Figure 1.4). Z-fighting may manifest itself through:

1. intersecting surfaces that result in intersecting primitives

2. overlapping surfaces, surfaces containing one or more primitives that are coplanar and overlap

3. non-convergent surfaces due to the fixed point round-off errors of perspective projection.

Traditional hardware-supported rendering techniques do not treat Z-fighting and render only one of the fragments that possess the same depth value. This results in dotted or dashed lines or heavily speckled surface areas. In this context, Z-fighting cannot be totally avoided and may be reduced by using a higher depth buffer resolution and inverse mapping of depth values in the depth buffer [27] or using depth bias [52]. Multi-fragment capturing techniques are even more susceptible to Z-fighting, because they need to examine all fragments (even those that are not visible) in a certain order (ascending, descending or both) before deciding what to render. Thus, they may encounter multiple Z-fighting triggered liabilities per pixel (see Figure 1.5).

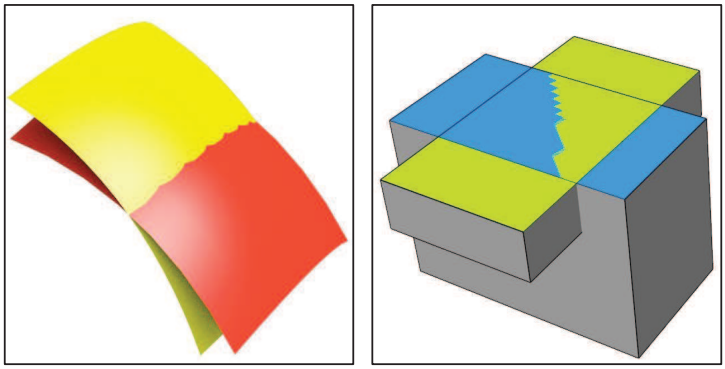


Figure 1.4: Illustrating unpleasant effects when rendering (a) intersecting or (b) overlapping surfaces on popular modeling programs such as (a) *Blender* 2.5 and (b) *Google SketchUp* 8.

Storing multiple fragments efficiently in a single geometry pass in terms of time and space is a challenging task. A-buffer [17] was the first method to capture all fragments per pixel based on variable-length lists during geometry rasterization, followed by a post-sorting process that correctly reorders fragments by depth. An actual GPU-accelerated implementation of the A-buffer based on atomic memory operations was introduced in [167]. The algorithm scales well and runs in linear time on the number of generated fragments, but its performance degrades rapidly in cases where heavy access on the GPU shared memory is necessary. If fragment overflow occurs, they propose to dynamically reallocate memory and then re-render the scene. Otherwise, much of the allocated memory goes unused. Moreover, FreePipe [90, 29] maintains multiple fragments using constant-size per pixel vectors. Despite its high computation speed, it suffers from large and potentially unnecessary memory consumption.

Several variants and optimizations [97] have been proposed recently to simulate the behavior of the A-buffer architecture with reduced memory requirements. k-buffer facilitates

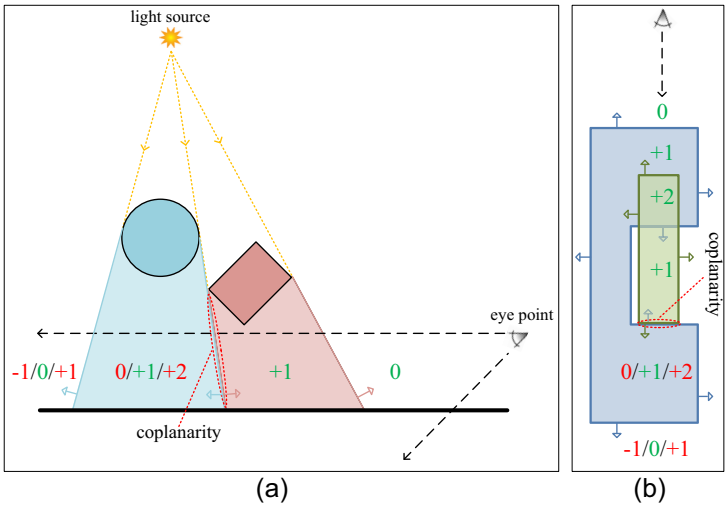


Figure 1.5: Illustrating the values of the popular winding number (used for in/out classifications) when ray casting for (a) shadow volume computations and (b) CSG modeling. Red-painted values highlight erroneous computations (in cases where only one of the two coplanar fragments is successfully captured).

novel approaches to multi-fragment rendering by maintaining the k-nearest fragments on the GPU. Various alternatives have been proposed to alleviate its memory hazards and to avoid completely or partially the necessity of geometry pre-sorting. However, that comes with the cost of excessive memory allocation and depth precision artifacts.

1.3 Related Work

This section presents a brief summary of prior art, focusing on the methods most closely related to the methods covered in this thesis. Readers may refer to Chapter 2 for a detailed mathematical and multi-layer rendering background.

1.3.1 Clustering Methods and Feature Space

Several approaches have recently adapted conventional static segmentation algorithms [130] to work with 3D deforming meshes exploiting the analysis of the motion information. While the output depends on the type of application, the main goal of segmentation is to partition the animated mesh into regions with similar motion characteristics [83]. Table 1.1 summarizes the global segmentation methods proposed in the last decade to assist several applications. More specifically, it includes the *feature space* and the *clustering technique* explored by each work.

More specifically, Du et al. [32] utilized a *multi-source region growing* algorithm based on similarity of statistical *variability* characteristics to favor grouping between surface regions. Similarly, Lee et al. [84] partitioned a mesh sequence into clusters with similar rigid motion growing feature clusters based on *geodesic* and *deformation* distances.

Methods proposed by [68, 69] used *uniform distribution* and *deformation gradients* [138] to initialize their skinning decomposition, respectively.

To this end, Le and Deng [80] replaced previous clustering methods with *K-means* and used bone *transformation matrices* as assignment attribute. Amjoun and Straber [2] further employed K-means based on the local similarity between the *trajectories* in a cluster-defined coordinate system to assist their compression method. When the number of resulting clusters is unknown a priori, *mean-shift* clustering was applied based on *rotation matrices* [55] and *geometric invariant* feature vectors [85] to segment animated objects into near-rigid components.

Works of [126] and [157] derived a *bottom-up hierarchical* clustering by merging (initially per facet assigned) clusters until one node remains based on *rigid* and *affine transformation* metrics, respectively. Conversely, Gunther et al. [41] applied a *top-down hierarchical* approach to break mesh down into sub-meshes with similar *affine motion*. Starting from one cluster which represents the entire object, a partition is created by segmenting it into two or more components. On the other hand, Arcila et al. [4] proposed to incrementally refine the final segmentation as a new pose arrives by splitting current components into parts which present consistent *rigid* motion.

Research work by [142, 31] and [48, 4] exploited *spectral clustering* to segment a deformable mesh into approximately rigidly moving groups using *euclidean distance* and *rotation angle* similarity metrics, respectively. Moreover, Feng et al. [37] used spectral clustering for effective curvilinear feature detection and deformation discontinuity detection. Finally, Wuhrer and Brunton [162] produced a near-rigid segmentation for finding the *minimum spanning tree* of the mesh's dual graph weighted by *dihedral angles* of neighbor faces.

Close to our method, *co-segmentation* techniques [133, 53] may be adjusted to consistently partition a sequence of animated poses. However, their performance is relatively slow and they are limited to work only on quasi-rigid animations with fixed frames and not-editable poses.

1.3.2 Skinning and Editing Mesh Animations

Animation Compression

SMA (Skinning Mesh Animations) [55] was the first to address the problem of generating progressive skinning approximations from MAs. While their algorithm works well for quasi-rigid input animations enabling hardware-accelerated rendering and efficient animation processing, suboptimal approximations have been observed when applied to highly deformable regions because they lack the near-rigid structure for skinning. **SAD** (Skinning Arbitrary Deformations) [68] presented optimal skinning approximations of highly deformable animations exploiting dual quaternions advantages [67] with the cost of higher preprocessing times and data storage due to the large number of required proxy-joints. Moreover, their uniform mesh clustering often leads to unpleasant artifacts on articulated

Prior Art	Clustering Method	Feature	Application
[131]	Region Growing	Euclidean & Angular distances	Metamorphosis
[124]	Clustered PCA ⋆	Euclidean distance	Compression
[155]	Clustered PCA ⋆	Euclidean distance	Editing
[32]	Region Growing	Variability characteristics	PSDM †
[84]	Region Growing	Geodesic & Deformation	Segmentation
[69]	Region Growing	Deformation gradients	Skinning
[68]	P-center	Euclidean distance	Skinning
[80]	K-means	Affine motion	Skinning
[2]	K-means	Trajectories similarity	Compression
[106]	K-means	Planarity	Triangle Sorting
[55]	Mean-shift	Rotation matrices	Skinning
[85]	Mean-shift	Geometric-invariant vectors	Segmentation
[126]	Hierarchical	Rigid motion	Skeletonization
[157]	Hierarchical	Affine motion	Skinning
[41]	Hierarchical	Affine motion	Ray Tracing
[4]	Hier. + Spectral	Rigid motion	Segmentation
[142]	Spectral	Euclidean distance	Animation Collage
[31]	Spectral	Euclidean distance	Skeletonization
[48]	Spectral	Rotation angle	Skeletonization
[37]	Spectral	Geodesic & Deformation	Geometry Images
[162]	MST ‡	Dihedral angle	Segmentation

Table 1.1: Summary of the major global segmentation techniques. ⋆ : Principal component analysis. † : Piecewise statistical deformable model. ‡ : Minimum spanning tree.

animations and highly detailed animated regions because it does not attempt to capture important parts of the animation. Subsequently, Xian et al. [163] reduced the dimensions of the SAD linear system substantially by representing animated meshes as progressive decimated models. As a result, the fitting time is reduced at the expense of a significant increase of the fitting error. **FESAM** (Fast and Efficient Skinning of Animated Meshes) [69] produced more accurate approximations of arbitrary deformations without any corrective technique in a fraction of the time than previous methods. However, this method supports spatial but not temporal correlation and cannot preserve locality since it does not use spatially consistent clustering. Extensive research has investigated how to convert MAs into skeletal-based animations [126, 31]. Extracting a skeleton and skinning weights enables modifications using the full repertoire of already existing editing tools for skeletal animation. However, these methods cannot support animations that lack any apparent or inherent skeletal hierarchy.

Various animation compression techniques have been successfully applied to mesh sequences [111]. Different techniques have been presented offering significant animation compression based on principal component analysis (PCA) [2], clustered PCA represen-

tations [124], wavelet approaches [42, 109], prediction coding [64] and geometry image coding techniques [37].

Although these methods deliver efficient compression and some of them provide fast GPU reconstruction and rendering [124, 37], robust mesh segmentation [124, 2, 37] and level of detail control [37], they mainly focus on optimal data reduction. Thus, such representations cannot be combined with a real-time skinning system, disallowing further hardware accelerated operations to be performed directly.

Animation Editing

Previous skinning mesh animation methods [55, 68, 69] do not focus on editing but rather on limiting bandwidth requirements for fast GPU rendering. In particular, SMA and SAD frameworks allow limited editing, either by manually modifying the bone transformations for each frame, or by adding small changes at the rest-pose. However, these editing tools cannot be applied in conjunction with skinning corrections, resulting at sub-optional edited approximations. Similarly, FESAM cannot support editing operations due to the rest-pose position optimization.

Jia et al. [58] proposed a fast method to transplant different sources of motion to skinned meshes without extracting a hierarchical bone structure which makes it applicable to arbitrary skeletal SMA. Based on the same idea, they also devised an SMA editing tool which allows users to edit frames interactively. Editing can then be propagated to all subsequent frames. Dutreve et al. [33] presented a method to add fine details, such as wrinkles and bulges, on a virtual face animated by common skinning techniques. However, it is limited to facial animations. Significant research has been conducted on editing operations on dynamic meshes [73, 165, 139]. Despite the high visual quality of the resulting edited animations, the main advantages of skinning (i.e. compact animation representation combined with efficient GPU rendering) are not met.

1.3.3 Multi-fragment Rendering

Efficient capturing of global information of the scene is an important feature in many graphics applications for simulating multi-fragment effects all of which require operations on more than one fragment per projected pixel area. Fragment level techniques work by sorting surfaces viewed through each sample position, avoiding the sorting drawbacks that occur in object/primitive sorting techniques [123, 134] (for example geometry interpenetration, primitive splitting, support of dynamic scenes) or hybrid methods that order the generated fragments by exploiting spatial coherency [160, 39, 18]. These algorithms can be classified in two broad categories, those using *depth peeling* and those employing *hardware implemented buffers*, according to the approach taken to resolve visibility ordering [97].

Depth Peeling

Given the limited memory resources of graphics hardware, *multi-pass rendering* is often required to carry out complex effects, often substantially limiting performance. Probably, the most well-known multi-pass peeling technique is *front-to-back depth peeling* (**F2B**) [36], which works by rendering the geometry multiple times, peeling off a single fragment layer per pass. *Dual depth peeling* (**DUAL**) [8] speeded up multi-fragment rendering by capturing both the nearest and the furthest fragments in each pass. Finally, *bucket uniform depth peeling* (**BUN**) [89] extended dual depth peeling by extracting two fragments per uniform clustered bucket. To reduce collisions at scenes with highly non-uniform distributions of fragments, they further proposed to adaptively subdivide depth range (**BAD**) according to fragment occupation [135] at the expense of extra geometry passes and larger memory overhead.

However, all currently proposed depth peeling techniques cannot deal with fragments of equal depth, thus detecting only one of them and missing the others. A number of solutions have been introduced to alleviate coplanarity issues in depth peeling. Cole and Finkelstein [25] proposed *id peeling*, which addresses artifacts where lines obscure other lines by allowing a line fragment to pass only if its index is lower than the highest index at the corresponding pixel in the previous iteration. Despite its accurate behavior, it peels only one fragment per peeling iteration and cannot support rendering of occluded edges. Recently, Busking et al. [13] introduced *coplanarity peeling* extending F2B with in/out classification masking. However, it can only distinguish coplanar fragments between different objects that do not self-intersect.

Hardware Buffers

On the other hand, buffer-based methods use GPU-accelerated structures to hold multi-fragments (even coplanar) per pixel. Extending Z-buffer [20], where the closest-to-viewer fragment is stored, A-buffer (**AB**) [17] was the first method to capture all fragments per pixel in a single geometry pass using variable-length lists per pixel. Yang et al. introduced an actual implementation of A-buffer on the GPU by performing concurrent linked list construction ($\mathbf{AB}_{LL}$) [167]. However, its performance degrades rapidly due to the heavy contention and the random memory accesses when constructing and assembling the entire fragment list, respectively (see Figure 2.6 (b)). A memory-friendly variation of this algorithm was described in [30], where paged per-pixel linked-lists ($\mathbf{AB}_{LL-P}$) improve caching and data bus occupancy.

There have been a few attempts to perform the entire rasterization process using a software graphics pipeline [90, 108]. Liu et al. [90] introduced a complete CUDA-based rasterization pipeline (**FreePipe**) maintaining multiple *unbounded* fragments per pixel in real-time. To supersede pixel level parallelism, Patney et al. [108] extends the domain of parallelization to individual fragments. However, both methods limit user to switch from the traditional graphics pipeline to a software rasterizer. FreePipe has been realized using modern OpenGL APIs ($\mathbf{AB}_{FP}$) [29]. The major limitations of this class are first,

the potentially large and possible wasted memory requirements due to their strategy to allocate the same memory for each pixel (see Figure 2.6 (a)) and second, the necessity of an additional full-screen post-processing pass to sort the captured fragments.

To avoid limitations of constant-size array and linked lists structures, several works organized linearly memory into variable contiguous regions for each pixel as shown in Figure 2.6 (c). However, the need of an additional rasterization step results in performance downgrade when compared to AB_{FP}. Peeper [110] computed buffer offsets for linearising A-buffer storage based on the maintenance of a fragment counter pass and a subsequent prefix sum on the fragment counter data. However, the order of this algorithm depends on the active screen dimensions resulting in a performance downgrade even when rendering sparse scenes in high resolutions. Closest to this thesis lies the *l-buffer* architecture [87] that exploits the sparsity in the pixel space. However, a *serialized* process on the used pixels is performed to compute the memory offsets for each pixel.

Regardless of the data structure, the aforementioned approaches suffer from (i) memory overflows resulting from the unbounded buffer needed to store all generated fragments, and (ii) performance bottlenecks arising when the number of per-pixel fragments to be post-sorted increases significantly. Various techniques have been proposed to simulate the behavior of the A-buffer architecture with reduced memory requirements. *F-buffer* [95] and *R-buffer* [161] replaced the linked list structure with a FIFO buffer to capture all incoming fragments. *Z^3-buffer* [61] set an upper bound for the number of fragments stored per pixel. k-buffer (**KB**) [14, 6] reduced memory requirements capturing the k-closest to the viewer fragments in a single geometry rasterization. However, it is susceptible to disturbing flickering artifacts caused from read-modify-write hazards (RMWH) during fragment insertion updates. Liu et al. [88] extended this work to a multi-pass approach (**KB-Multi**) achieving robust rendering behavior with the trade-off of low frame rates. Moreover, Bavoil and Mayers [105, 8] eliminated most of the memory conflicts by performing *stencil routing* operations (**KB-SR**). This approach avoids RMWH but is incompatible with hardware supported multi-sample anti-aliasing (MSAA) and additional stencil operations. Finally, Zhang [169] explored a *memory-hazard-aware* solution (**KB-MHA**) based on a depth-error correction coding scheme. In practice however, they cannot guarantee correct results in all cases. The image quality of this class of methods is highly dependent on a coarse CPU-based pre-sorting in primitive-space which eliminates the arrival of out-of-order fragments. Conversely, Wang and Xie [158] proposed to partitioning the input scene into components with a few layers and then rendering them individually in order to fit into the limited KB-SR buffer size. However, this comes with the limitation of non-supporting animated scenes and having limited order-dependent applicability.

Multi depth test scheme (**KB-MDT**), developed in both CUDA [90] and OpenGL [96] APIs, guarantees correct depth order results by capturing and sorting fragments on the fly via 32-bit atomic integer comparisons. Since 64-bit atomic operations are not supported by available APIs to update the depth and color buffers simultaneously, a costly additional geometry pass is therefore suggested. Furthermore, noisy images may be generated due

to the lost precision when converting floating depth values of close fragments. Recently, Salvi [121] extended original k-buffer to avoid fragment racing by employing hardware-aware pixel synchronization (**KB-PS**). However, this method is currently compatible only with graphics cards based on the Haswell architecture.

Finally, Yu et al. [168] proposed two linked-list-aware solutions to accurately compute the k-foremost fragments. The idea of the first one is to capture all fragments by initially constructing AB_{LL}, followed by a step that selects and sorts the k-nearest fragments (**KB-AB**$_{LL}$). On the other hand, the second approach directly computes depth-ordered per-pixel linked lists avoiding the unnecessary A-buffer construction (**KB-LL**). Despite the fact that the second approach requires less storage, fragments are sparsely stored in graphics memory causing the additional allocation of contiguous blocks of memory.

Finally, readers may refer to a comprehensive survey [97] for a detailed description of the pros and cons in terms of memory and performance of many of the aforementioned alternatives.

Rendering Effects

Interior Visualization. Visualising the interior of solids has been attempted using clipping, wireframe drawing, capping and volume and transparency rendering. *Hidden edges* and *silhouettes* may be overlaid with a shaded rendering of the visible surfaces [119]. *Wireframe rendering* has been used prior to graphics hardware rasterization techniques. Baerentzen et al. [5] presented an efficient approach to wireframe drawing combined with antialising and hidden line removal. *Surface transparency* and depth ordering [8] may be used to modulate the color based on the translucent surfaces seen through a pixel. *Volume rendering* may be used to modulate color based on the thickness of solid layers traversed by a ray from the viewpoint [159]. *Clipping* is commonly used for inspecting the interior of solids (see Figure 1.6(c)). A portion of the boundary may be removed using clipping planes or solids, exposing hidden surfaces and potentially back-facing portions of the boundary. Because such images may appear confusing to the casual viewer depending on the illumination and the complexity of the interior, *capping* [117] enables the user to see portion of non self crossing manifold cross hatching the sliced (clipped) parts (see Figure 1.6(d)). Nooruddin and Turk [107] introduced a method for automatically classifying and visualizing the interior of polygonal models. In this work, a polygon is characterized as interior if it is not visible by any viewing directions started from a point outside the model.

Self-trimming. Static interior classification of self intersecting curves has been studied extensively in 2D (see for example [50]). Suzuki et al. [141] presented a technique for creating "1-simple" interior surfaces from dual cycles containing at most a single pair of self intersecting curves. This static technique treats an interesting albeit restricted class of surfaces. Samoilov and Elber [122] presented two techniques for eliminating self-intersections in planar curves. The authors introduced sophisticated techniques for eliminating self-

intersections demonstrating the importance of having rendering alternatives other than modifying the geometry to avoid self-intersections. Finally, this work offered a promising higher dimension perspective for examining and classifying 2D curves. When the bounding loops of several connected regions of the plane evolve over time and self-overlap, one may wish to decide automatically which region should be visible at each pixel. The union of the boundaries of these regions decomposes the union of the regions into cells. A valid association of each cell with a unique region must satisfy constraints that ensure that the arrangements can be realized by placing and interleaving physical cut-out regions on a plane [98].

The problem becomes more complex when these regions are connected, and hence can no longer be differentiated. In particular, several authors have addressed the problem of defining the interior of self-crossing curves of a particular class, which may be characterized by stating that the curve is the boundary of a topological disk that does not contain any fold-over (i.e. each portion of that disk is front facing). The issue of deciding whether a self-crossing curve is in this class and whether it is the projection of the boundary of a front-facing disk in 3D has been discussed by [34]. A simple approach that works well in practice has also been proposed by [103]. Figure 1.6(b) illustrates the boundary that should be trimmed away (painted in pink) of a self-crossing surface.

Direct CSG Rendering. A complex solid may often be designed as a Boolean combination of other solids. The design sequence may be captured as a CSG representation [114] that defines a Boolean expression with union, intersection, and difference operators and with primitive solids as operands. In early solid modeling systems [154] the primitives were restricted to simple quadrics (block, sphere, cone, cylinder and torus). Computing the boundary of solids defined in CSG with more general primitives (triangle meshes, NURBS, subdivision surfaces) is computationally expensive and numerically delicate. To address this problem, several screen-based techniques have been proposed for rendering CSG models directly on the GPU by classifying surfels against each primitive. These techniques consider, for each pixel, the set of candidate surfels (surface points) on the boundary of each layer of each primitive that project onto the pixel. They use several passes to generate layers of these candidate surfels. For each layer, they classify the surfels by trimming them against each primitive and combine these trimming results to decide which surfels are on the boundary of the CSG solid and which of these are visible.

Some approaches [35, 40, 57] use a disjunctive form (union of intersections) formulation of the Boolean expression to simplify trimming. A surfel lies on a product if it is the furthest of the front facing surfels at that pixel and if it lies in front of all back facing surfels at that pixel [57]. Unfortunately, the number of intersections (products of a disjunctive form) may grow exponentially with the number of primitives. To avoid this complexity exposure, Constructive Solid Trimming (CST) [44] trims the boundary of each primitive against the Blist [43] of its active zone [120], which defines the solid where the boundary of the primitive contributes to the boundary of the solid. Recently, Rossignac showed that

6 stencil bits per pixel suffice for rendering arbitrarily complex CSG models, by providing a linear cost algorithm that swaps left and right operands of the Boolean expression of n literals so that it may be evaluated using $O(\log \log n)$ space [115]. The boundaries of the CSG primitives may be defined as subdivision surfaces [59], which reduces the cost of transferring detailed geometry to the GPU. Recently, Zhao et al. [170] introduced a fast hardware assisted method for approximately reconstructing CSG results using rasterized views. However, these methods are not capable of handling self-trimmed surfaces as these shown in Figure 1.6(e,f).

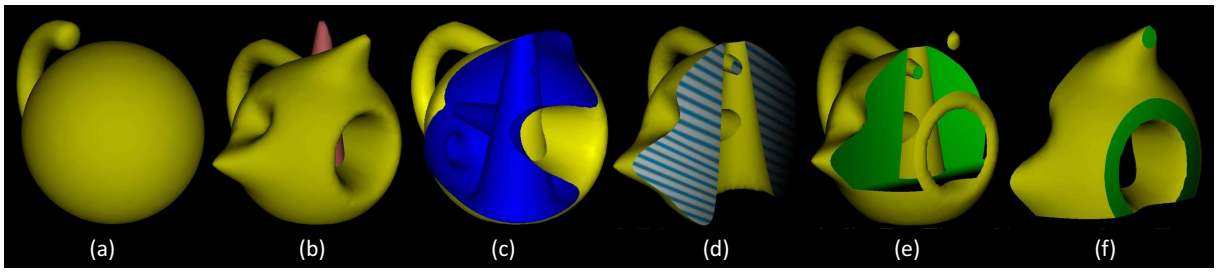


Figure 1.6: (a) The initial non self crossing surface. (b) After applying a set of concurrent disjoint deformations on (a), we use pink to illustrate boundary that should be trimmed away. (c) Rendering the same self trimmed surface using clipping and (d) using capping. (e and f) Rendering of Boolean combinations of (b) with an orthogonal parallelepiped.

1.4 Thesis Contributions

This dissertation deals with four significant problems: (i) segmentation and (ii) skinning of animated meshes and (iii) real-time rendering and trimming of (self)-intersecting deformable surfaces using (iv) multi-fragment rendering techniques.

Transforming an MA into a skinning representation is mostly guided by a deformation-driven segmentation to distribute the skeletal joints and determine the influencing vertex weights. However, current segmentation and skinning methods cannot adapt the final result when the mesh sequence is being modified [19]. To this end, we investigate a pose-to-pose scheme for segmentation and skinning MAs which enables the support of editing operations at arbitrary animation frames.

Correct detection and trimming of (self)-intersecting animated surfaces in the geometry-space is an extremely costly operation. Finding the intersection curves and computing the final trimmed mesh make it unsuitable to support real-time performance during deformation animations or interactive editing operations. To this end, we investigate classification rules to guide efficient collision detection and trimming in the image space, realized through a multi-fragment rasterization framework, without computing any geometry-interpenetrations between (individual) deformable objects. Since capturing multiple fragments efficiently on the GPU is a challenging task in terms of time, space and robustness [97], we study the multi-fragment rendering problem in various perspectives:

- reduction of *fragment-contention*: performance of hardware-accelerated A-buffer [167] degrades rapidly in cases where heavy access on the GPU shared memory is necessary.

- elimination of *z-fighting* artifacts: multi-pass rendering methods [36, 8] are susceptible to noisy rasterization when two or more fragments have identical depth values.

- avoidance of *fragment-overflow*: a class of A-buffer variants [90] utilize unbounded data structures resulting in potentially large and possible wasted memory requirements.

The structure of the rest of the thesis is as follows. In Chapter 2, we provide the important preliminaries regarding the problems of segmenting and skinning MAs, and the background of multi-fragment rendering frameworks. A brief discussion of the graphics rendering pipeline is also offered. Finally, we include a detailed overview of the related state-of-the-art approaches.

In Chapter 3, we present a generic framework for efficiently generating multi-resolution segmentations that account for both articulated and highly-deformable MAs. Based on the observation that only a limited part of the surface region is modified from frame to frame, we built an over-segmentation by combining precomputed *per-pose partitionings*. The desired segmentation resolution is dynamically chosen by the user applying a fast refinement process which aims at cleaning "noisy" segments created when successive partitionings are merged. A temporally-coherent edge collapsing solution is offered to refine the segmentation graph guided by an area-aware criterion. A smooth clustering transition from frame to frame is further offered providing perceptual consistency between consecutive poses. Finally, we have included extensive comparative results with respect to computation cost and skinning quality. This work, "*Pose Partitioning for Multi-resolution Segmentation of Arbitrary Mesh Animations*", has been accepted to be presented at Eurographics 2014 [152].

In Chapter 4, we introduce a novel *pose-to-pose* skinning technique that aims at preserving temporal coherence. This scheme results in reducing the approximation error and further supporting arbitrary pose editing. Editing can then be smoothly propagated at the subsequent frames generating new deforming mesh sequences without altering the skinning representation. Although fitting is performed from pose to pose, a reproduction scheme from the rest pose to an arbitrary pose can be produced efficiently. Finally, we present optimization techniques yielding results comparable to previous approaches in terms of accuracy and performance. This work, titled "*Pose-to-Pose Skinning of Animated Meshes*", had been accepted as poster at SCA 2011 [145].

In Chapter 5, we introduce an efficient and memory-friendly GPU-accelerated A-buffer architecture for multi-fragment rendering. S-buffer organises memory into variable contiguous regions for each pixel by exploiting fragment distribution for precise allocation of the needed storage and pixel sparsity for computing the memory offsets for each pixel in a parallel fashion. An experimental comparative evaluation of the proposed technique

over previous multi-fragment rendering approaches in terms of memory and performance is provided. This work, "*S-buffer: Sparsity-aware Multi-fragment Rendering*", had been accepted as short presentation at Eurographics 2012 [149].

In Chapter 6, we introduce image-based coplanarity-aware algorithms for reducing (may miss fragments but are usually faster), eliminating (guaranteed to extract all fragments) and highlighting z-fighting flaws in scenes suffering from coplanar geometry. We adapt previously presented single-pass and multi-pass rendering methods, providing alternatives for both commodity and modern graphics hardware. Inspired by occlusion culling, an optimization is further introduced improving the performance when multi-pass rendering is performed on multiple deformable objects. We present quantitative and qualitative results with respect to performance, space requirements, and robustness. Visual output is further provided illustrating the effectiveness of our variants over the conventional methods for a number of Z-fighting sensitive applications. This work, titled "*Depth-fighting Aware Methods for Multifragment Rendering*", has been published in TVCG journal [150]. An early version of this research, titled "*Z-fighting aware Depth Peeling*", had also been accepted as poster at SIGGRAPH 2011 [148]

In Chapter 7, we introduce k$^+$-buffer, an improved GPU-accelerated k-buffer framework, which handles memory-hazards and depth precision conversion artifacts, and avoids geometry pre-sorting and the requirement for unbounded memory. Two GPU-accelerated data structures have been developed: the max-array and the max-heap. These bounded-memory data structures maintain accurately the k-foremost fragments per pixel in a single geometry pass avoiding memory-overflow. A consecutive geometry rendering may be executed to enable precise memory allocation. Extensive experimental comparison demonstrates the superiority of our framework as compared to previous k-buffer alternatives with respect to storage requirements, performance and image quality. This work, "*k^+-buffer: Fragment Synchronized k-buffer*", has been accepted for presentation at I3D 2014 [151].

In Chapter 8, we present rendering algorithms for (self)-crossing surfaces and in particular we explore static and dynamic semantics utilized to perform efficiently trimming, clipping, capping and boolean operations. Fast collision detection can further be realized using this framework. This classification and rendering is accomplished in real-time through a rasterization process without computing any self-intersection curve, and hence is suited to support complex MAs. We further adapt several state of the art multi-fragment techniques to achieve efficient rendering of self-trimmed surfaces and we provide comparative results in terms of time and memory requirements. This work, titled "*Direct Rendering of Boolean Combinations of Self-Trimmed Surfaces*", has been published in CAD journal [116].

Finally, Chapter 9 provides an overall review of the results of our research and indicates limitations and interesting directions for future work.

Chapter 2

Background Material and State-of-the-art

The purpose of this chapter is twofold: first, we recapitulate the mathematical and graphical tools that will be important in the following chapters. Second, we present a broader body of related work, discussing also classification rules and alternative approaches to skinning mesh animations and multi-fragment rendering.

2.1 (Self-intersecting) Mesh Geometry

Let M be a three dimensional (3D) boundary surface mesh of fixed connectivity represented by a graph $M = (V, E, F)$, where $V = \{v_i | v_i \in \mathbb{R}^3, 0 \leq i < n\}$ is the set of vertices ($|V| = n$), $E = \{e_{ij} = (v_i, v_j) | v_i, v_j \in V, i \neq j\}$ is the set of edges, $F = \{f_{ijk} = (v_i, v_j, v_k) | v_i, v_j, v_k \in V, i \neq j, i \neq k, j \neq k\}$ is the set of faces. A normal vector n_i at each vertex v_i is computed as the normalized average of the surface normals of the faces that contain that vertex. Each face in a mesh has a perpendicular normal vector. A vertex in a closed surface is *visible* only if it is *front-facing*: the surface in this vertex is oriented towards the observer. If $\cos(\theta) > 0$ the vertex is front-facing (otherwise is *back-facing*), where θ equals to the angle between the normal vector n_i and the vector representing the viewing direction. A *scene* is a collection of meshes compromising everything that is included in the environment to be rendered.

Formally, a surface M is *manifold* when it is a compact and orientable two-manifold without boundary. We say that M is *self-crossing* when its immersion contains non-manifold self-intersection edges where M passes through itself, as shown in Figure 1.2. Hence, two or more different points of M coincide at each point of a self-crossing curve of the immersion. We say that M is a *boundary* when M is the boundary of some solid (closed-regularized point set) that we denote $I(M)$ and call the *interior solid* of M. For a set of vertices V, the *minimum bounding box* refers to the 3D box with the smallest measure volume within which all the points lie.

Consider an initial manifold boundary M that is not self-crossing and a continuous process D_t that deforms this surface while keeping it an immersed sub-manifold. If p_t denote the instance (*pose*) $D_t(M)$ of the deformed surface at time t then $A = (p_0, p_1, p_2, \ldots, p_k)$ denotes the animation sequence with k animation surface poses $p_1, p_2, \ldots, p_k$ and a *rest-pose* p_0, and v_j^t denotes the j-th vertex of pose p_t.

Definition 2.1. Let $A = \{p_t = (V_t, E_t, F_t), t = 1, \ldots, k\}$ be a mesh sequence: V_t is the set of vertices of the $D_t(M)$ deformed mesh of the sequence, E_t its the set of edges and F_t its set of faces. If the connectivity is constant over the whole sequence, that is to say if there is an isomorphism between any E_i and E_j, $1 \leq i, j \leq k$, then MA is called a temporally coherent mesh sequence (TCMS). Otherwise, MS is called a temporally incoherent mesh sequence (TIMS).

Assume that p_t is self-crossing, then we say that p_t is a *Self-Crossing Surface* (**SCS**), i.e., a compact, immersed, and orientable surface with transverse self-intersections. Different semantics have been introduced for defining the solid that M represents, which we call its interior $I(p_t)$, and hence also its *trim* $T(p_t)$, which is the subset of p_t that is the boundary of $I(p_t)$. We say that $T(p_t)$ is a *Self-Trimmed Surface* (**STS**). The interior along with the STS define a manifold or a non-manifold object.

An SCS M partitions the 3D space W into open full dimensional components C_i (i.e., the maximally connected components of $W - M$), one of which is *infinite* (denoted here by C_0). Each component is classified as either *interior* (also denoted as **in**), i.e., part of

the interior of the solid, or *exterior* (also denoted as **out**). The solid $I(M)$ represented by M is the closure of the union of all interior components. To obtain a bounded solid, C_0 should not be included, and hence is classified as out. Other components may be classified as in or out, depending on the chosen rule. The trim, $T(M)$, is the boundary of $I(M)$. Hence, trimming amounts to discarding portions of M that separate either two interior or two exterior components.

2.1.1 Classification Rules

Various rules (semantics) may be used to associate a solid $I(M)$ with an SCS M. One may conceive interesting rules that compute new bounding surfaces for solid $I(M)$ (for instance by using the convex hull of S or a visibility graph). Here, we focus on rules that have the *boundary diminishing* property, which states that the boundary $T(M)$ of $I(M)$ must be a subset of M. Note that this property is satisfied by Boolean and regularized Boolean operations [143, 114].

In 2D, the *index* (also called *winding number*) $w(p, C)$ of an oriented, closed-loop, self-crossing curve C around a given point p that is not on C is an integer representing the total number of times the curve travels counter-clockwise around the point. The winding number depends on the orientation of the curve, and is by convention negative if the curve travels around the point clockwise. All points in a given component (maximally connected component of the complement of C have the same winding number. The infinite component has a winding number of 0. One may easily keep track of the winding number by propagating it from one component to an adjacent one. Crossing the curve once increments or decrements the winding number, depending on the orientation of the curve relative to the direction of the crossing.

In three dimensions, the index $w(p, M)$ of a point p with respect to an SCS M may be defined as follows: We assume that p is not on M. Consider any path P from infinity to p. Let k_i be the number of times that P enters M (i.e., crosses the boundary in a direction opposite to the outward normal) and k_o be the number of times that P exits M (i.e., crosses the boundary in a direction confluent to the normal). Then, $w(p, M) = k_i - k_o$. Figure 2.1 shows an SCS cross section (green self-crossing polyline) with the triplet $(k_i, k_o, w(p, M))$ indicated for each area, where k_i and k_o were computed from the left. Note that points of the same component may have different k_i and k_o (for example when counting from the bottom) but they have the same index. Note that in situations where P simultaneously crosses several neighborhoods of M, the crossing of each neighborhood must be accounted for separately. For conciseness, we denote $w(p, M)$ by $w(p)$ or by simply w. Heisserman [50] provides an equivalent definition of the index (winding number) as the number of times the surface encloses a point.

The index is the signed generalization of the *overlap count* which is defined as the unsigned count of the number of fragments (depth, color, and normal values of M associated with points of T that project on a pixel center) that are generated by rendering scenes where $I(M)$ is *capped* [94, 117] and combined with other shapes through *CSG*

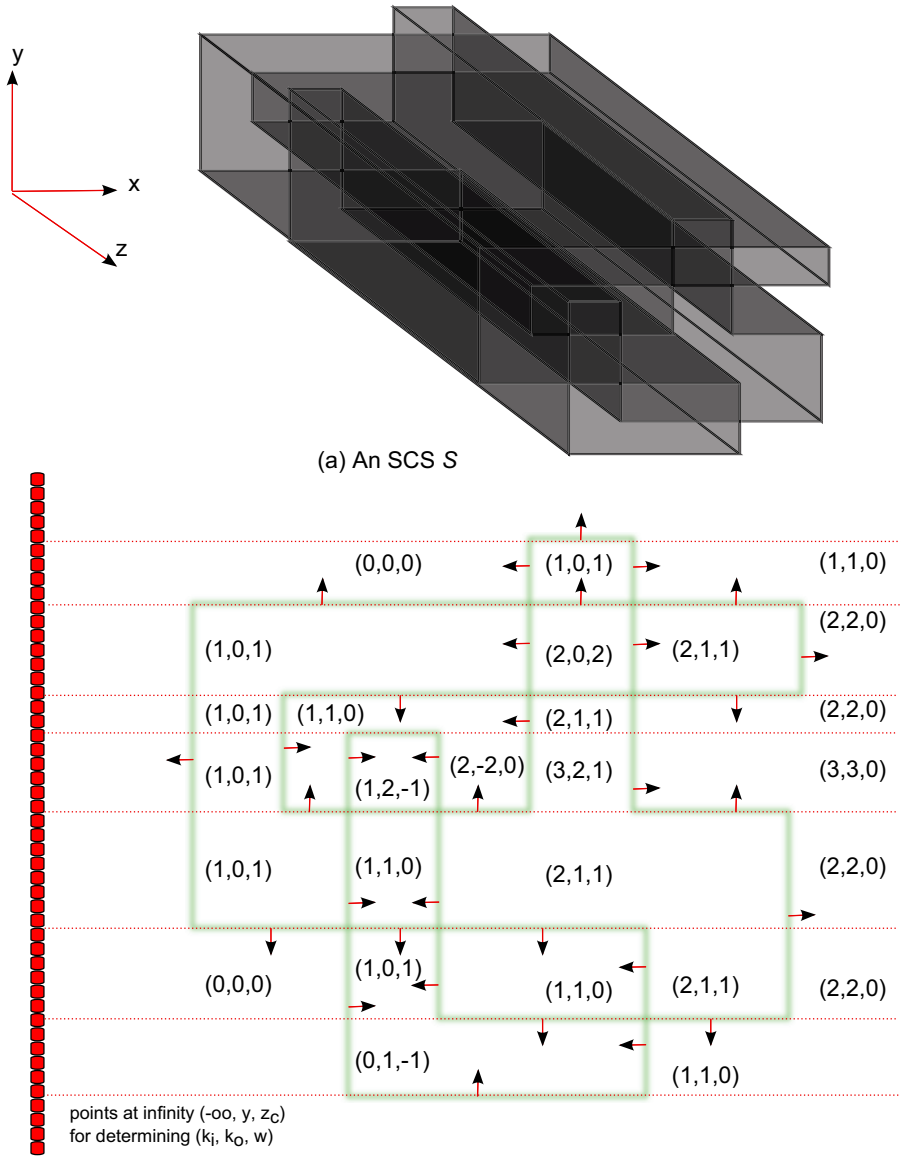


Figure 2.1: (a) An SCS M. (b) A cross section of M on plane $z = z_c$ illustrated with thick green transparent line with normal vector orientation marked. We indicate the values of the triplets (k_i, k_o, w) using $(-\infty, y, z_c)$ as point at infinity for determining the characterization for a point $p(x, y, z_c)$. The horizontal dashed red rays originating from the points at infinity along with the green polyline partition the plane into areas with the same triplet value. Note that k_i, k_o depend on the selection of the point at infinity, whereas the winding number is independent of this choice.

operations [43, 44]. Capping returns the intersection of $I(M)$ with a 3D region R that is the intersection or the union of (usually) linear half-spaces and displays the caps, i.e., the portions of the boundary of R in *I(S)*. Note that the index is in general not equal to the overlap count, which is defined at a point p as the number of times a specific ray from p to infinity hits the surface. When the ray is aimed at the viewpoint, the overlap count may be computed on the GPU for each pixel. Note that the parity of the overlap count is independent of the direction of the ray, identical to the parity of the index, and constant throughout a component.

We can view self-trimming classification rules as a generalization of Boolean operations. Consider the simple situation of Figure 2.2 (top) where an elongated initial zero-genus shape I bounded by S is deformed to self-penetrate. One could split I into two disjoint parts I_1 and I_2, deform one of them, and define the desired result as $I = I_1 \cup I_2$ which has genus 1. Hence, in this case, the self-trimming, which takes the deformed S and produces the boundary T of I is an extension of Boolean union. Now, consider the example of Figure 2.2 (bottom) where the initial flattened zero-genus shape is deformed so that S crosses itself. Here, one could decompose the complement of the initial shape in two disjoint parts, deform them, union them, and take the complement to obtain I. De Morgan's laws state that $!(!A \cup !B) = A \cap B$. Hence, we suggest that this operation is an extension of the Boolean intersection. It produces a set that also has genus one.

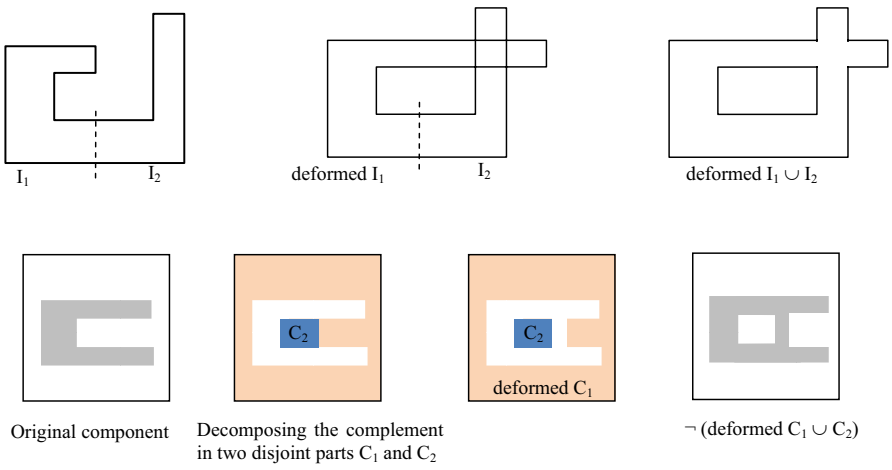


Figure 2.2: Self trimming as generalization of Boolean operations.

2.2 Mesh Segmentation

Using a subset of elements from the faces F, edges E or vertices V, an induced sub-mesh $M' \subset M$ can be created as follows. Let S be the set of mesh elements which is either V, E or F. Let $S' \subset S$ be a subset of mesh elements, and let V' be the set of all vertices which are included in (or are) the elements in S'. A sub-mesh M' is defined as the mesh $M' = \{V', E', F'\}$, where $E' = \{(v_i, v_j) \in E | v_i, v_j \in V'\}$ are all edges in which

both vertices are a part of V', and $F' = \{(v_i, v_j, v_k) \in F | v_i, v_j, v_k \in V'\}$ are all faces in which all vertices are a part of V'. The basic definition of a static mesh segmentation is therefore [130]:

Definition 2.2. Let M be a 3D mesh surface, and S the set of mesh elements which is either V, E or F. A *static segmentation* (also named as *partitioning* or *clustering*) of M is the set of sub-meshes $C(M) = \{C^0, \ldots, C^{l-1}\}$ induced by a partition of M into l disjoint subsets (clusters).

As can be seen, S can either be the vertices V, edges E or faces F of the mesh M and the partitioning of S induces a segmentation of M. Segmentation algorithms usually partition the faces of the mesh (i.e. $S = F$), some partition the vertices ($S = V$), and few the edges ($S = E$). Note that if $S = V$ or $S = E$ then some faces (that include vertices from different parts of S) will not be part of any sub-mesh C^j , and must be joined to one of the adjacent parts.

The key question in all mesh segmentation problems is how to partition the set S. Obviously, this relies heavily on the application in mind. Shamir [130] poses the segmentation problem as an *optimization problem* by defining a criterion function of the partitioning of M, $J : 2^M \rightarrow \mathbb{R}$ for each application in the following manner:

Definition 2.3. Given a mesh M and the set of elements $S \in \{V, E, F\}$, find a disjoint partitioning of S into $C^0, \ldots, C^{l-1}$ such that the criterion function $J = J(C^0, \ldots, C^{l-1})$ be minimized (or maximized) under a set of *constraints*.

A number of geometric attributes and partitioning criteria are commonly used by many segmentation techniques to define the optimization function. The decision which attribute to use has a significant effect on the segmentation result and is strongly linked to the segmentation aim. The main goal when segmenting a MA is to partition the animated mesh into regions with similar vertex motion characteristics. A *pose partitioning feature* is computed from an animation pose p_t and a reference pose. While the rest (p_0) or the previous pose (p_{t-1}) is commonly used as the reference, an alternative approach is to use an averaged rest shape [48]. In this thesis, we have utilized *rotation angle distance* extracted from *deformation gradients* as primary segmentation measure.

In contrast to single mesh segmentation that consists in grouping mesh vertices into spatial regions, the segmentation of a MA can have various interpretations with respect to time and space [4]:

Definition 2.4. A *coherent segmentation* is a set of partitionings (clusterings) $C_t(M) = \{C_t^0, \ldots, C_t^{l_t-1}\}$ of each animate pose p_t of 3D mesh M, such that:

- the number l of sub-meshes is the same for all segmentations: $l_i = l_j \forall i, j \in [0, k]$.
- there is a one-to-one correspondence between sub-meshes of any two meshes.
- the connectivity of the segmentations is preserved over the sequence.

Definition 2.5. A *variable segmentation* is a set of partitionings $C_t(M) = \{C_t^0, \ldots, C_t^{l_t-1}\}$ of each animate pose p_t of 3D mesh M which is not a coherent segmentation.

A coherent segmentation, or simple segmentation, of a mesh sequence can be thought as a segmentation of some mesh of the sequence which is mapped to the other meshes. On the other hand, a variable segmentation solves the problem of clustering data over time, maintaining simultaneously two conflicting criteria: (a) remain faithful to the past-data and (b) effectively altered when moving on to future data. This application is helpful to detect motion changes at each time step.

2.2.1 Deformation Gradient

Deformation gradient is a quantity that encloses both shape and orientation of the deformation. The affine transformation Φ_f of the f_{ijk} triangle of mesh M that contains vertices $v \in \{v_i, v_j, v_k\}$ is given by [140]:

$$\Phi_f(v) = \mathrm{X}_f \cdot v + \mathrm{T}_f$$

where X_f is a 3×3 transformation matrix which contains the rotation, scaling and skew components of the deformation and T_f the translation component. The deformation gradient of the triangle between its status in a pose p_t and a reference pose p_r is enclosed in the Jacobian matrix:

$$D_p\Phi_f(v) = \mathrm{X}_f$$

Note that the three vertices of a triangle (e.g $\{v_1, v_2, v_3\}$) are not enough to describe the deformation towards the direction perpendicular to the triangle. To alleviate this, a fourth vertex (v_4) is introduced in the direction of the triangles normal vector, with length proportional to the edges of the triangle [138]:

$$v_4 = v_1 + \frac{(v_2 - v_1) \times (v_3 - v_1)}{\sqrt{|(v_2 - v_1) \times (v_3 - v_1)|}}$$

The affine transformation of each facet is then described by:

$$\mathrm{X}_f \cdot v_i^j + \mathrm{T}_f = v_i^0,\ 1 \leq i \leq 4,\ 1 \leq j \leq k$$

and in matrix form it is given by:

$$\mathrm{X}_f[v_2^j - v_1^j\ v_3^j - v_1^j\ v_4^j - v_1^j] = [v_2^0 - v_1^0\ v_3^0 - v_1^0\ v_4^0 - v_1^0] \Leftrightarrow \tag{2.1}$$

$$\mathrm{X}_f = [v_2^0 - v_1^0\ v_3^0 - v_1^0\ v_4^0 - v_1^0][v_2^j - v_1^j\ v_3^j - v_1^j\ v_4^j - v_1^j]^{-1} \tag{2.2}$$

Both rotation and stretch information can be extracted from X_f by applying polar decomposition [132]. Performing thin singular value decomposition (tSVD) upon X_f we get:

$$\mathrm{X}_f = \mathrm{U}\Sigma\mathrm{V}^{\mathrm{T}} = (\mathrm{U}\mathrm{V}^{\mathrm{T}})(\mathrm{V}\Sigma\mathrm{V}^{\mathrm{T}}) = \mathrm{R}_f\mathrm{S}_f$$

where where $\mathrm{R}_f \in \mathbb{R}^{3\times3}$ is the orthogonal matrix representing the rotational component, and $\mathrm{S}_f \in \mathbb{R}^{3\times3}$ is a symmetric matrix that applies stretching to the triangle before the rotation. If the Euler angle θ_f is not a multiple of π, the angle can be computed from the elements of the rotation matrix R_f as follows:

$$\theta_f = \arccos\left(\frac{1}{2}(\mathrm{R}_f[1,1] + \mathrm{R}_f[2,2] + \mathrm{R}_f[3,3] - 1)\right)$$

Finally, the rotation angle per vertex θ_v is computed as the number average of the angles θ_f of the surrounding facets. Figure 2.3 shows a segmentation example using as partitioning criteria the average rotation angles from all poses for a highly-deformable flapping flag.

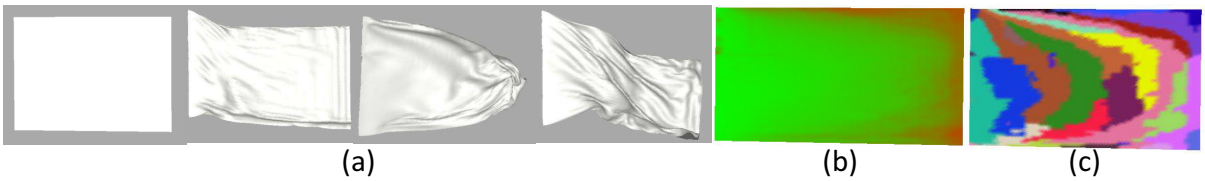


Figure 2.3: (a) Highly deformable animation of a flag under the influence of wind, (b) normalized rotation angles computed from the deformation gradients (c) resulted segmentation (each segment is painted with different color).

2.3 Mesh Skinning

In general, skeletal animation could theoretically refer to any technique that computes shape of the skin for a given skeletal posture. However, the term skeletal animation is usually used only when there exists a direct geometric relationship between the skeleton and skin. In our case, the deformation structure, i.e., the parameters controlling the deformation, is simply a list of 3D transformations. Traditionally, these transformations describe the actual position and orientation of the joints in the animated skeleton. However, as has been pointed out [55], no skeleton is actually required in skinning, the transformations alone are sufficient.

Following Kavan's thesis [66], we use the term *skinning* to refer to a skin deformation algorithm based on

1. a reference surface mesh M: any 3D object representing the rest-pose (undeformed) model.

2. a list of B individual rigid or affine transformations $M_1^t, \ldots, M_B^t$: represents the deformation from the rest-pose p_0 to a new pose p_t. Each transformation M_j^t influences a surface part of the mesh M.

3. two lists of vertex indices and weights that describe which transformations influence which part of the mesh M: For each vertex v_j we have two lists of length $l \leq B$. The

lists, $i_{1,j}, \ldots, i_{l,j} \in [1, B]$, and $w_{1,j}, \ldots, w_{l,j} \in (0, 1]$, are sequences of transformation indices and non-zero weights that influence vertex v_j, respectively. Weight $w_{b,j}$ describes the amount of influence of transformation $M_{i_{b,j}}$ on vertex v_j. Weights $w_{1,j}, \ldots, w_{l,j}$ must be convex, i.e., besides non-negativity, they are required to satisfy $\sum_{b=1}^{l} w_{b,j} = 1$. Practically, it has been proved that $l = 4$ influences per vertex are adequate for a satisfactory result.

Describing the movement of highly deformable objects requires the use of affine transformation matrices, to capture deformations other than rotation and translation. Rigid body motion can be used but not without penalty in the quality of the approximation, since there is no guarantee that the deformation of a vertex is purely rigid.

When skinning is used in conjunction with a hierarchy of transformations (e.g., a skeleton), then $M_1^t, \ldots, M_B^t$ are rigid transformations corresponding to individual joints (nodes of the skeletal tree). In this case, the rigid transformation M_j^t represents rotation and translation of joint j from the rest-pose to the current (animated) posture p_t. The indices $1, \ldots, B$ of transformations $M_1^t, \ldots, M_B^t$ are often called *bones* or *joints* (*proxy-bones* or *proxy-joints* if no actual skeleton is present). Note that both the rest-pose mesh M, the number of joints B as well as the vertex binding (indices and weights) are constant; only the individual transformations $M_1^t, \ldots, M_B^t$ vary during an animation A with k poses ($1 \leq t \leq k$).

In the following sections, we describe the most widely used character skinning techniques, (i) linear blend skinning and (ii) dual quaternion skinning, due to its computational efficiency and straightforward implementation in graphics hardware [9, 82, 67]. A small discussion is also offered for a simpler skinning model which is equipped only with rigid skinning, which means that vertices are influenced by either 0 or 1 joints.

2.3.1 Rigid Skinning

In *rigid skinning* [45], every motion is transferred to the mesh by assigning to each vertex v_j one bone as driver ($w_{l,j} = 1,\ l \leq 1$). A skin vertex is pinned in the local coordinate system of the corresponding bone, repeating whatever motion this bone experiences. Mathematically, its deformed position from rest-pose p_0 to pose p_t is computed as $v_j^t = M_1^t v_j^0$. Despite its simplicity and computational efficiency, rigid skinning makes it difficult to obtain sufficiently smooth skin deformation in areas around the joints. Vasilakis and Fudos [146] introduced novel blending patch construction techniques that approximately preserve the initial volume of the joint and ensure mesh robustness. Briefly, this approach first removes the skin vertices of the overlapping component parts and then adds new vertices to fill in the gap. Finally, it constructs a blending mesh that produces a smooth surface using a robust triangulation method. To achieve real-time performance they further developed a full GPU realization of their entire skinning algorithm [147].

2.3.2 Linear Blend Skinning

Linear Blend Skinning (LBS) was initially presented in the game development community [78, 79]. With LBS, we assume that $M_1^t, \ldots, M_B^t$ are represented by homogeneous matrices $\in \mathbb{R}^{4\times4}$:

$$\begin{pmatrix} n_{00} & n_{01} & n_{02} & t_0 \\ n_{10} & n_{11} & n_{12} & t_1 \\ n_{20} & n_{21} & n_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the matrix $N = \{n_{ij}, 0 \leq i, j \leq 2\}$ is the original linear transformation and $T = (t_0, t_1, t_2)^{\mathrm{T}}$ represents a translation vector. The deformed vertex position is determined by a weighted combination of the transformed positions influenced by all its assigned bones. Mathematically the LBS algorithm is expressed as the following equation:

$$v_j^t = \sum_{b=1}^{l} w_{b,j} M_{i_{b,j}}^t v_j^0 \Leftrightarrow \tag{2.3}$$

$$v_j^t = (\sum_{b=1}^{B} w_{b,j} M_b^t) v_j^0, \;\; j = 0, \ldots, n-1 \tag{2.4}$$

This technique naturally suffers from inherent flaws caused by elongations and interpenetrations especially in areas around joints since linear blending of rigid transformations does not result in rigid motion.

2.3.3 Dual Quaternion Skinning

Several alternatives have appeared modifying the standard skinning formula, aiming at expanding the space of possible deformations. Numerous methods have replaced the linear blending domain of rigid transformations based on the observation that curved vertex trajectories generate more natural deformations than straight lines. Quaternions and dual quaternions offer an alternative representation for rotation and rigid body transformations, respectively. The restriction of representing translations on a compact skinning framework with classical quaternions [51] solved by a novel algorithm based on dual quaternions (DQS) [68] to describe the motion of rigid bones. Similar to the way that rotations in 3D space can be represented by quaternions of unit length, rigid motions in 3D space can be represented by dual quaternions of unit length. A dual quaternion is an ordered pair of quaternions $\hat{Q} = (Q_A, Q_B)$ and therefore is constructed from eight real parameters. A 3D vertex v_j can be represented by a dual quaternion $\hat{v}_j$, where $\hat{\cdot}$ denotes a quantity expressed as a dual quaternion. If $\hat{M}_1^t, \ldots, \hat{M}_B^t$ denote the dual quaternions that represent the transformations of joints $1, \ldots, B$, the dual quaternion vertex $\hat{v}_j^0$ in the rest-pose p_0 is deformed to a pose p_t by multiplying it with the weighted dual quaternion

result ($\sum_{b=1}^{B} w_{b,i}\hat{M}_b^j$) from the right and with its conjugate inverse from the left:

$$\hat{v}_j^t = \sum_{b=1}^{B} w_{b,j}\hat{M}_b^t \cdot \hat{v}_j^0 \cdot \overline{\left(\sum_{b=1}^{B} w_{b,j}\hat{M}_b^t\right)}^{-1}, \; j = 0, \ldots, n-1 \tag{2.5}$$

When compared to matrices, dual quaternions appear to be advantageous in terms of required storage and in composition of transformations. However, they require considerably more operations to apply the transformation. For complete quaternion and dual quaternion tutorials readers are referred to [99, 47, 67].

2.3.4 Skinning Mesh Animations

Approximating an animation sequence carried out through linear and non-linear skinning processes to produce a more succinct representation is common in both cases of articulated and highly deformable meshes. In general, knowing the transformation that each bone undergoes in each pose and the amount of influence of each bone to the vertices of the mesh, we can approximate the whole animation sequence using only the rest pose. A global formulation for the problem of skinning approximation supporting both skinning methods can be stated as minimizing:

$$\sum_{j=0}^{n-1} \left\|\mathbf{v}_j^t - v_j^t\right\|^2, \; t = 0, \ldots, k-1 \tag{2.6}$$

where $\mathbf{v}_j^t, j \in [0, \ldots, n), t \in [0, \ldots, k)$ correspond to the original n vertex positions of the input animation sequence.

The E_{RMS} error metric proposed by [69] is a widely used metric to measure the mean skinning approximation error of the animation sequence:

$$\mathrm{E}_{RMS} = 1000 \cdot \frac{\|A - A'\|_F}{3nk} \tag{2.7}$$

where A and A' are $3k \times n$ matrices that contain the original ($\{\mathbf{v}_j^t, j \in [0, \ldots, n), t \in [0, \ldots, k)\}$) and skinned approximated ($\{v_j^t, j \in [0, \ldots, n), t \in [0, \ldots, k)\}$) coordinates of each vertex throughout the animation sequence, respectively.

2.4 Graphics Rendering Pipeline

This section presents what is considered to be the core component of real-time graphics, namely the *graphics rendering pipeline* [1]. The main function of this pipeline is to generate (*render*) a 2D image given a virtual camera, 3D objects, light sources, textures and more. The process of using this pipeline is depicted in Figure 2.4. The locations and shapes of the objects in the generated image are determined by their geometry, the characteristics of the environment, and the placement of the camera in that environment.

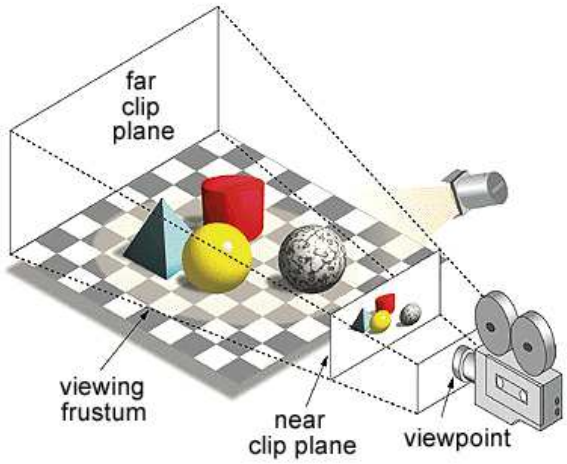


Figure 2.4: In 3D graphics, objects are created on a 3-dimensional stage where the current view is derived from the camera angle and light sources, similar to the real world. (Image courtesy of Intergraph Computer Systems)

The appearance of the objects is affected by material properties, light sources, textures and shading models.

A coarse division of the real-time rendering pipeline consists of three conceptual stages:

1. *application stage*: driven by the implemented application and is therefore developed in software running on general-purpose CPUs. The developer has full control over what happens in the application stage. At the end of this stage, the geometry to be rendered is fed to the following stage. These are the *rendering primitives* (points, lines, triangles) that might eventually end up on the screen.

2. *geometry stage*: responsible for the majority of the per-polygon and per-vertex operations. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. More specifically, it deals with the model and view transformations, projections, clipping and screen mapping. The geometry stage is typically performed on a GPU that contains many programmable cores as well as fixed-operation hardware.

3. *rasterizer stage*: generates the final image by computing the colors for the pixels covered by visible objects (given the transformed and projected vertices with their associated shading data). This stage is processed completely on the GPU. *Pixel* p is the smallest controllable element of a screen image with [sc.width,sc.height] resolution size. From now on, we denote as $p.xy$ the 2D position of pixel p. More specifically, each pixel that has its center (or a sample) covered by the triangle is checked and a *fragment* f generated for the part of the pixel that overlaps the triangle (this process often called *scan conversion*). Each triangle's fragment properties, e.g. color ($f.color$), depth ($f.z$), normal and more, are generated using data interpolated among the three triangle vertices. Colors are represented by four-element vec-

tor, such as (R:red,G:green,B:blue,A:alpha), where each element has range $\in [0, 1]$. We denote as pixel density p_d the percentage of the pixels at the entire screen size that have $f(p) > 0$, where $f(p)$ is the number of generated fragments at pixel p. *Per-pixel shading* computations (e.g. texturing) are subsequently performed using the interpolated shading data as input. In OpenGL this stage is controlled by fully-programmable *fragment shaders*. The color(s) and depth information for each pixel is respectively stored in one or more *color buffers*, named *multiple render targets* (**MRT**), and a *depth buffer*, and then passed on to the next stage. *Merging* stage is responsible for resolving visibility via *Z-testing* and performing optionally various (*blending*, *stencil test*, etc.) per-fragment operations.

This structure is the engine of the rendering pipeline which is used in real-time computer graphics applications. Each of these stages is usually a pipeline itself, which means that is consists of several substages. It is the slowest of the (sub-)stages that determines the *rendering speed*, the update rate of the images. This speed may be expressed in *frames per second* (**FPS**), that is, the number of images rendered per second.

2.4.1 Hardware Occlusion Culling

Occlusion culling is a visibility determination algorithm that is used to identify those objects that reside in the view volume but still aren't visible on the screen due to occlusion. That means they are hidden by objects that reside closer to the camera. For several generations now GPUs allow hardware accelerated methods to perform occlusion culling in the form of *occlusion queries*. OpenGL provides the functionality via the extension ARB_occlusion_query. This extension defines a mechanism whereby an application can query the number of pixels (or, more precisely, fragments - pixel samples) drawn by a primitive or group of primitives. Typically, the application will render the major occluders in the scene, then perform an occlusion query for the bounding box of each detail object in the scene. Only if said bounding box is visible, i.e., if at least one fragment is drawn, should the corresponding object be drawn.

2.4.2 Multi-fragment Rendering Frameworks

Several rendering effects in a large set of applications (from computer games to visualizations tools) require robust multi-fragment storage. Maule et al. [97] conclude that sorting is the main topic of the overall research on multi-fragment rendering and used it as the criterion to classify methods into the following categories:

- *Geometry-sorting*: sort geometry (meshes or primitives) before rasterization.
- *Fragment-sorting*: sort generated fragments before computing the desired effect, using *buffer-based* or *depth peeling*.
- *Hybrid-sorting*: combine geometry-sorting with fragment-sorting.

- *Depth-sorting-independent*: blend fragments without considering their depth order.
- *Probabilistic*: estimate visibility without sorting.

In this thesis, we analytically discuss the classes of fragment-sorting and hybrid-sorting methods since they are the best option for applications (such as rendering of self-trimming surfaces) where accurate fragment extraction is of utmost importance.

Multi-pass Depth Peeling

Depth peeling methods extract layers of visibility from a graphical model using multiple geometry passes. Layers are captured in-depth order, which eliminates the need to sort the resulting fragments.

Front-to-back Depth Peeling. The classic front-to-back method [36] proposed a solution for sorting fragments by iteratively peeling off layers in depth order. Specifically, the algorithm starts by rendering the scene normally with a depth test, which returns the closest per-pixel fragment to the observer. In a second pass, previously extracted fragments are eliminated based on the depth value extracted during the last iteration (pass) returning the next nearest layer underneath. The iteration loop halts either if it reaches the maximum number of iterations set by the user or if no more fragments are produced via hardware occlusion queries. Figure 2.5 (top row, red painted boxes) illustrates the consecutive color layers when depth peeling a duck model in a front-to-back direction.

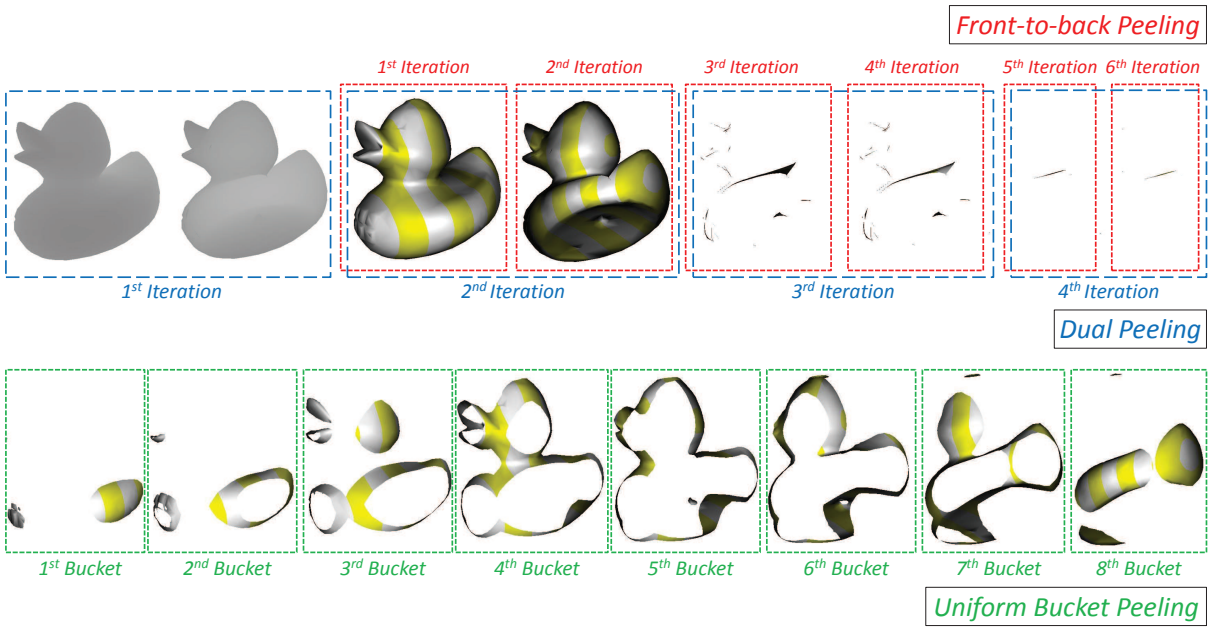


Figure 2.5: The color-buffer result of each extracted layer when depth peeling is performed using (top row) F2B, DUAL and (bottom row) BUN.

Dual Depth Peeling. Dual depth peeling [8] increases performance by applying the F2B method for the front-to-back and the back-to-front directions simultaneously. Due

to the unavailable support of multiple depth buffers on the GPU, a custom min-max depth buffer is introduced. In every iteration, the algorithm extracts the fragment information which match the min-max depth values of the previous iteration and performs depth peeling on the fragments inside this depth set. An additional rendering pass is needed to initialize depth buffer to the closest and the further layers. Figure 2.5 (top row, blue painted boxes) shows that the number of rendering iterations needed is reduced to half when simultaneous bi-directional depth peeling is used.

Bucket Uniform Depth Peeling. Liu et al. [89] presents a multi-layer depth peeling technique achieving partially correct depth ordering via bucket sort on the fragment level. This algorithm can be regarded as a simultaneous application of the DUAL peeling algorithm into uniformly consecutive subintervals of the depth range. To approximate the depth range of each pixel location, a quick rendering pass of the scene's bounding box is initially employed. Figure 2.5 (bottom row, green painted boxes) illustrates the peeling output for each bucket for a scene divided into eight uniform intervals.

Buffer-based Methods

Buffer-based methods use a buffer to store fragments while they are generated. After rasterization, a sorting step computes the correct depth ordering. The A-buffer [17] was the first method to capture all fragments per pixel during a single rendering. However it suffers from fragment overflow due to their strategy to consume unbounded memory. Several variants have been proposed to simulate the behavior of the A-buffer architecture employing different data structures reducing more or less its memory hazards. The advantage of these methods is performance superiority when compared to the depth peeling methods due to their constant rendering (one or two geometry passes) of the scene.

A-buffer using Fixed-size Arrays. Bounded buffer-based methods [90, 29] store fragment data in a global memory array using a fixed-sized array per pixel. A per-pixel offset counter indicates the next available address position for the incoming fragment. After a complete insertion in the storage buffer, the counter is atomically incremented (see Figure 2.6 (a)). The fragments are stored in order of arrival rather than in sorted order. In a post-processing phase, the fragments are finally sorted. The limitation of this class of approaches is the large memory requirements because of the fixed-size buffer, as well as loss of fragments when overflow occurs.

A-buffer using Linked Lists. Yang et al. [167] introduced a method to efficiently construct highly concurrent per-pixel linked lists via atomic memory operations on modern GPUs. The basic idea behind this algorithm is to use one buffer to store all linked-list *node* data and another buffer to store *head* offsets that point the start of the linked lists in the first buffer. A single shared counter (*next*) is atomically incremented to compute the mapping of an incoming fragment, followed by an update of the head pointer buffer

to reference the last rasterized fragment (see Figure 2.6 (b)). Shared counter needs to be updated atomically, which represents a bottleneck, since all threads trying to store a new fragment will be blocked here. The random memory accesses of the linked list also degrades performance. A subsequent full-screen pass is finally employed to sort the captured fragments by their depth values. The necessity of atomic operations for GPU memory - available only in the state of the art graphics hardware and APIs - makes it non-portable to other platforms such as mobile, game consoles etc.

A-buffer using Variable-length Arrays. Peeper [110] proposed an A-buffer architecture by extending hardware to efficiently store a variable amount of data for each pixel (see Figure 2.6(c)). In operation, a pre-pass is performed to generate the counts of the fragments per pixel in a count buffer, followed by a prefix sum pass on the generated count buffer to calculate locations in a fragment buffer in which to store all the fragments linearly. An index is generated for a given pixel in the prefix sum pass and stored in the *head* buffer. Access to the pixel fragments is then accomplished using this offset. Linear storage of the data allows for a fast rendering pass that stores all the fragments to a memory buffer without needing to look at the contents of the fragments. This is then followed by a resolve pass on the fragment buffer to generate the final image. Although this solves the problem of memory efficiency, there is a performance penalty in computing the prefix sum and rendering the scene twice.

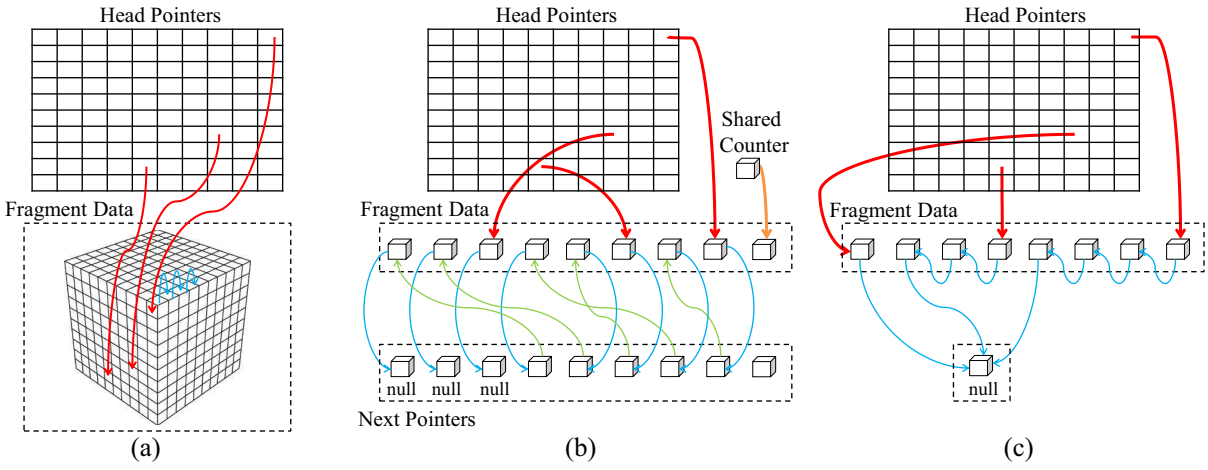


Figure 2.6: A-Buffer realizations using per-pixel (a) *fixed-size arrays*, (b) *linked-lists* and (c) *variable sequential regions*. (a) and (c) structures pack pixel fragments physically close in the memory avoiding random memory accesses of (b) when accessing the entire fragment list. However, (a) allocates the same number of entries per pixel resulting at significant waste of storage and possible fragment overflows.

Hybrid Depth Peeling

The core of hybrid-sorting techniques is the k-buffer [14, 6], a fixed-size fragment buffer that holds up to k fragments per pixel. The main algorithm first uses an approximate sorting in object space, which is not guaranteed to produce the correct ordering, but it allows for fragments to be generated during rasterization in a nearly sorted fashion. This important nearly sorted property allows the sorting to be concluded in image-space with a k-buffer, which only needs to have as many entries as necessary to fix the ordering of samples. Fragments are composited when the k-buffer becomes full, to make room for other incoming samples, and after all fragments are generated (which requires the k-buffer entries to be flushed). The k-buffer handles interpenetrating geometry, since fragment-sorting is involved. The final image quality depends on how well object-sorting reduces the number of out-of-order fragments. In situations resulting in poor fragment-sorting, the number of entries in the k-buffer might be smaller than necessary, which might introduce artifacts due to out-of-order blending. Also, in its proposition, the algorithm was prone to artifacts caused by RMWH during k-buffer updates.

The k-buffer variation given by Myers and Bavoil [105, 7] uses the stencil buffer to route fragments into a multi-sample buffer. The scene is first rasterized to capture up to k fragments per pixel, where k is the maximum number of samples available in the MSAA implementation. Each incoming fragment is routed to a sample of the MSAA buffer. If there are more than k fragments per pixel, overflow occurs and additional geometry passes are needed. Fragment routing uses a stencil mask that stores an incoming fragment in the next free MSAA sample, building a vector of fragments per pixel. Once all fragments are captured, a full-screen quadrilateral is rendered with a pixel shader that reads all fragments of a given pixel; it sorts and blends them accordingly. This approach avoids RMWH but is incompatible with hardware supported multi-sample anti-aliasing and additional stencil operations.

Chapter 3

Pose Partitioning for Multi-resolution Segmentation of Arbitrary Mesh Animations

---

Mesh segmentation has become an important component in many applications in computer graphics. In the last several years, many algorithms have been proposed in this growing area (see Section 1.3.1), offering a diversity of methods and various evaluation criteria. However, most of these segmentation techniques are limited to work well only on off-line quasi-rigid animations. Moreover, they cannot produce variable or multi-resolution segmentations. To this end, we investigate a complete solution to support all these applications. Section 2.2 provides the important preliminaries regarding the problem of segmenting MAs.

3.1 Framework Overview

In this chapter, we introduce a generic framework for efficiently generating multi-resolution segmentations that account for both articulated and highly-deformable TCMAs [152]. An over-segmentation is first constructed by combining a set of individual partitionings corresponding to each input pose of the animation sequence (Section 3.1.1). Partitioning of each pose is precomputed in parallel using any of the numerous available features and clustering methods. The major advantages of the proposed method are efficiency and suitability for off-line as well as streamed and dynamically created mesh animations by exploiting different merging strategies (see Figure 3.1 (left)(a),(b)). A progressive simplification is subsequently applied to generate a segmentation of different resolutions (Section 3.1.2) that aims at removing noisy components created from joining successive partitionings (see Figure 3.1 (right)). We propose a pose order-dependent edge-collapsing strategy that refines the segmentation graph guided by a temporal-coherent area-aware edge collapsing solution. Figure 3.2 shows an overall information flow diagram showing the individual steps of our framework when segmenting a horse animation.

Despite the independent per pose partitioning, a variable segmentation sequence can also be maintained over time by merging the partition of the current pose with that of the following one (see Figure 3.1 (left),(c)). Thus, each segmentation is similar to the one at the previous step and accurately reflects the new data arriving [21]. Finally, a novel visualization scheme is introduced that provides perceptual consistency between consecutive poses. An extensive evaluation of this method is finally provided showing the improvement over the state of the art in terms of skinning quality under a variety of skeletal and highly deformable mesh animations (Section 3.3).

3.1.1 Pose partitioning-aware over-segmentation

We begin this section with a few definitions and then we outline our over-segmentation algorithm.

Definition 3.1. We define a clustering animation vector (**CAV**) for each vertex $v \in V$

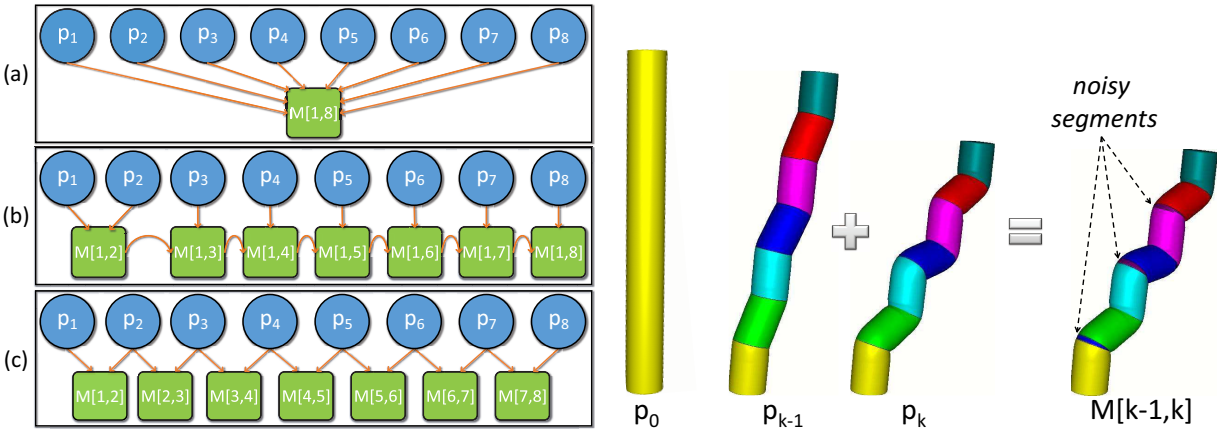


Figure 3.1: (left) Three different ways of joining a sequence of partitionings: (a) *Off-line*, (b) *real-time* and (c) *variable* segmentations. (right) Illustrating the noisy segments created when two consecutive pose partitionings are successfully merged. M$[x, y]$ denotes the over-segmentation result between $[p_x, \ldots, p_y]$ poses.

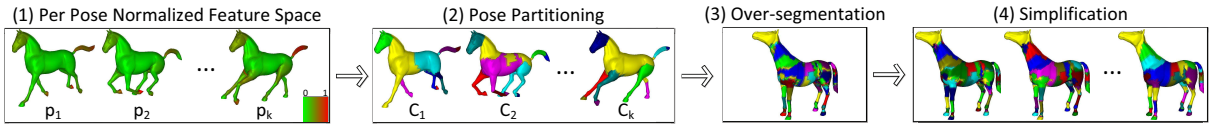


Figure 3.2: Diagram of our pose-partitioning aware segmentation pipeline.

such that $cav(v) = (m_1, m_2, \ldots, m_k)$, where $cav(v)[j] = m_j \in 1, \ldots, |C_j(M)|$ is the cluster index that v belongs to in pose p_j. Similarly, $cav(v)[i, j]$ is the vector of cluster indices where v belongs to in poses $p_i, p_{i+1}, \ldots, p_j$.

Definition 3.2. Let NS_i (neighbor similarity) for animation A be a binary relation between vertices. We say that for two vertices u NS_i v if and only if $cav(v)[1, i] = cav(u)[1, i] \wedge ((u, v) \in E \vee u = v)$. Clearly, NS_i is reflexive and symmetric for all $i \in [1, \ldots, k]$. By taking the transitive closure of NS_i, denoted by TNS_i we have an equivalence relation. The equivalence relation partitions V in equivalence classes.

Definition 3.3. The *over-segmentation* $OS(A)$ of A over $C_i(M), i = 1, \ldots, k$ is a partitioning of V in equivalence classes called segments based on the transitive closure of the binary relation NS_k. Then, we denote by $segment(v, i)$, the segment (equivalence class) where v belongs based on the equivalence relation TNS_i. For $i = k$, we obtain segments of the final over-segmentation $OS(A)$. For other $i < k$, we obtain the corresponding segment that is produced by the over-segmentation of $OS((p_0, p_1, \ldots, p_i))$.

Property 3.1. $segment(v, j) \subseteq segment(v, i),\ i \leq j$

Proof. For $i = j$, the two segments are identical, since we have equivalence classes. For $i < j$, let vertex $u \in segment(v, j)$, then there is a path between u and v, and for every vertex w in the path it holds $cav(w)[1, j] = cav(v)[1, j]$. Then, for all vertices w in this path from u and v, it holds $cav(w)[1, i] = cav(v)[1, i]$. So, $u \in segment(v, i)$. ∎

Immediate from the property above is that $segment(u_1, i) = segment(u_2, i)$, for all vertices $u_1, u_2 \in segment(v, j), i < j$. Thus, we may denote this new segment by $segment(segment(v, j), i)$.

To algorithmically obtain the segments of the over-segmentation (equivalence classes of $OS(A)$), we can easily prove that it is equivalent to detect for a vertex v the maximal connected induced sub-graph of M where v belongs and all its vertices have the same CAV. To compute the over-segmentation, we consider each vertex and perform a pruned breadth-first-search to detect connected vertices with the same CAV. During this process, we mark each edge so that we will not visit it again. This takes time $O(|E|)$ which for regular non-manifold objects is $O(|V|) = O(n)$. The details of this algorithm are shown in Algorithm 3.1, where $cav(S_j)$ corresponds to the CAV of S_j cluster of the resulted over-segmentation. Figure 3.3 illustrates the CAV generation for each segment S_j (vertices belong to same segment have the same CAV) created from three pose partitionings.

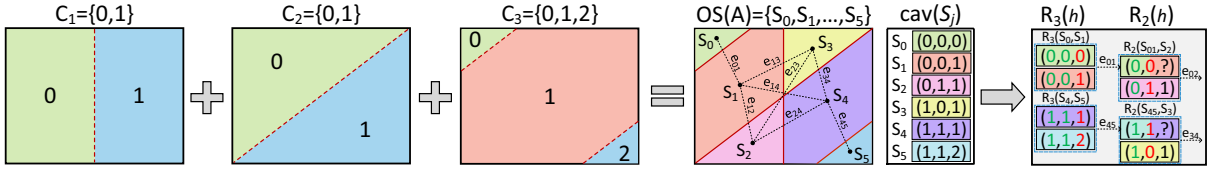


Figure 3.3: Illustrating the CAV for each vertex and segment created from joining three pose clusterings C_1, C_2 and C_3. The over-segmentation graph OS(A) is decimated only at the edges $\{e_{01}, e_{45}, e_{02}, e_{34}\}$ which satisfy the temporal-coherent reduction rule.

Algorithm 3.1 Over-segmentation(Mesh Animation A)

$OS(A) \leftarrow \oslash$; $VL \leftarrow$ list of all vertices in V;
$k_{max} \leftarrow 0$; $\forall v \in V : v.segment \leftarrow -1$;
Mark all edges as not visited;
while $VL \neq \oslash$ **do**
    $v \leftarrow VL.next$;
    if $v.segment == -1$ **then**
        $S_{k_{max}} \leftarrow \{v\}$; $v.segment \leftarrow k_{max}$;
        $k \leftarrow k_{max}$; $k_{max} \leftarrow k_{max} + 1$;
    else
        $k \leftarrow v.segment$;
    end if
    for each not visited edge $e_i \leftarrow (v, v_i)$ **do**
        Mark edge e_i as visited;
        if $cav(v) == cav(v_i)$ **then**
            $S_k \leftarrow S_k \cup \{v_i\}$;
            $v_i.segment \leftarrow k$;
        end if
    end for
    Remove vertex v from list VL;
end while
$OS(A) = \{S_0, S_1, \ldots, S_{k_{max}-1}\}$;

3.1.2 Progressive decimation of over-segmentation

Following the generation of the over-segmentation, we perform a cleaning with parameter $h \in [0, 1]$ called *p2p-cleaning* (pose-to-pose cleaning) starting from pose p_k, then for pose p_{k-1} towards the first animation pose p_1. Each cleaning operation $R_i(h)$ on pose p_i is based on the following **reduction rule**:

Given a pose p_i and a pair of segments (S_A, S_B), S_B absorbs S_A if and only if the following holds for S_A:

$$a(S_A) \leq h \cdot a(C_i^{cav(S_A)[i]}) \tag{3.1}$$

$$a(S_A) \leq h \cdot a(segment(S_A, i-1)) \tag{3.2}$$

and S_B is a segment such that,

$$S_B \in N(S_A) \wedge cav(S_A)[1, i-1] = cav(S_B)[1, i-1] \tag{3.3}$$

$$\wedge\, cav(S_A)[i] \neq cav(S_B)[i]$$

$$a(S_B) > a(S_A) \tag{3.4}$$

where $N(S_B)$ are all neighbor clusters of S_B in the final over-segmentation and $a(S) = \sum a(f_{ijk}) = \sum(0.5 \cdot |(v_j - v_i) \times (v_k - v_i)|)$ is the area of segment S which consists of a set of facets $\{f_{ijk},\ i, j, k \in [0, n)\}$. Note that after the reduction, the area of the new cluster is considered for the purposes of reductions in this pose to be this of the absorbing cluster S_B.

The conditions for S_B (3 and 4) state that S_B is the largest neighbor of S_A (larger than S_A) in the over-segmentation and they belong to the same cluster in all poses from p_1 to p_{i-1} and to a different cluster in pose p_i. This means that S_A and S_B were split to represent separate segments in pose p_i. Thus, we need to check whether one of them was erroneously created due to small cluster border differences, and should therefore be absorbed by the other. This is checked by the two conditions for S_A. The first condition ensures that S_A is small as compared to the cluster that contains it in pose p_i. thus it is not a significant part of a segment at pose p_i. The second condition state that S_A is small as compared to the superset of segment S_B in pose p_{i-1} and they have been split into two or more in pose p_i. This corresponds to a segment of the over-segmentation of the animation $(p_0, p_1, \ldots, p_{i-1})$ and at this phase is a candidate group of clusters (including S_A and S_B) to become one (or more) independent meaningful segment(s) after cleaning.

This reduction rule exploits temporal coherency, which means that we can perform an educated reverse pose-to-pose cleaning of non meaningful clusters preserving useful deformation information. We have explored global rules (not per pose) and we have observed that they tend to favor larger clusters without respecting other cluster characteristics. This fact makes them inappropriate for mesh segmentation of animation sequences. Figure 3.3 illustrates the possible reduction steps applied to an over-segmentation graph when our p2p-cleaning is employed. Note that from all potential collapsing edges, only four satisfy our history-based condition.

The details of this algorithms are shown in Algorithm 3.2. The initialization takes k steps. At each step it takes $O(n)$, $n = |V|$ to initialize the areas, to build the graph of adjacent segments and to reconstruct the partial over-segmentation. Then, with careful updating of visited edges in the segment neighbor graph we are able to carry out each step at time $O(r)$, where $r = |OS(A)|$. Thus, the cleaning process takes time $O(k(n+r))$. The following substantiates the correctness of the p2p-cleaning process.

Lemma 3.1. *The p2p-cleaning process will have a unique result in a finite number of steps.*

Proof. Since we reduce the number of segments by one at each step, the p2p-cleaning process may take at most $|OS(A)|$ reduction steps overall for all poses. Next, we shall prove that at each pose, we obtain a unique segmentation which is not affected by the order in which we apply the reductions. We can think of the cleaning process as a rewrite system, which will have a unique eventual result. More specifically, we will prove that the reduction rules are compliant to the *Church Rosser* property. Thus, if at some step of the cleaning process at pose p_i, we have a partially cleaned over-segmentation L and we have two candidates: a reduction $R_i(S_A, S_B)$ yielding a new over-segmentation Q and a reduction $R'_i(S'_A, S'_B)$ yielding a new over-segmentation Q', then it suffices to prove that there is a sequence of reductions from Q and a sequence of reductions from Q' so that we obtain the same partially cleaned over-segmentation configuration U (see Figure 3.4 (top, left)). Figure 3.4 illustrates how a portion of the over-segmentation at pose p_i is build when the C_i (painter with red) breaks the existing segments of the over-segmentation of the animation $(p_0, \ldots, p_{i-1})$ (painted with green) into one or more components. Since this is always a partitioning, we distinguish among the following cases:

1. S_A, S_B, S'_A and S'_B are four disjoint sets (for example $S_A = c_c$, $S_B = c_d$, $S'_A = c_b$ and $S'_B = c_e$), in which case we can always carry out the other reduction, and reach the same U.

2. If we have two pairs of identical sets then this is the same reduction, since this is only feasible when $S_A = S'_A$ and $S_B = S'_B$.

3. There is just one pair of identical sets. Then, we cannot have $S_A = S'_A$ since only one of the two reduction would have been eligible. If we have $S_B = S'_B$ (for example $S_A = c_b$, $S_B = S'_B = c_e$, $S'_A = c_e$), then if S'_A is absorbed by $S'_B = S_B$ then the new cluster will also absorb S_A and vice versa. It remain to consider the case were $S_A = S'_B$ (or $S'_A = S_B$). For example $S_B = c_e$, $S_A = S'_B = c_d$ and $S'_B = c_c$. If we carry out first $R_i(S_A, S_B)$, the new area of the merged cluster will be the one of S_B which by definition is larger than S_A, thus the other reduction $R'_i(S'_A, S'_B)$ will still be eligible. If $R'_i(S'_A, S'_B)$ is carried out first then the new merged cluster will have the area of $S_B = S'_A$, therefore it will be still eligible to be absorbed by S'_B.

∎

Algorithm 3.2 p2p-Cleaning(Over-segmentation $OS(A)$, Float h)

1: **for** $p \leftarrow k, 1$ **do**
2: OS(A).computeArea();
3: **for each** $S_A \in OS(A)$ **do**
4: **if** $Clean(S_A, C_p^{cav(S_A)[p]}, h, p)$ **then**
5: $maxArea \leftarrow a(S_A)$;
6: **for** $S_B \in N(S_A)$ **do**
7: **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**
8: $maxArea \leftarrow a(S_B)$;
9: **end if**
10: **end for**
11: **if** $maxArea > a(S_A)$ **then**
12: $S_B.Copy(S_A)$; ▷ neighbors,vertices,faces
13: **comment:** Leave $a(S_B)$ unchanged
14: $OS(A).Remove(S_A)$;
15: **end if**
16: **end if**
17: **end for**
18: **end for**

19: **function** CLEAN(Segment S_A, Segment S_P, Float h, Pose p)
20: **if** $a(S_A) > h \cdot a(S_P)$ **then**
21: **return** false;
22: **end if**
23: S_L = new List<Vertex>();
24: **comment:** The following is realized with a breadth
25: first search from S_B (similar to Algorithm 3.1)
26: **for each** $v \in segment(S_A, p-1)$ **do**
27: $S_L.Add(v)$;
28: **end for**
29: **return** $a(S_A) \leq h \cdot a(S_L)$;
30: **end function**

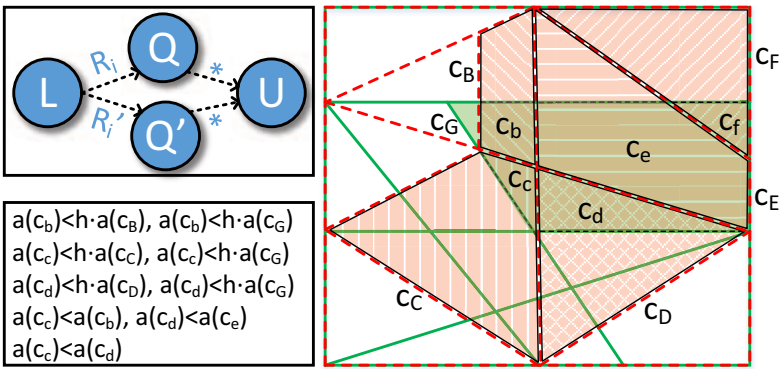


Figure 3.4: Illustrating the correctness proof: In this example, we close up at the reduction that will restructure the green cluster c_G, which belongs to the over-segmentation until pose p_{i-1}, when is decomposed by the red partitioning of p_i. For small h, segment c_e will absorb c_f and segment c_d will absorb c_c.

3.2 Applications

In this section, we offer several applications enabled by the idea of individual pose-partitioning animated meshes.

3.2.1 Smooth Visualization of Cluster Transitions

Since clusters may vanish, shift or arise when moving through time, we introduce a perceptually friendly visualization scheme to propagate as mush as possible the segment colors between consecutive frames. A user should perceive the transition from one frame to the next one avoiding if possible to encounter totally different coloring of clusters. We follow a strategy that aims at covering a high distribution of the color space minimizing the possibility of "close" colors be assigned to neighbor clusters. The algorithm starts by initially painting the partitioning of the first pose, followed by a propagation of the cluster colors from pose to pose.

Rest-pose coloring

A breadth-first traversal is applied by picking a random cluster as root node. At each node visit, we set the next color from the palette shown in Figure 3.5 that does not conflict with an assigned color from its neighborhood. A 2-ring neighborhood can also be used to increase color distribution. The palette is an *RGB color wheel* with 12 divisions: an illustrative organization of color hues around a circle that consists of the primary, secondary and tertiary colors. Note that complementary colors lie opposite to each other in the color sphere. In case of color overflow (chromatic number of the cluster graph > 12), a larger ring of colors should be used.

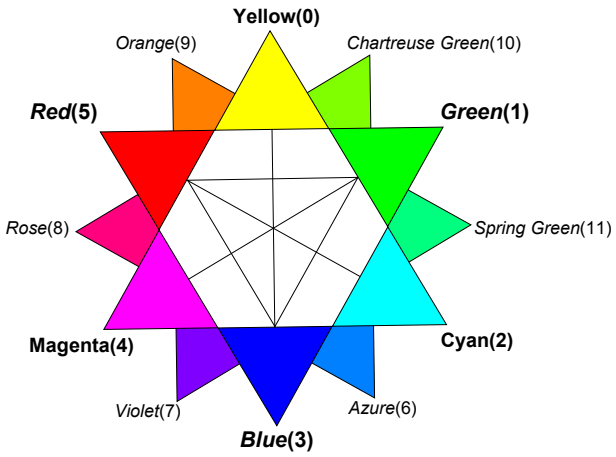


Figure 3.5: RGB color wheel which consists of ***primary***: {red, green, blue}, **secondary**: {cyan, yellow, magenta} and *tertiary* colors: {colors between primary and secondary ones}. Note that complementary colors lie opposite each other at the color sphere. The order of each color is shown in brackets.

Pose-to-pose color propagation

A new *future-cluster* initially computes how much area covers from each of its overlapping *past-clusters* in linear time. A breadth-first traversal is afterwards applied picking as root the cluster with the largest covering area. Each future-cluster inherits the color of the first of its past-clusters (sorted by the overlapping area) that is not covered by any of the rest of the future-clusters. (For example, it is possible that the largest covered past-cluster of a future-cluster is covered more from another future-cluster with not assigned color.) Note that this color must not conflict with any previously assigned color from its neighborhood. If a future-cluster covers only one past-cluster and cannot inherit any color, we assign it the next available color from the palette. Otherwise, we use a mixed color of its two largest covering past-clusters.

To avoid the following problems that arise when mixing neighbor cluster colors during the color propagation phase: (a) producing a color slightly different from the existing ones and (b) giving a neutral grey color, we propose (a) changing the cyclic traversal order of colors and (b) assigning to a cluster a color that is non-complementary with respect to the neighbor clusters.

Figures 3.10 (a), 3.6 (b), 3.9 (a), and 3.11 (a) illustrate the perceptual intuitiveness of our color propagation scheme when moving throughout the sequence. The same algorithm can also be used for any segmentation pair without the need of history context. Thus, we have applied this approach to demonstrate the color transition results between (a) the variable segments and (b) the variable resolution segments obtained by our method. Finally, Table 3.1 shows its interactive nature when rendering several clustering sequences (from the high-detail elephant gallop (188 FPS) to the low-detail hand animation (715 FPS)).

3.2.2 Real-time Segmentation

Our framework can efficiently handle segmentations of streamed or dynamically created mesh animations without the need of downloading the pre-processed animation frames for off-line segmentation. This is a key feature that is not offered by previous approaches. The idea is to merge the newly "arrived" pose partitioning with the segmentation resulted from joining the partitionings of all previous poses (see Figure 3.1(left),(c)).

Figures 3.6 (a) and 3.9 (b),(c) provide thorough examples of incrementally merging a number of pose-partitionings to generate the final segmentation of real-time mesh animations. Observe that the cluster-refinement is omitted at each frame in Figure 3.6 (a). On the other hand, the intermediate segmentations are enhanced by the cleaning procedure in Figure 3.9 (c).

3.2.3 Variable Segmentation

Despite clustering independence between individual successive poses, we offer users with a smooth transition between pose-to-pose clusterings, by joining the optimal partitioning of the current pose with that of the next pose (see Figure 3.1 (left),(c)). Due to the small number of resulting clusters, cleaning can be avoided. Our method is highly efficient when compared to previous variable segmentation methods (evolutionary versions of classic clustering methods [21] and splitting/merging operation on the past-clustering [4]) and can achieve interactive performance when a fast per-pose clustering is employed (for computation times see Table 3.1).

Figures 3.6 (c) and 3.11 (b) include variable segmentations of a cloth simulation and a dance animation, respectively. Observe that the temporal consistency is better preserved when the joined clustering sequence is used as compared to the individual pose clusterings.

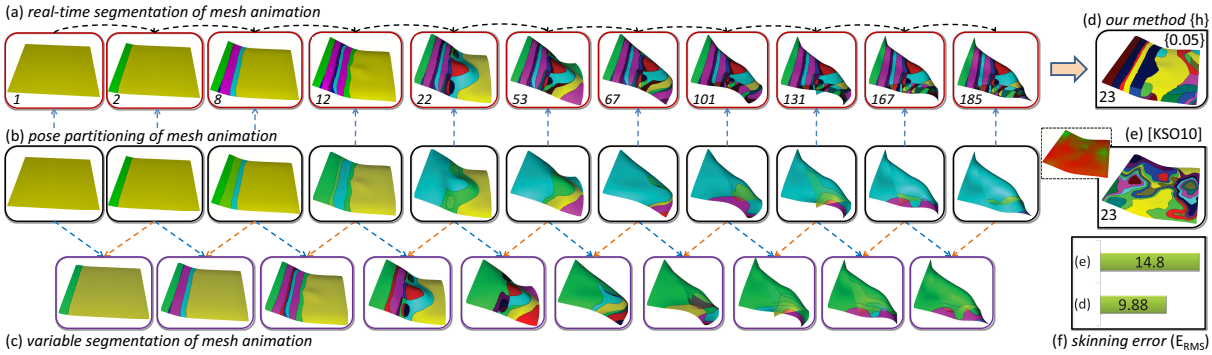


Figure 3.6: *Flowing cloth mesh animation*: (a) Real-time segmentation construction process. (b) Pose partitioning enhanced by our color propagation scheme. (c) Variable segmentation. (d) A snapshot of our joint segmentation which consists of 23 components. Our output is superior in the context of (f) skinning error when compared to (e) the one derived from [69] (thumbnail of the normalized feature space is also provided).

3.2.4 Multi-resolution Segmentation

Contrary to bottom-up and top-down hierarchical clustering methods which can only merge or only split clusters to reach the desired segmentation solution, we provide users with an interactive tool to adapt resolution of the final segmentation. As discussed above, our approach aims at cleaning small-area clusters which usually correspond to highly-deformable regions. Starting from a noisy over-segmentation, users can efficiently simplify it by adjusting the h parameter. The more h is increased, the larger the parts to be removed. However, they are upper-bounded by the resolution of the initial over-segmentation. Figures 3.10 (c) and 3.11 (c) illustrate how the rigidity in the segmentation is preserved when the level of detail is decreasing. Table 3.1 shows the performance efficiency of the p2p-cleaning process for various mesh animations. Note that this process does not depend on mesh geometry size.

3.2.5 Combine Segmentations of Mesh Animations

Except from joining pose partitionings to derive a final segmentation, our framework can easily combine global segmentations of one or more mesh animations. Note that our approach can only work when a bijective vertex mapping between the animations has been established.

Figure 3.2.5 (top) demonstrate how we can perform a global segmentation accounting both velocity and acceleration features [12]. A unique global segmentation is computed from each feature (using the numerical mean of the per-pose values). Afterwards, a final segmentation is produced by joining the per-feature global segmentations. We observe that the merged segmentation preserves better both features when compared to the global segmentation created using a normalized feature vector space where final output is mostly influenced by the velocity feature.

Figure 3.2.5 (bottom) illustrates the final segmentation result of joining two individual mesh animations. This is very helpful since we can avoid the burdensome manual work of animators to produce the intermediate result. The global segmentation of each mesh animation produces a better partitioning of each movement leading at a superior final segmentation when compared to the one derived from merging both animations. This is due to the reduced multi-modal distribution of the feature space computed from the motion of the merged animation.

3.2.6 Modifying Mesh Animations

Using our framework, we can avoid the segmentation re-computations when the user performs editing or extending operations on the original animation sequence [19]. Since the over-segmentation result does not depend on the joining order, we can simply join the new partitioning (from the edited or the added pose) with the final segmentation of the original animation.

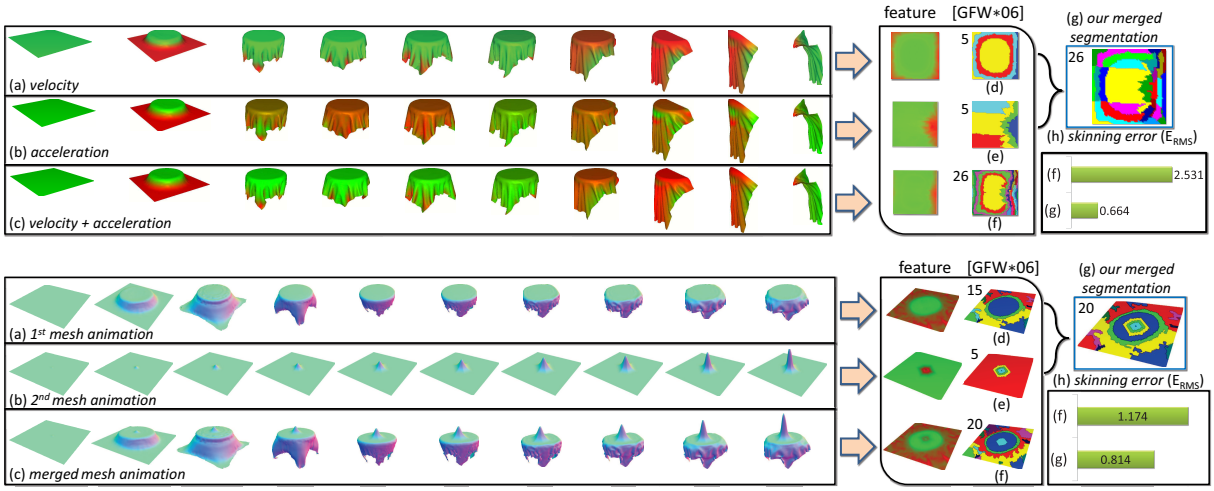


Figure 3.7: Two representative examples of combining individual global segmentations extracted from different *tablecloth mesh animations*. Global segmentations, illustrated at both scenarios, are computed using [41]. (top) Merging (d),(e) two individual global segmentations of a mesh animation extracted using (a),(b) two different features. Note that (g) our merging output preserves better both features when compared with the (f) global segmentation derived using (c) the normalized two-dimensional feature space. (bottom) Merging (d),(e) the global segmentations of (a),(b) two individual mesh animations. (h) We observe the skinning error superiority of (g) our merged segmentation when compared with (f) the global segmentation of (c) the animation created by merging both animations.

Figure 3.8 illustrates how a segmentation, computed from merging clusterings of an initial set of flamingo poses, is adjusted to reflect the motion of two newly added poses. On the other hand, Figure 3.9 demonstrates how the final segmentation of a mesh animation is efficiently altered when pose editing is performed. For clarity, we provide the intermediate steps of the incremental merging strategy despite the fact that the same segmentation can be produced by merging the partitioning of the edited poses with the final segmentation of the original animation.

3.3 Experimental Study

We evaluate our proposed segmentation technique with respect to performance and quality under a set of various testing inputs. These include rigid, highly-deformable and hybrid mesh animations. Table 3.1 summarizes the geometry properties and clustering details for each animation. The experiments were performed on a Intel Core i7 870 (8M Cache, 2.93 GHz, 8 threads) CPU using multi-threaded implementation.

A variation of a top-down hierarchical clustering technique [41] is used in our experiments for primary pose-decomposition. Rotation angles, extracted from the deformation gradients [138] computed with respect to the rest-pose, define the one-dimensional feature space (see Section 2.2.1). We compare our segmentation results with the ones derived by

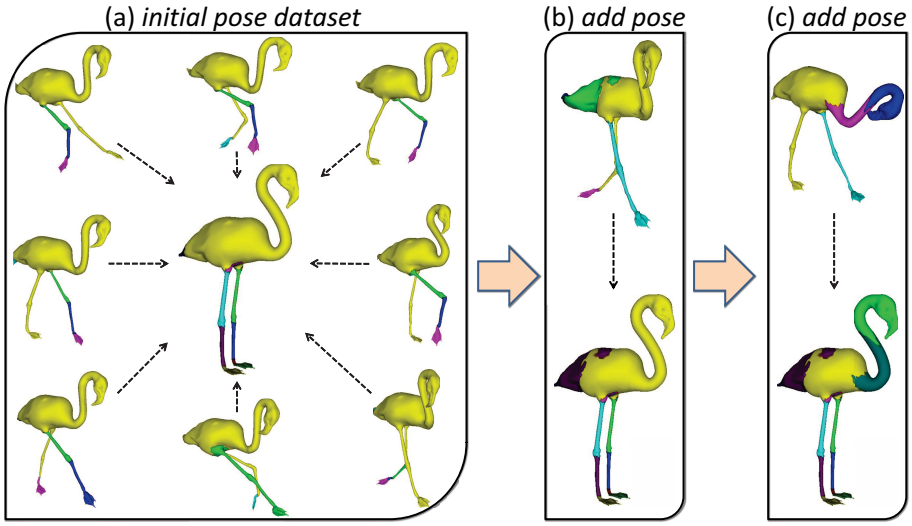


Figure 3.8: *Flamingo pose dataset*: (a) Final segmentation constructed by joining 8 initial pose partitionings. The segmentation is refined after adding (b) initially a new pose, (c) followed by a second one. All pose partitionings consist of 5 components.

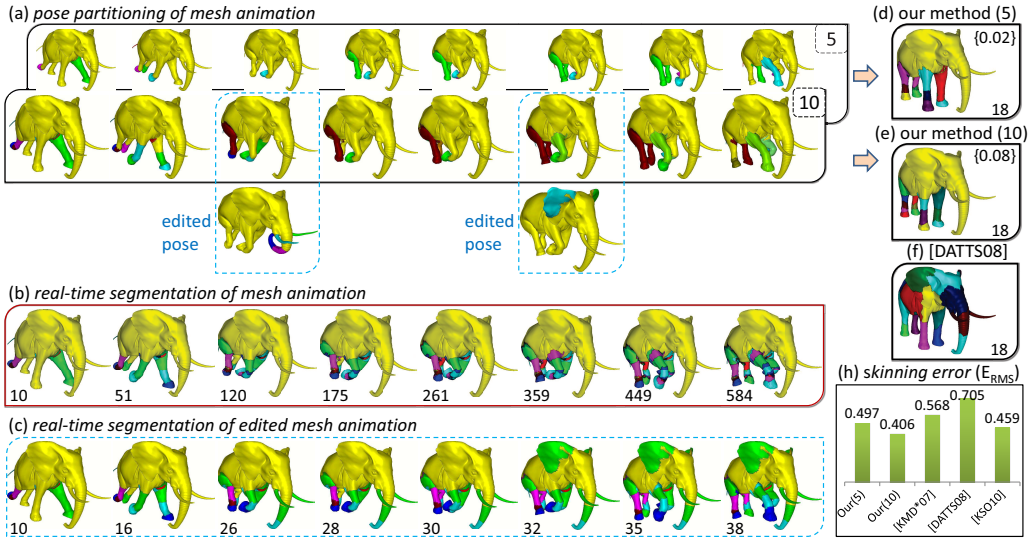


Figure 3.9: *Elephant gallop mesh animation*: (a) Smooth visual transitions of the pose partitionings with 5 and 10 components. Editing operations are highlighted from the partitioning of the modified poses. Intermediate real-time segmentation steps (b) before and (c) after editing is applied. (d),(e) Contrary to our refined segmentations, (f) output of [31] results at wrongly decomposing non-animated areas. (h) Observe the insufficient quality of (e) the final segmentation created by our method when a limited per-pose clustering output is used.

a variety of widely-accepted global segmentation methods using the same number of desired segments. Without loss of generality, we have used uniform seeding and the same number of iterations (5) for all clustering algorithms and 1% of the vertices are used as initial input for spectral clustering [31]. Finally, K-means [80] and spectral clustering [31] may result in segments with several disconnected components when segmenting non-rigid animations making these results unable to support several graphics applications.

3.3.1 Performance Analysis

Table 3.1 presents a comparative performance overview of the intermediate steps employed by our framework to produce a final segmentation. The computation times for all steps exhibit a linear behavior on the mesh geometry size, which is consistent with our time complexity analysis. Furthermore, note how the over-segmentation and cleaning performance scales linearly when the number of per-pose segments increases at the elephant mesh animation. The efficiency of our framework is constrained by the individual pose decompositions which take more that 90% of the total computation time. Finally, note that we cannot support interactive performance for segmenting mesh animations when the top-down hierarchical clustering is used. Moving to multi-source region growing [69] as initial partitioning, we achieve 6 FPS when real-time segmenting the hand animation (Figure 3.10). A GPU-accelerated clustering may be explored as an alternative to speed up performance.

Figure 3.11 (f) shows that our algorithm is better in terms of performance when compared to a variety of global segmentations. This is due to the fact that segmentations that explore spectral clustering [31] or skinning transform matrices [41, 126, 80] as motion characteristic suffer from high computation times. Mean-shift clustering aware methods [55, 85] should also be avoided to produce fast global segmentations. On the other hand, region-growing [69] is faster than our method in low resolutions. However, this comes with the price of limited quality of the generated partitions. Contrary to other methods, our method is only slightly affected when changing from one segmentation resolution to another.

				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

Table 3.1: Extensive performance comparison (in seconds) of the algorithmic steps of our method to derive segmentations of various mesh animations.

3.3.2 Quality Analysis

Contrary to static meshes where several definitions and metrics have been introduced to define optimal segmentation depending on the application objective [23, 91], a framework for the objective evaluation of segmentating mesh animations is missing.

In this work, we evaluate our method in a *skinning context*: how well the segmentation-aware compressed animation reproduces the original animation when linear blend skinning is used (see Section 2.3.2). A simple influence assignment method is initially employed to set the per-vertex skinning weights [68], followed by a fitting process which computes the transformation matrices that describe the transition from the rest-pose to the subsequent poses. The E_{RMS} metric [69] is finally used to measure the mean skinning approximation error of the animation sequence (see Section 2.3.4).

Rigid Animations

When mesh movement is defined as a function of an underlying skeleton, the segmentation objective is to partition the surface into meaningful volumetric parts. Figure 3.10 (e) shows the comparison of our method in terms of extracting rigid parts when segmenting an animated hand. Low-resolution global segmentations [126] fail to accurately partition most of the articulations (e.g middle finger). These segments are captured at higher-detail representation with the cost of noise cluster creation. From the error table, we observe that the behavior of our method starts to change when the number of the final segmentation resolution is low. This is reasonable since high-resolution segmentations consist of numerous noisy tiny clusters generated between consecutive partitionings. Increasing the number of per-pose extracted segments will enhance the skinning approximation.

In Figure 3.9 pose partitionings of 5 and 10 components are merged to construct multi-resolution segmentations of an elephant gallop animation. First, we observe that the quality of the former is insufficient due to the low number of segments per pose. This results in a significant loss of semantic part information such as the knee of the front-left foot (red-colored). The segmentation quality is sufficiently improved when more per-pose clusters are used. The accompanying table illustrates the superiority in terms of the error measure of our method when compared with several methods on a 18-component final segmentation.

Deformable Animations

Figures 3.6 and 3.2.5 describe objects that deform under no skeletal influence. In that particular case, segmentation is targeted at decomposing the mesh into surface patches with similar motion characteristics. Figure 3.6 shows a segmentation that consists of 23 components from a cloth simulation. Note that the number of per-pose clusters is not-constant. Our segmentation preserves better spatial coherency without creating irregular shapes when compared to the global segmentation [69] extracted from the illustrated feature space.

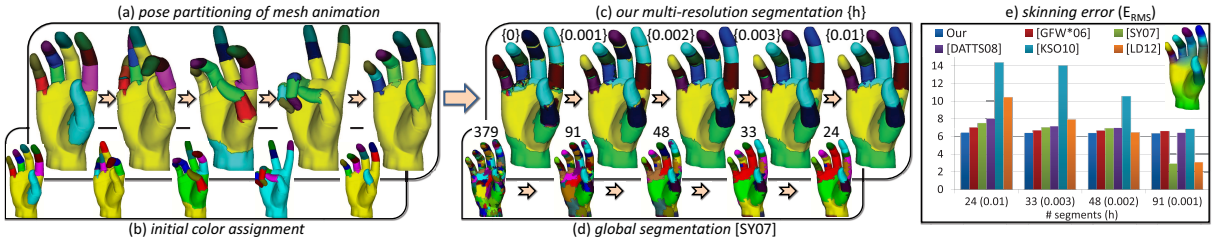


Figure 3.10: *Hand mesh animation*: (a) A smooth view of pose-to-pose partitioning transition is shown by propagating the cluster colors from the rest-pose to the subsequent ones. (b) Thumbnails illustrate the initial random painting of five representative ones. (c) Five multi-resolution segmentations efficiently constructed by refining the over-segmentation derived from the individual partitionings. Our segmentations are superior in terms of (e) skinning error when compared to the ones derived from previous works ((d) illustrated segmentations of [126]) in most of the testing resolutions. Skinning weights computed from the smallest segmentation of our method is also shown.

Two representative examples of combining individual global segmentations [41] extracted from different tablecloth mesh animations are shown in Figure 3.2.5. Two individual global segmentations, computed based on the arithmetic mean of vertex velocity and acceleration characteristics of the animation, are efficiently merged in Figure 3.2.5 (top). Observe that our joined output preserves better both motion features when compared with the global segmentation extracted using the normalized two-dimensional feature space (mostly influenced by the mean velocity feature). Figure 3.2.5 (bottom) shows an example of a *segmentation transfer* between different mesh animations. The global segmentation of the first mesh animation is efficiently transferred to the second one. Mean rotation angle was used to define the feature space. Note that the merged segmentation is superior when compared to the global segmentation of the animation created by blending both animations (e.g the highly-animated protrusion region was captured by only one segment).

Hybrid Animations

Highly deformable objects can be used to model clothes in conjunction with skeletal animation. Figure 3.11 illustrates how our approach produces segmentations that accurately partition the rigid parts (head, arms and legs of the dancer) from the highly-deformed surfaces (the dress follows the motion of the dancer) of a samba dancing animation. On the other hand, global segmentation [126] produces low-quality partitions. Despite the sufficient rigidity captured at low-detail, the right leg is wrongly connected to the dress. Moving to higher resolutions, we observe that rigid components (head and legs) are significantly being "pruned" creating meaningless parts. This leads to a decreasing consistency of the overall segmentation. Similarly to the hand animation, we observe that moving from a high-to-low dimension our method behaves better as compared to the rest of methods when skinning error is used to approximate the initial mesh animation.

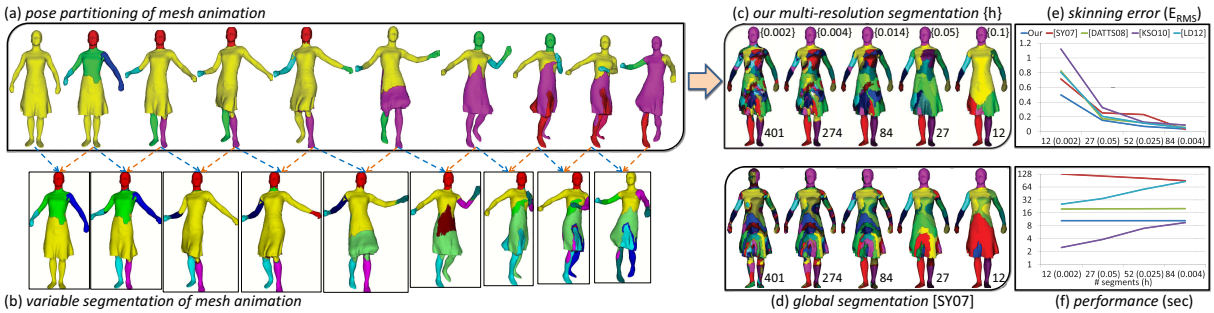


Figure 3.11: *Samba mesh animation*: (a) Propagating component colors through the animation sequence. (b) Smooth transition between pose-to-pose partitionings via variable segmentation. (c) Our multi-resolution segmentations accurately divide rigid parts from non-rigid surfaces even in high resolutions as opposed to (d) the ones of [126]. (e) Observe the superiority of our output in the context of skinning quality when compared with a variety of state-of-art methods.

3.3.3 Limitations

On the other hand, our framework has some limitations that we will briefly discuss in this section. Additional memory is required by our framework when compared to the global segmentation methods in order to store the individual per-pose partitionings in the case of segmenting off-line mesh animations. Furthermore, the final segmentation output depends on the quality and the number of the individual partitionings extracted from each pose. Merging partitionings that do not capture the desired information, would normally lead to poor final segmentation (for example the segmentation quality of Figure 3.9 (d) is limited when 5 clusters per pose are used). An efficient low-detail segmentation can only be derived in case where the individual pose partitionings exhibit high similarity. Moreover, it is not suitable for computing accurate high-resolution segmentations when the number of per-pose segments is maintained at low levels.

3.4 Conclusions

In this chapter, we have proposed a novel approach for the multi-resolution segmentation of mesh animations involving two main steps:

1. An over-segmentation is initially computed based on a precomputed set of initial partitioning for each input pose.

2. A robust cleaning process is subsequently applied to refine the over-segmentation from the wrongly-created segments generated at the partition union of successive poses.

Contrary to prior global segmentation methods, our pipeline

- handles both off-line, real-time and editable mesh animations

- supports rigid, highly-deformable and hybrid mesh animations.
- carries out interactive selection of the segmentation level of detail,
- offers smooth variable segmentations in real-time,
- achieves a consistent colorization of the segments throughout the animation,
- behaves better when skinning is used to approximate the initial mesh animation.

Chapter 4

Pose to Pose Skinning of Animated Meshes

In computer animation, key-frame compression is essential for the efficient storage and processing of the animation sequence (see Section 1.3.2). Previous work has adjusted efficient skinning techniques for data reduction (see Section 1.1.4) using affine or rigid transformations to derive the skin from the rest pose using a relatively small number of control points (mostly defined by a primary segmentation of the animated mesh). However, these methods are not capable of preserving temporal coherence for skinning and thus cannot support arbitrary pose editing and other applications. Readers may refer to Section 2.3 for a detailed mathematical background of the state-of-the-art skinning frameworks.

4.1 Framework Overview

In this chapter, we introduce *p2p-skinning* [145], a novel variation of the classic skinning framework, which supports both linear and non-linear fitting for skinning approximations and can be used for all types of object animation: skeletal, highly deformable and hybrid

animations. The neat idea is that the transformations are applied so as a new pose is derived by transforming the vertices of the previous pose (Section 4.1.1). Although fitting is performed from pose to pose, a reproduction scheme from the rest pose to an arbitrary pose can be produced efficiently. So, it enables the full spectrum of applications supported by previous approaches in conjunction with a novel pose editing of arbitrary animation frames, which can be smoothly propagated at the subsequent frames generating new deforming mesh sequences without altering the skinning representation (Section 4.1.3). Finally, we present refinement techniques (Section 4.1.2) that can improve the visual fidelity of the approximation (Section 4.2) without increasing the storage requirements.

4.1.1 Pose-to-pose Fitting

The problem addressed in this section is how to compute the transformation matrices that describe the bone movement throughout the animation. Describing the movement of highly deformable objects requires the use of affine transformation matrices to capture deformations other than rotation and translation. We assume that the bone distribution and influence (weight computation) have been established driven by a segmentation method.

As discussed in Section 2.3.2, a linear blend skinned vertex position v_j^t for a pose p_t is computed using B number of weights per vertex $[w_{1,j}, w_{2,j}, \ldots, w_{B,j}]$ and B transformation matrices $[M_1^t, M_2^t, \ldots, M_B^t]$:

$$v_j^t = T_j^t \cdot v_j^0, \tag{4.1}$$

$$T_j^t = \sum_{b=1}^{B} w_{b,j} M_b^t \tag{4.2}$$

where v_j^0 is the position of vertex j at the rest pose p_0. The vertex weights determine the bones that influence a vertex and are normally considered to be convex: $\sum_{b=1}^{B} w_{b,j} = 1$ and $w_{b,j} \geq 0, \forall b \in [1, B]$.

Rigid body motion using dual quaternions [67] can also be used with a trade off that consists of a reduction of the quality of the approximation, since there is no guarantee that the deformation of a vertex is purely rigid (see Section 2.3.3). The dual quaternion vertex $\hat{v}_j^0$ in the rest-pose p_0 is deformed to a pose p_t by

$$\hat{v}_j^t = \hat{O}_j^t \cdot \hat{v}_j^0 \cdot \hat{I}_j^t, \tag{4.3}$$

$$\hat{O}_j^t = \sum_{b=1}^{B} w_{b,j} \hat{M}_b^t, \tag{4.4}$$

$$\hat{I}_j^t = \left(\overline{\hat{O}_j^t}\right)^{-1} \tag{4.5}$$

where $\left[\hat{M}_1^t, \hat{M}_2^t, \ldots, \hat{M}_B^t\right]$ are the rigid dual quaternion transformations. In this formulation, we use the hat to denote a quantity expressed as a dual quaternion and the over-bar to denote dual conjugation.

As described at Section 2.3.4, a global formulation for the problem of skinning approximation supporting both skinning methods can be stated as minimizing:

$$\sum_{j=0}^{n-1} \left\| \mathbf{v}_j^t - v_j^t \right\|^2, \tag{4.6}$$

where $\mathbf{v}_j^t, j \in [0, \ldots, n), t \in [0, \ldots, k)$ correspond to the original n vertex positions of the input animation sequence consisting of k poses.

Previous approaches [55, 68, 69] compute the transformation matrices that describe the transition from the rest-pose to an arbitrary pose of the animation sequence. While these fitting techniques perform well for a variety of deformations, artifacts tend to be the more persistent the farther a deformation deviates from the rest-pose shape. The problem is that these methods cannot accurately capture extreme deformations from different poses when transforming a specific rest-pose due to the insufficient degrees of freedom in the skinning methods. One solution is to generate an averaged rest shape that is used as the basis for all deformations [48]. This approach is appropriate for human shapes producing sub-optimal averaged rest-poses but is not capable of handling arbitrary deformations. Moreover, a pose-invariant representation based on the theory of multidimensional scaling [28] can be explored. However, unnatural self-intersecting meshes may be generated. Previous approaches optimized skinned approximations using rest-pose displacement corrections [77], limiting users to perform efficient rest-pose editing. Thus, we exploit the temporal coherence of the animation sequence by observing that only small deformation variations will normally occur between sequential poses. We reformulate equations 4.1 and 4.3 to handle a pose-per-pose deformation scheme using the following formulas:

$$v_j^t = \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right) v_j^{t-1} \tag{4.7}$$

$$\hat{v}_j^t = \left(\sum_{b=1}^{B} w_{b,j} \hat{Q}_b^t \right) \hat{v}_j^{t-1} \overline{\left(\sum_{b=1}^{B} w_{b,j} \hat{Q}_b^t \right)}^{-1} \tag{4.8}$$

where Q_b^t and $\hat{Q}_b^t$ are the affine matrices and dual quaternions, respectively of the proxy joint b that weighted derive a vertex of pose p_t from pose p_{t-1}. Note that transformations Q_b^1 and $\hat{Q}_b^1$ are used to derive the skin vertices of pose 1 from the rest pose (see Figure 4.1). The fitting of pose-per-pose transformations can be performed in the same way as with the rest-pose scheme for both skinning techniques.

Pose-per-pose matrices are transformed to the classical LBS representation by using the following recursive formula for defining the T_j^t matrix:

$$T_j^t = \left(\sum_{b=1}^{B} w_{b,j} Q_b^t \right) T_j^{t-1}, \tag{4.9}$$

where T_j^0 at the rest-pose corresponds to the 4×4 identity matrix. Similarly for the dual

quaternions scheme we have:

$$\hat{O}_j^t = \left(\sum_{b=1}^{B} w_{b,j} \hat{Q}_b^t \right) \hat{O}_j^{t-1} \tag{4.10}$$

$$\hat{I}_j^t = \hat{I}_i^{t-1} \left(\overline{\sum_{b=1}^{B} w_{b,j} \hat{Q}_b^t} \right)^{-1} \tag{4.11}$$

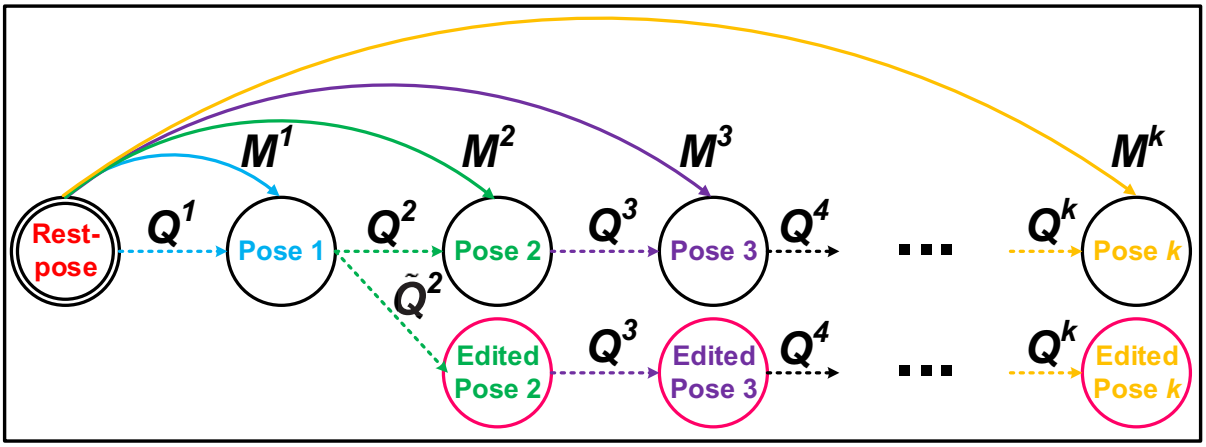


Figure 4.1: Illustrating the transformation matrices that describe the bone movement between rest-pose (M^j) and pose-to-pose (Q^j) throughout the animation sequence. (top) The original animation sequence. (bottom) After editing the second pose ($\widetilde{Q}^2$).

4.1.2 Skin Corrections

Inspired by a volume correction method which extends LBS [156], we introduce skin correction techniques based on rest-pose vertex position and weight displacements. The corrections are embedded in the resulted skinned mesh and need not be stored separately as EigenSkin corrections [77]. We present closed-form solutions which are computed directly and eliminate distortion artefacts produced by transformation fitting.

Rest-pose Corrections

Given the computed weight values and the transformation matrices from all poses, we define a displacement field $e^V = [e_0^V, \ldots, e_{n-1}^V] \in \mathbb{R}^{4\times n}$ that if added to the vertex positions of the rest pose, will correct skinned approximation. To ensure validity in terms of the homogeneous coordinates, (i.e. after the correction is applied the rest-pose vertices lie on the same plane in homogeneous coordinates $w = 1$), we set the w coordinates of e^V to zero. Formally, the problem can be stated as the minimization of

$$\sum_{t=0}^{k-1} \left\| T^t(v_j^0 + e_j^V) - \mathbf{v}_j^t \right\|^2, \; j \in [0, \ldots, n) \tag{4.12}$$

The solution of this problem is equivalent to finding the least squares solution of $\overline{T}^t \bar{e}_j^V = \bar{\mathbf{e}}_j^t$ where $\mathbf{e}_j^t = \mathbf{v}_j^t - v_j^t$ and an over-bar denotes: the $3D$ vector for a homogeneous vector, the top left 3×3 sub-matrix for an affine matrix. The above system can be rewritten as a linear system of the form $Ax = b$ where A is a block vector of n blocks. Each of these $3k \times 3$ blocks contains the weighted transformation matrices. Finally, b is formed by stacking $\mathbf{e}_0^t, \ldots, \mathbf{e}_{n-1}^t$. The displacement field e^V is then extracted from vector x and further added to the rest-pose vertex positions.

Weight Corrections

Similarly, this technique can be adapted to handle vertex weight displacement corrections $e^W = [e_1^W, \ldots, e_B^W] \in \mathbb{R}^{1 \times B}$ of the rest-pose.

The limitation with this correction is that we cannot directly use the pose-to-pose approximated skinned vertices into the minimization system due to the sequential nature of its fitting process. Logically, this will lead to a highly complex system of non-linear equations. To solve this problem, we assume that the skinned vertices match the optimal ones $v_j^t = \mathbf{v}_j^t,\ \forall t \in [0, \ldots, k)$. However, since there is a difference between them, the vertex correction process must be performed after correcting the influence skinning weights. More specifically, we would have to minimize

$$\sum_{t=0}^{k-1} \left\| \sum_{b=1}^{B} (w_{b,j} + e_{b,j}^W) Q_b^t \mathbf{v}_j^{t-1} - \mathbf{v}_j^t \right\|^2,\ j \in [0, \ldots, n) \tag{4.13}$$

This is equivalent to the least squares of the linear system:

$$\sum_{b=1}^{B} e_{b,j}^W Q_b^t \mathbf{v}_j^{t-1} = \mathbf{v}_j^t - \sum_{b=1}^{B} w_{b,j} Q_b^t \mathbf{v}_j^{t-1} \tag{4.14}$$

This is expressed as a system of linear equations of the form $Ax = b$ where A is a vector of $3P \times B$ blocks containing the transformed rest pose vertices for each of the influencing proxy joints. The b vector is the same as in the vertex correction technique.

Direct solution of this system can result in weights with potentially large positive and negative values. To avoid *over-fitting*, we constraint the weights to non-negativity and convexity.

The *convexity* constraint is imposed by eliminating one weight variable from the equation

$$\sum_{b=1}^{B} (w_{b,j} + e_{b,j}^W) = 1 \Leftrightarrow \sum_{b=1}^{B-1} e_{b,j}^W = -e_{B,j}^W,\ j \in [0, \ldots, n) \tag{4.15}$$

and can be added implicitly to the linear system by subtracting the last column of matrix A from each of the rest. To handle the *non-negativity* of the corrected weights, we must solve our linear system $Ax = b$ subject to two constraints:

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. $w_{B,j} + e^W_{B,j} \geq 0 \Leftrightarrow \sum_{b=1}^{B-1} e^W_{b,j} \leq w_{B,j}$

Note that solving a constrained least squares linear system is considerably slower than solving an unconstrained system.

Rigid Fitting

For brevity, we will only provide the minimization equations which must be solved to support corrections with DQS. The minimization functions for the vertex position and the weight corrections are given by:

$$\sum_{t=0}^{k-1} \left\| \hat{O}^t_j (\hat{v}_j + \hat{e}^V_j) \hat{I}^t_j - \hat{\mathbf{v}}^t_j \right\|^2 \tag{4.16}$$

$$\sum_{t=0}^{k-1} \left\| \left(\sum_{b=1}^{B} (w_{b,j} + e^W_{b,j}) \hat{Q}^t_b \right) \hat{\mathbf{v}}^{t-1}_j - \hat{\mathbf{v}}^t_j \left(\overline{\sum_{b=1}^{B} (w_{b,j} + e^W_{b,j}) \hat{Q}^t_b} \right) \right\|^2 \tag{4.17}$$

4.1.3 Applications

Using our system, we are able to recreate original input mesh sequences supporting all classes of animations using hardware accelerated implementations of linear and non-linear skinning methods. Efficient compression of the animation is also feasible due to the compact skinning representations. Finally, rest and arbitrary pose animation editing tools are also supported. Next section reports on how pose editing is supported by our methods.

Animation Editing

Similar to displacement editing presented by [55, 68], geometry editing defined in the rest pose is allowed to be automatically propagated at the subsequent poses. However, previous methods further optimize skinned approximations using rest-pose displacement corrections [77], limiting users to perform efficient editing on the original reference mesh. However, our novel weight corrections could be applied to correct approximations without modifying the rest-pose. On the other hand, our p2p-skinning scheme can handle efficient editing of arbitrary key-frames with the extra cost of recomputing the transformation fitting moving to the newly edited pose from its previous one.

Let p_e be the edited pose and $\widetilde{Q}^e_b$ the matrix that transforms pose p_{e-1} to the edited one (see Figure 4.1). An LBS representation is computed for all poses after the edited one using the following formula:

$$T^t_j = T^{t,e-1}_j \left(\sum_{b=1}^{B} w_{b,j} \widetilde{Q}^e_b \right) T^{e-1,1}_i, \tag{4.18}$$

$$T^{t_0,t_1}_j = \prod_{t=t_0}^{t_1} \left(\sum_{b=1}^{B} w_{b,j} Q^t_b \right) \tag{4.19}$$

Similarly, animation editing can also be supported using dual quaternions. However, we omit this discussion since it does not introduce any novel techniques. Figure 4.2 demonstrates the use of our method for producing approximate skinning and then modifying the rest-pose. The result of arbitrary editing on a deforming skirt animation is illustrated in Figure 4.3.

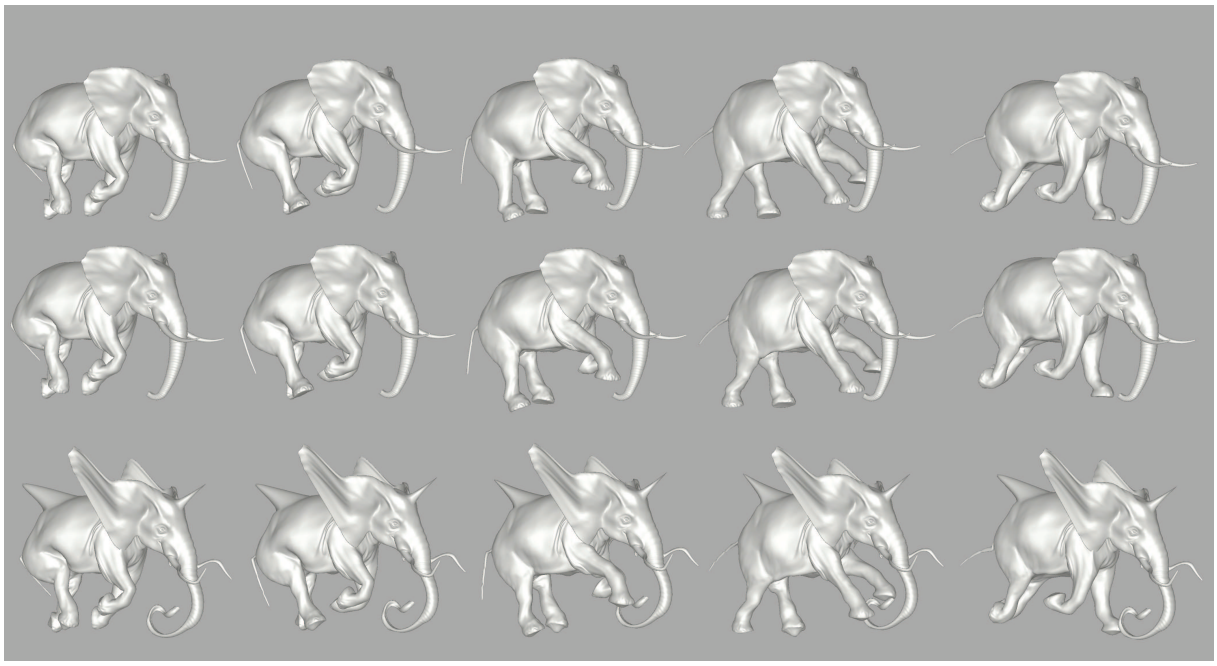


Figure 4.2: (top row) The original animation sequence. (middle row) The approximate animation sequence using our p2p-skinning. (bottom row) The result of editing the reference pose and subsequently applying the pre-computed pose-to-pose transformations derived previously.

4.2 Experimental Study

We evaluate our proposed skinning technique with respect to performance and quality under a set of various testing inputs. These include rigid, highly-deformable and hybrid mesh animations. Table 4.1 summarizes the geometry properties and proxy joints details for each animation. The experiments were performed on a Intel Core i7 870 (8M Cache, 2.93 GHz, 8 threads) CPU using multi-threaded implementation.

4.2.1 Performance Analysis

Figure 4.4 offers a performance comparison between affine (LBS) and rigid (DQS) transforms of an articulated (horse - 8431 vertices) and a highly-deformable (tablecloth - 4225 vertices) animation. While for small models the average fitting times are comparable, we observe a considerable gain of the rigid approach when we move to a higher geometry

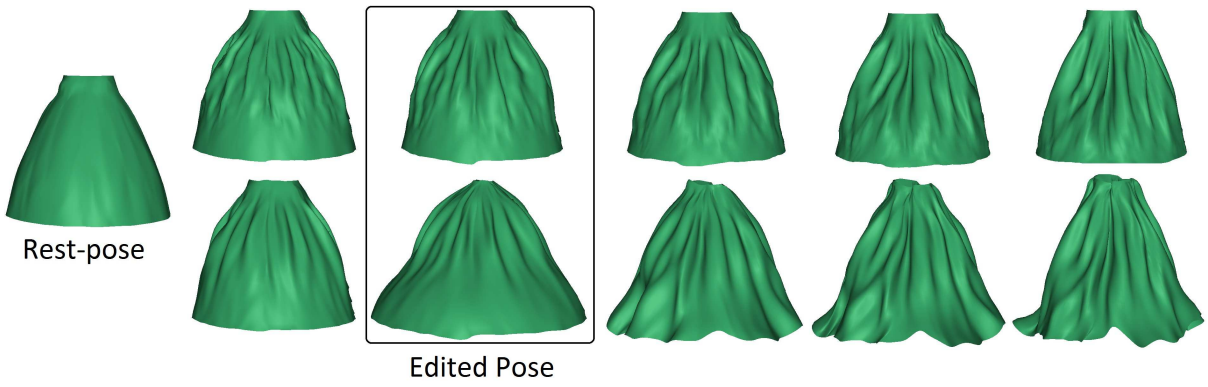


Figure 4.3: (top row) The original animation sequence. (bottom row) The result of editing the second pose and subsequently applying the pre-computed pose-to-pose transformations.

resolution.

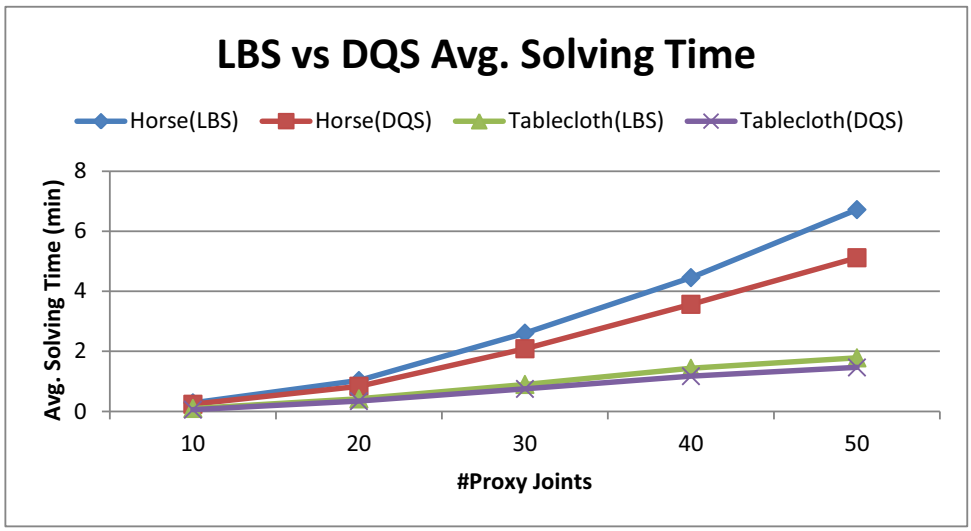


Figure 4.4: Fitting times for the LBS and DQS variants of our p2p-skinnning method.

Table 4.1 demonstrates the computation time for solving the fitting system for SAD, FESAM and p2p-skinning methods. We observe that our method is considerably faster as compared to FESAM. On the other hand, SAD is superior than our method due to the processing cost of vertex and weight corrections utilized by our framework.

4.2.2 Quality Analysis

For measuring the mean skinning approximation error, we use the translation invariant E_{RMS} metric (see Section 2.3.4). Table 4.1 demonstrates the superior skinned approximation results of p2p-skinning when compared to the SAD method. Conversely, p2p-skinning is slightly worse than the optimal FESAM method. The testing scenario properties and animations of Figure 4.4 is further used for quality experimental study in Figure 4.5. We

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

Table 4.1: Performance (in seconds) and quality (in E_{RMS}) comparison between SAD, FESAM and p2p-skinning

observe that the DQS variant exhibits consistently an error increase by 15% to 25% as compared to its affine counterpart.

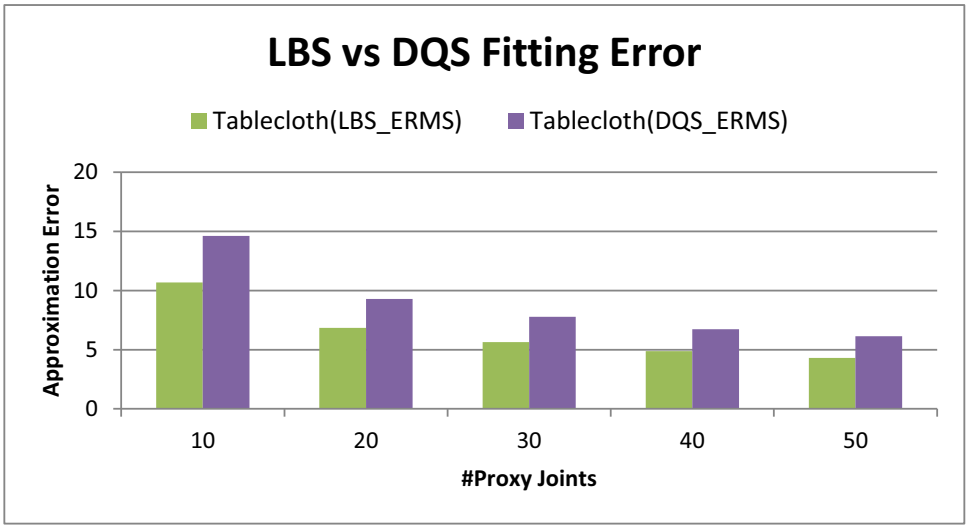


Figure 4.5: Illustrating the fitting error comparison of the LBS and the DQS versions of the p2p-skinning method.

Figure 4.6 illustrates a visual comparison of the (left) SAD, (middle,left) FESAM (middle) p2p-skinning, (middle,right) p2p-skinning with vertex and weight corrections and (right) the original pose. We observe that our method with corrections exhibits visual results comparable to FESAM. Finally, Figure 4.7 demonstrates the high quality result of the p2p-skinning technique when both vertex and weight corrections are employed on a highly-deformed facial expression.

4.3 Conclusions

We have introduced pose to pose approximate rigid and affine fitting schemes exploiting coherence between frames and enabling arbitrary pose editing. Further, corrections

Figure 4.6: (From left to right) An approximated pose using SAD, using FESAM, using p2p-skinning, using p2p-skinning with corrections and finally the original pose for comparison purposes.

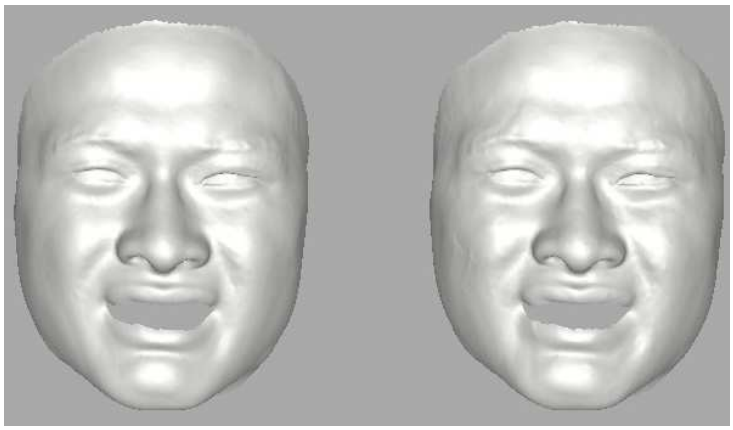


Figure 4.7: (left) An original facial expression pose from an animation sequence. (right) The corresponding approximated pose using our technique with weight and vertex corrections.

have been introduced that decrease significantly the approximation error with no additional storage requirements. Experiments have demonstrated the characteristics of this system in terms of efficiency and accuracy. Finally, visual results have been presented to demonstrate the editing capabilities provided by our novel scheme.

Chapter 5

S-buffer: Sparsity-aware Multi-fragment Rendering

A number of conventional methods exist that simulate complex rendering effects in many graphics applications via an A-buffer variant (see Section 1.3.3). However, most use either a fixed storage per pixel or a linked list approach. The major limitations of the latter is the potentially large and possible wasted memory requirements due to their strategy to allocate the same memory for each pixel. On the other hand, heavy fragment contention and random memory accesses result in a performance bottleneck when linked-lists are employed (see Section 2.4.1). Section 2.4 includes notations and background for fragment generation and shading operations.

5.1 Framework Overview

In this section, we introduce *S-buffer* (**SB**) [149], an efficient and memory-friendly algorithm built on the A-buffer architecture on the GPU without relying on linked-lists [167] or fixed-array structures [90]. Inspired by [110, 87], we perform an additional fast geometry pass for accumulating the fragments which influence a pixel into a counter buffer

which enables us to dynamically allocate the exact amount of memory that we shall need (Section 5.1.1). To optimize caching and data bus occupancy, we organize storage into variable contiguous regions (bins) for each pixel. A memory offset buffer computation is initially performed aiming at packing fragments for each pixel in adjacent position of memory (Section 5.1.2). Contrary to linear [87] and common parallel [110] prefix sum for generating per-pixel memory indices, we employ a randomized prefix sum in parallel by exploiting *pixel sparsity* (i.e. the fact that in many scenes there are many *fragmentless* pixels). Then, a subsequent rasterization of the scene is performed to store the out-of-order fragments per-pixel starting from the memory location captured at the address buffer. Finally, a sorting mechanism is employed to reorder the fragments for each pixel before generating the final image (Section 5.1.3). S-buffer successfully integrates into the standard graphics pipeline and can take advantage of features such as multi-sample rendering, GPU tessellation and instancing.

5.1.1 Fragment Count Pass

First, a geometry pass is employed to simultaneously extract the number of fragments affecting each pixel and the total number of fragments generated for all pixels. More specifically, the fragment accumulation can by implemented by turning off depth test and performing for each rendered fragment per pixel either ADD blending *one* into a 32-bit floating pixel format texture (R_32F) or thread-safe increment operations on a 32-bit unsigned integer buffer (R_32UI). Despite that the former solution is slightly faster than the latter one, a full-screen pass is needed to transform the generated buffer from a floating to an unsigned integer pixel format (additional memory consumption). The total number of rasterized fragments is computed by hardware occlusion queries and used to precisely estimate the size of the *node buffer* that will store the information for all fragments (RG_32F, R: color, G: depth).

5.1.2 Memory Referencing

Prefix sums on the counter buffer have been used in [110] to generate the access location of *all* pixels in the node buffer. To avoid overheads for pixels with zero fragments, [87] perform a prefix sum only on the non-empty pixels in a linear fashion, regardless of the order pixels are processed. This can be implemented using one *shared counter* (32_UI) in the GPU memory which can be updated via atomic memory operations provided by the recent APIs. For each pixel processed, the current shared counter value is written out to the pixel local *address buffer* location, followed by an increment of the shared counter value by the pixel fragment count. We can implement both operations simultaneously by the *atomicAdd()* function (see equation 5.1) which is supported on the recent OpenGL APIs.

To alleviate congestion from all pixels trying to update the same memory location, we propose to apply S *multiple* GPU-accelerated shared counters: $C = \{C(0), \ldots, C(S-$

1)}. More specifically, non-empty pixels are decomposed into non-uniform groups using a simple hash function: $H(p) = (p.\mathrm{x} + sc.\mathrm{width} * p.\mathrm{y})\%S$. We associate one shared counter to each group and perform in parallel the linear prefix sums for all groups.

$$\left.\begin{array}{r} p.\mathrm{address} = C(H(p)); \\ C(H(p)) += p.\mathrm{counter}; \end{array}\right\} \Longrightarrow$$
$$p.\mathrm{address} = atomicAdd(C(H(p)), p.\mathrm{counter}); \tag{5.1}$$

, where $C(i)$ define the i-th shared counter and all shared counters are initially set to zero. After the completion of this process, each group of pixels maps to its own memory space by performing a prefix sum on the final values of the shared counters: $C_{pr}(i) = \sum_0^{i-1} C(i)$, where $C_{pr}(i)$ is the i-th resulting memory reference value. An inverse mapping technique is applied to boost by a factor of two the latter prefix sum process using information from the total number of the rendered fragments. We accomplish that by splitting the shared counters into two groups, $G_1 = \{C(0), \ldots, C(\lfloor \frac{S}{2} \rfloor)\}$ and $G_2 = \{C(\lfloor \frac{S}{2} \rfloor + 1), \ldots, C(S-1)\}$. The key idea is to perform forward prefix sum for the G_1 group and inverse prefix sum for the G_2 group. We define as *inverse*, the prefix sum that starts accumulating from the end of the processing set towards the start. Then, the memory offset for each pixel p is computed using the following equation,

$$p.\mathrm{offset} = \begin{cases} A(p), & \text{if } p \in G_1 \\ total\_fragments\text{-}1\text{-}A(p), & \text{otherwise} \end{cases} \tag{5.2}$$
$$\text{where } A(p) = p.\mathrm{address} + C_{pr}(H(p))$$

Figure 5.1 illustrates a simple example of creating memory offsets applying 3 shared counters with forward mapping ($\{C(0), C(1), C(2)\} \in G_1$). A geometry pre-pass calculates the per-pixel fragment counters. We illustrate pixels with the same hash value by painting them with the same color. A sequential prefix sum is applied for each pixel group via atomically updating the associated shared counter using equation 5.1. Without loss of generality, we assume that pixels are processed from the top row to the bottom row. Forward prefix sums are performed to compute $C_{pr}(i)$ mapping each group of pixels to its own memory space. The head memory location for each pixel is finally computed using equation 5.2.

5.1.3 Fragment Storing Pass - Resolve Pass

In this phase, we perform an additional geometry pass to store pixel fragment data to each bin indicated by the location information generated in the previous phase. For each rasterized fragment, we compute the start memory location for the covering pixel using equation 5.2. The associated fragment information is then written out to the given buffer index. The address buffer for the current pixel is then adjusted to the next free space. The process halts when all fragments are stored to the node buffer.

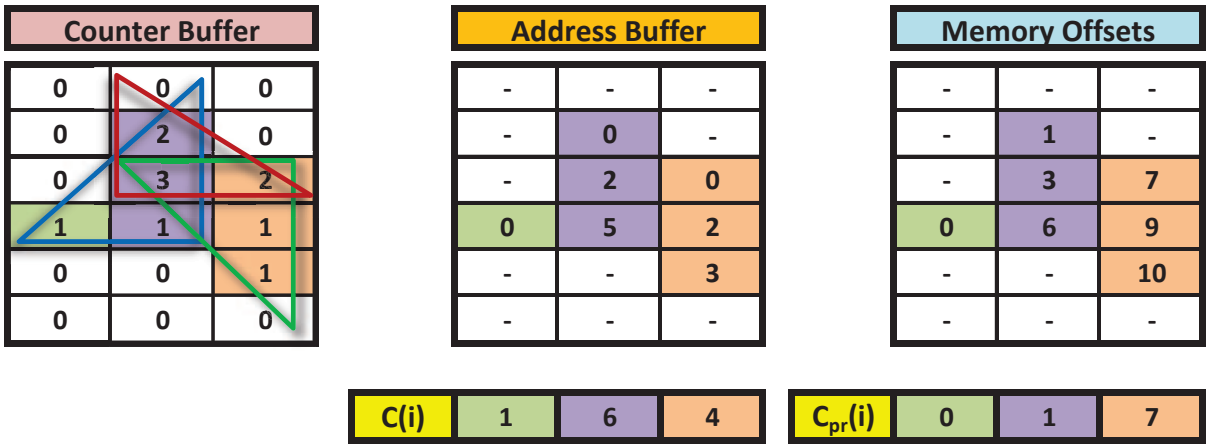


Figure 5.1: S-buffer workflow when rendering a red, a blue and a green triangle.

Finally, we use *insertion sort* to correct the ordering of the captured sample fragments since it performs well when the number of generated fragments per pixel remains small (see also [167]). A large repertoire of multi fragment effects can be supported after sorting. Figure 5.2 illustrates transparency effects and CSG operations using S-buffer.

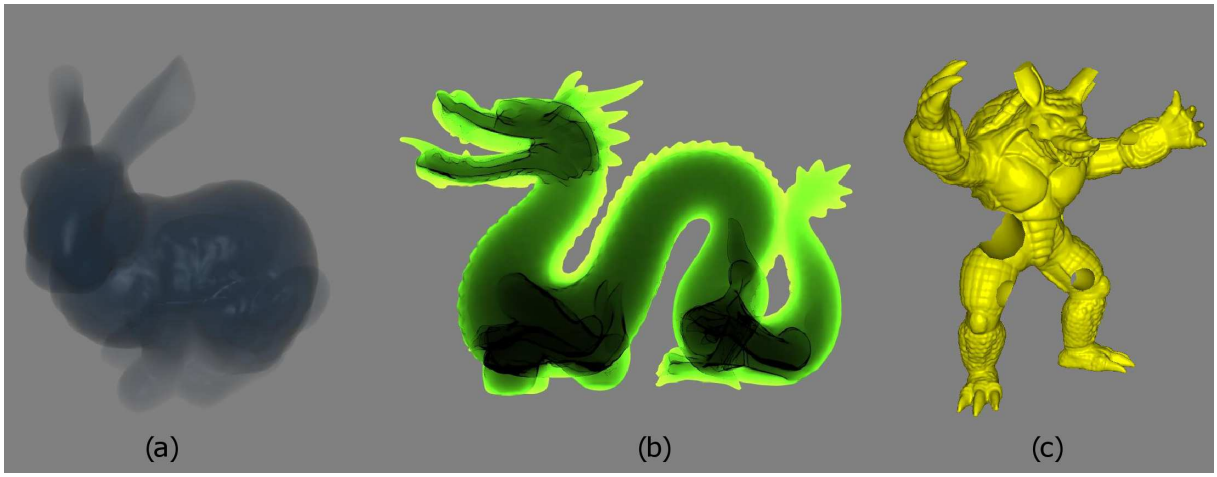


Figure 5.2: Example effects using the S-buffer for multi-fragment processing. (a) Transparency rendering of the Stanford Bunny via accounting for the density between layers. (b) The Dragon model is rendered with translucency attenuating contribution of each fragment with Fresnel's terms. (c) CSG result of applying intersection operations on the Armadillo model.

5.2 Experimental Study

We present an experimental analysis of our S-buffer approach versus the other A-buffer realizations. We have measured performance in terms of FPS and ms and memory requirements in terms of MB for a set of different testing conditions. For the purposes of

comparative time and space complexity evaluation, we have developed *PreCalc_OpenCL*, a faster variation of *PreCalc* [110] which handles memory offsetting using an OpenCL-accelerated parallel prefix sum (provided by NVIDIA Corporation). Moreover, we have implemented *PreCalc_Fixed*, the fastest A-buffer which exploits a one-pass scheme by adapting per-pixel fixed-size arrays based on [110]. This allows prefix sums to be efficiently obtained using a full-screen pass ($p.\text{address} = (p.x + sc.\text{width} * p.\text{y}) * array\_size$). Finally, our variation that uses only one shared counter may be consider as an advanced l-buffer implementation [87]. All methods were implemented using the OpenGL 4.2 API and were tested on NVIDIA GTX 480 hardware (1.5 GB memory, 35 multiprocessors).

Figure 5.3 shows how the performance of the memory-friendly A-buffer variants scales by moving from a sparse to a dense rendering of the Stanford Bunny positioned inside a cube (69463 faces, 12 depth layers) under a 1024×1024 viewport. l-buffer exhibits performance downgrade due to the linearisation of prefix sum which leads to $O(n)$ time complexity, where n is the number of the non-empty pixels. Performance is significantly boosted by increasing the number of S-buffer shared counters. Even with two global counters we match the PreCalc_OpenCL performance when the pixel sparsity remains high. Observe that our buffer exhibits its performance peak using about 30 counters. Since, final memory mapping is obtained through a linear prefix sum on the shared counters, performance starts downgrading when the number increases out of proportion. The Linked Lists technique, using only one geometry pass, has the worst behavior since it suffers from an $O(m)$ complexity, where $m > n$ is the number of generated fragments. Finally, an interesting observation is that the performance of PreCalc_OpenCL converges to S-buffer when the number of used pixels increases rapidly. Even when the rendering scene covers all pixels, our buffer performance is slightly better (7% faster) than the full parallel prefix sum solver of PreCalc_OpenCL.

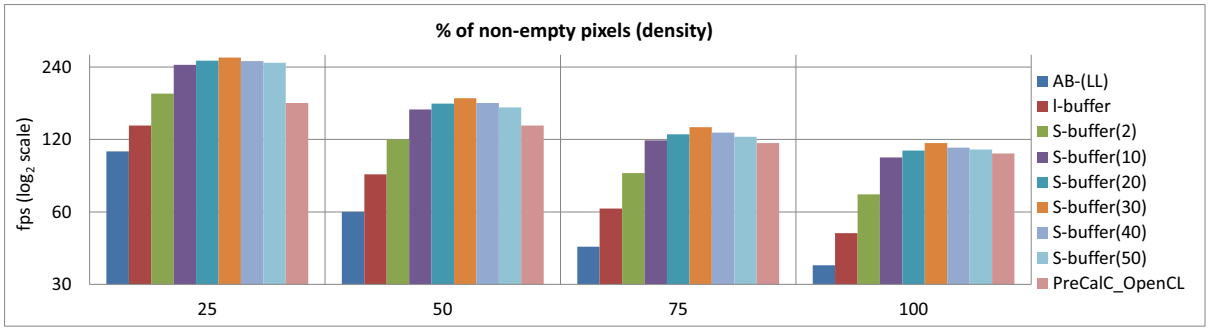


Figure 5.3: Performance evaluation in FPS ($\log_2$ scale) for rendering Stanford Bunny positioned inside a Cube at different clipping stages. S-buffer with 30 shared counters has the best performance for sparse renderings and is comparable with PreCalc_OpenCL in low pixel sparsity.

Figure 5.4 illustrates the performance evaluation of all A-Buffer variants on rendering the Minoan Palace of the Knossos model (109168 faces, 25 max depth layers, $p_d = 45\%$) for a set of different screen resolutions. In general, we observe that fixed-size AB_{FP} and

PreCalc_Fixed solutions outperform memory-aware variants. But this comes with the cost of memory limitations which is discussed later on. KB and KB_{SR} have the worst behavior since they have to carry out multiple iterations for capturing the entire scene information. PreCalc_OpenCL appears to perform quite well despite the synchronization penalties of OpenGL/OpenCL interoperability. S-buffer using 30 counters outperforms the other memory-friendly A-buffer variants, rendering at a 85% to 90% of the optimal frame rate (PreCalc_Fixed). Note that inverse memory mapping boosts S-buffer performance by $(13\%, 10\%, 6\%)$, where percentages in brackets denote of the acceleration for each of three testing resolutions: 640×480, 1024×768, and 1600×1200.

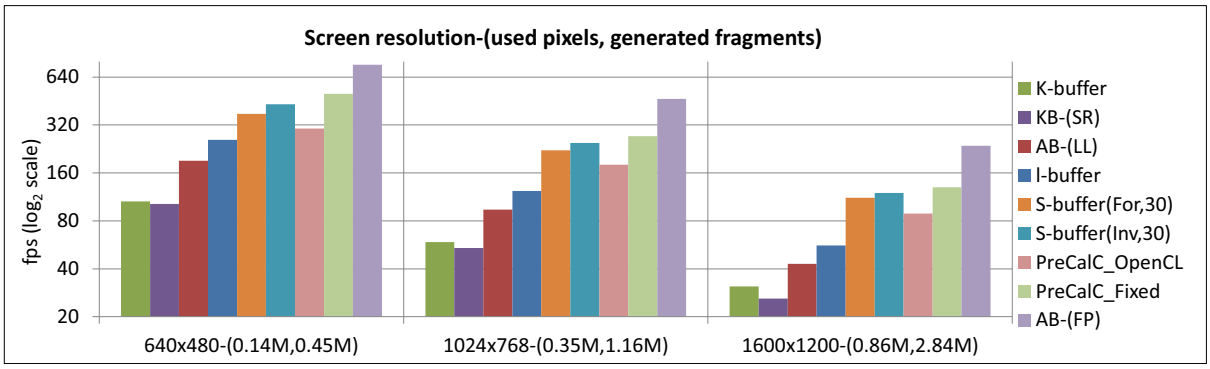


Figure 5.4: Performance evaluation in FPS ($\log_2$ scale) for rendering the Knossos model at different rendering dimensions (in brackets are shown the corresponding used pixels and generated fragments). S-buffer with inverse mapping outperforms the other memory-friendly A-buffer variants.

We further provide a time comparison of the memory referencing step for the buffers that include this step. S-buffer with 30 counters needs $(0.215ms, 0.425ms, 1.08ms)$ to compute memory offsets which is $\approx 10\times$ slower than the fastest PreCalc_Fixed $(0.027ms, 0.05ms, 0.11ms)$. Moving from inverse to forward mapping results at an extra $0.05ms$ cost for all resolutions which explains why inverse mapping boost is decreasing when moving to higher resolutions. PreCalc_OpenCL takes $(1.5ms, 2.45ms, 4.85ms)$ to compute the parallel prefix sum regardless of the pixel sparsity, which is $5\times$ to $7\times$ slower than our method. Finally, fragment-aware Linked Lists exhibits the worst performance by taking $(4.66ms, 10.4ms, 21.98ms)$ which corresponds to an average 20-24 times downgrade.

In the context of storage requirements for the latter scenario, Figure 5.5 shows that AB_{FP} and PreCalc_Fixed lead to increased memory requirements (60.94MB, 159MB, 380.86MB) most of which is not actually used (88%) due to their strategy to allocate the same memory for each pixel. K-Buffer (21.09MB, 54MB, 131.84MB) and KB_{SR} (23.44MB, 60MB, 146.48MB) due to their nature, capture up to 8 fragments per pass and therefore need 30% less memory resources than previous bounded buffers. Conversely, PreCalc_OpenCL (8.02MB, 20.53 MB, 50.10MB) and S-buffer (6.99MB, 17.90 MB, 43.69MB) allocate the exact amount of memory needed since the number of fragment insertions is known apriori. AB_{LL} needs slightly more memory resources for storing memory point-

ers with an extra linked list (8.73MB, 22.35 MB, 51.7MB). However, in cases where the number of fragments varies (camera or mesh animation) overflows may occur.

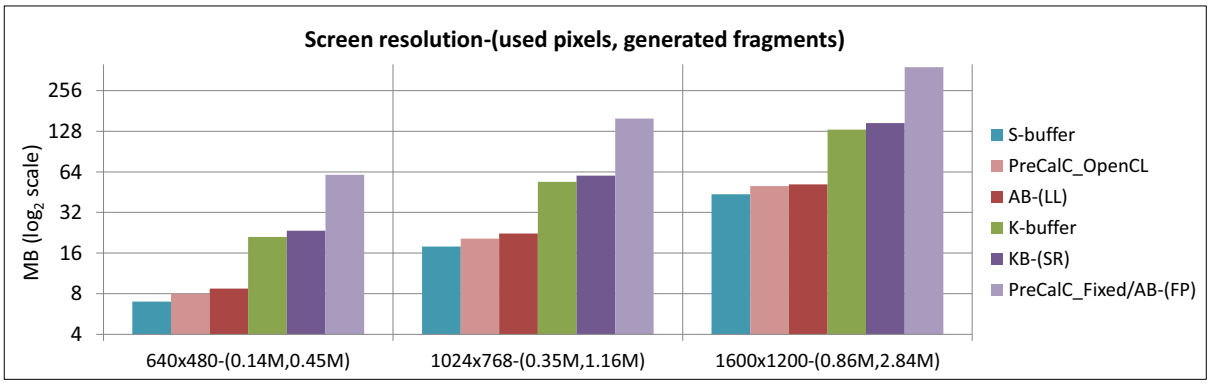


Figure 5.5: Memory evaluation in Mbytes for rendering the Knossos model at different rendering dimensions (in brackets are shown the corresponding used pixels and generated fragments). S-buffer outperforms the rest A-buffer implementations.

5.3 Conclusions

We have presented S-buffer, a two-pass A-buffer implementation on the GPU designed so as to take advantage of the fragment distribution and the sparsity of the pixel-space. An inverse mapping strategy is also presented to slightly improve performance. S-buffer exhibits improved combined memory usage and performance behavior even in low pixel sparsity rasterizations. However, the need of an additional rasterization step results in performance downgrade when compared to AB_{FP}.

Chapter 6

Depth-Fighting Aware Methods for Multi-fragment Rendering

Several approaches have been introduced that process for each pixel one or more fragments per rendering pass, so as to produce a multi-fragment effect. However, multifragment rasterization, more specifically all currently proposed depth peeling techniques (see Section 2.4.1), is susceptible to flickering artifacts when two or more visible fragments of the scene have identical depth values. This phenomenon is called *coplanarity* or *Z-fighting* and incurs various unpleasant and unintuitive results when rendering complex multilayer scenes (see Section 1.2).

6.1 Correcting Multi-fragment Rendering Pipelines

In this section, we investigate two approaches to treat fragment coplanarity in image space that can be applied to several depth peeling methods [150]. Both approaches can be successfully integrated into the standard graphics pipeline and can take advantage of features such as MSAA, GPU tessellation and geometry instancing.

First, we introduce an additional term to the depth comparison operator (Section 6.2). Second, we present an efficient pipeline that can capture multiple coplanar fragments per depth layer by exploiting the advantages of buffer-based techniques (Section 6.3). The core methodology for these extensions is explained in detail by applying it to the F2B depth peeling method. Then, a brief discussion is provided for applying it to the other depth peeling techniques. We classify our algorithms based on the fragment hit ratio R_h, also called robustness ratio (i.e., the total number of extracted fragments over the total number of fragments). *Robust algorithms* succeed to capture all fragment information of a scene regardless of the coplanarity complexity (i.e., $R_h = 1$). On the other hand, *approximate algorithms* are not guaranteed to extract all fragments (i.e., $R_h \leq 1$). The main advantage of the latter is the superiority of the performance over the robust methods at the expense of higher memory space requirements.

We describe features and trade-offs for each technique, pointing out GPU optimizations, portability, and limitations that can be used to guide the decision of which method to use in a given setting.

6.2 Robust Algorithms

We introduce two robust solutions for peeling the entire scene through single-pass and multi-pass rendering pipelines. The first one extracts a maximum of two coplanar fragments per iteration, implemented with a constant video-memory budget. Each iteration carries out one or more rendering passes depending on the algorithm. The second technique is able to capture at once all fragments that lie at the current depth layer before moving to the next one using dynamic creation of per-pixel linked lists.

6.2.1 Extending F2B

The classic F2B method [36] proposed a solution for sorting fragments by iteratively peeling off layers in depth order (see Section 2.4.2). Unfortunately, fragments with depth identical to the depth layer detected in the previous iteration are discarded and thus not considered in the underlying application. We introduce a robust coplanarity aware variation of F2B (**F2B-2P**) by adapting the F2B algorithm so as to peel all fragments located at the current depth before moving to the next depth layer. The basic idea of this technique is to use an extra rendering pass to count per pixel the (non-peeled) coplanar fragments at a specific depth layer. To extract all coplanar fragments, we use the GPU auto-generated primitive identifier (*gl_PrimitiveID* [128]) that is unique per primitive geometric element and is inherited downwards to fragments produced by this primitive. To avoid artifacts from the first primitive processed by the drawing command which is assigned the number zero, we increase all primitive identifiers by one. This approach is compatible with GPU tessellation and geometry instancing by combining primitive and instance identifiers (*gl_InstanceID* [72]). To decide, at iteration i, which fragments among the remaining coplanar ones to extract, we store the minimum and maximum identifiers (denoted as id^i_{min} and id^i_{max}, respectively) of these fragments:

$$id^i_{min} = \min\{f.\text{id}\}, id^i_{max} = \max\{f.\text{id}\}, \forall\ f.\text{id} \in (id^{i-1}_{min}, id^{i-1}_{max})$$

We define as *non-peeled* a fragment f that has a primitive identifier (denoted as f.id) in the range of the identifiers determined during the previous step $i-1$. This strategy guarantees that all coplanar fragments will survive since:

$$id^1_{min} < id^2_{min} < \cdots < id^2_{max} < id^1_{max}$$

Finally, a subsequent rendering pass extracts the fragment information of the corresponding identifier and decides whether the next depth layer underneath should be processed by accessing the counter information. If the counter is larger than two, we have to keep peeling at the current layer since there is at least one more fragment to be peeled.

We use one extra color texture (with internal pixel format RGBA_32F) to store the min/max identifiers at the RG channels and the counter at the A channel. Querying and counting for the identifier range and the counter may be performed in one rendering pass using 32bit floating point blending operations. When computing the output color, two blending operations are used: MAX for the RGB portion of the output color, and ADD for the alpha value. To query the minimum identifier using maximum blending, we store the negative identifier of the primitive.

To avoid storing the 32bit B component of this texture, modern graphics hardware (via OpenGL 4.0+ API) provide the ability to set individual blend equations for each color output. Thus, two textures can be used, one for the counter (R_32F) and one for detected identifiers (RG_32F) and further operate on them using separate ADD and MAX blending operations, respectively.

A second rendering pass is employed to simultaneously extract the fragment attributes and the next depth layer exploiting MRT. Depth testing is again disabled while the blending operation is set to MAX for all components of the MRT. The custom (under-blending) min depth test is implemented adapting the idea of the min/max depth buffer of DUAL [8] with the use of a color texture (R_32F). If the counter is less or equal than two, then we have extracted all information in this layer. We move on to the next one by keeping (blending) the fragments with depth greater than the previously peeled layer. Otherwise, we discard all fragments that do not match the processing depth. The min and max color textures (RGBA_8) are initialized to zero and updated only by the fragments that correspond to the captured identifiers. The algorithm guarantees that no fragment is extracted twice. Initially, we render the scene so as to efficiently capture only the closest depth layer before proceeding with the counter and identifier computation pass.

The details of this method are shown in Algorithm 6.3, where IN.xxx denote the input texture fields (initialized to zero).

Algorithm 6.3 F2B-2P(Pixel p, Fragment f)

/* 1st Geometry Pass using MAX Blending */

1: **if** $f.\text{z} < -\text{IN.z}$ **then**
2: discard;
3: **end if**
4: $p.\text{color}_{min} := (-\text{IN.id}_{min} == f.\text{id})\ ?\ f.\text{color} :\ 0.0$;
5: $p.\text{color}_{max} := (\ \text{IN.id}_{max} == f.\text{id})\ ?\ f.\text{color} :\ 0.0$;
6: $p.\text{z} \leftarrow (\text{IN.counter} > 2\ \textit{or}\ -\text{IN.depth} \neq f.\text{z})\ ?\ -f.\text{z} :\ -1.0$;

/* 2nd Geometry Pass using MAX and ADD Blending */

1: **if** $(\text{IN.counter} \leq 2\ \textit{or}\ f.\text{id} \in (-\text{IN.id}_{min}, \text{IN.id}_{max}))\ \textit{and}\ (-\text{IN.z} == f.\text{z})$ **then**
2: $p.\text{id}_{min} := -f.\text{id}$;
3: $p.\text{id}_{max} := f.\text{id}$;
4: $p.\text{counter} := 1.0$;
5: **else**
6: discard;
7: **end if**

The drawback of this technique is the increase of the rasterization work as compared to the original F2B algorithm by a factor of two. Moreover, the requirement for per-pixel processing via blending may result to a rasterization bottleneck after multiple iterations.

Understanding the strengths and weaknesses of new generation graphics cards is important for achieving the best performance using this technology. *Pre-Z pass* [112] or *lay down depth first* [26] is a general rendering technique for enhancing performance despite the additional rendering of the scene. Specifically, a *double-speed* rendering pass is firstly employed to fill the depth buffer with the scene depth values by depth testing and turning off color writing. Shading the scene with depth write disabled, results on enabling *early-Z culling*; a component which automatically rejects fragments that do not pass the depth

test. Therefore, no extra shading computations are required.

We introduce the **F2B-3P** technique, an F2B-2P variant which follows the above pipeline. The idea is to carry out the first rendering pass of F2B-2P in two geometry passes. A double-speed depth rendering pass is performed to compute the (next) closest depth layer. Then, by exploiting early-Z culling, we perform counting and identifier queries by enabling blending, turning off depth writing and changing depth comparison direction to EQUAL. The difference from the second pass of Algorithm 6.3 is that depth comparisons inside the shader are not needed, thus minimizing the number of texture accesses. Shading is performed in a subsequent pass by matching the fragments of the extracted identifier set without modifying pixel-processing modes (blending or Z-test) of the previous pass. This modified GPU-accelerated version uses the same video memory resources and performs slightly better than its predecessor in some cases despite the cost of the extra rendering pass.

6.2.2 Extending DUAL

DUAL depth peeling [8] increases performance by applying the F2B method for the front-to-back and the back-to-front directions simultaneously (see Section 2.4.2). To handle coplanarity issues raised at both directions, we have developed a variation of DUAL (**DUAL-2P**), which adapts the F2B-2P algorithm for working concurrently in both directions. We omit the analytic description of this algorithm since it won't offer any useful contribution.

To implement the min depth for front layers and the max depth for back layers per pixel by MAX blending effectively and simultaneously, [8] inverses the depth value and stores it in a different component of a color texture (RG_32F - R: front, G: back). A separate color texture (RGBA_32F) is required for storing the counter and identifier info for the back layer. As compared to the F2B-2P implementation, we need to replace counter texture with an RG_32F format than can capture both the front and back counter. Similarly, a color texture (RGBA_32F) should be used to maintain up to 4 fragment identifiers (RG: front min/max ids and BA: back min/max ids).

Developing manually a min-max depth buffer requires turning off the hardware depth buffer. Thus, we cannot benefit from advanced extensions of the graphics hardware in the DUAL workflow (such as the ones used for F2B-3P). DUAL-2P depth peeling as compared to the F2B-2P and F2B-3P variations, reduces the rendering cost to half by extracting up to four fragments simultaneously. The cost for providing this feature is that it requires twice as much memory space.

6.2.3 Combining F2B and DUAL with AB_{LL}

Yang et al. [167] introduced a method to efficiently construct highly concurrent per-pixel linked lists via atomic memory operations on modern GPU (see Section 2.4.2). Although fast enough for most real-time rendering applications, the creation of these lists may

incur a significant cost on video memory requirements when the number of fragments to be stored increases significantly. We propose two efficient multi-pass coplanarity-aware depth peeling methods (**F2B-LL** and **DUAL-LL**) by combining F2B and DUAL with LL. The idea is to store *all* fragments located at the extracted depth layer(s) using linked-list structures. Coplanarity issues can be easily handled using this technique without wasting any memory.

Two buffers are required that store: (a) linked-list fragment data RGBA_8 in the *node* buffer and (b) reverse chained pointers R_32UI that reference the head of the linked lists of the node buffer in the *head* buffer. The access to the node buffer is managed through a global unsigned int address counter (*next*), which represents the location of the next available space in the node buffer. Each pixel contains only the index(-ices) (F2B-LL: R_32UI, DUAL-LL: RG_32UI) of the last node (*head*) it references. Unsigned integer 32-bit memory atomic operations are used for updating. The rendering workflow of F2B-LL consists of two passes: Firstly, a double speed depth pass is carried out enabling Z-buffering. Secondly, we construct linked lists of the fragments located at the captured depth by changing depth comparison direction to EQUAL and turning off depth writing (which results in early-culling optimizations).

The details of this method are shown in Algorithm 6.4, where *ll*.xxx denote the linked list fields and IN.xxx the input texture fields (initialized to zero).

Algorithm 6.4 F2B-LL Depth Peeling (Linked List ll, Fragment f)

```
/* 1st Geometry Pass using LESS/EQUAL Z-test comparison */
1: if f.z <= IN.z then
2:     discard ;
3: end if
/* 2nd Geometry Pass using EQUAL Z-test comparison */
1: ll.next ← ll.next+1 ;                ▷ where ← denotes an atomic store operation
2: ll.head[ll.next] := IN.head ;
3: ll.node[ll.next] := f.color ;
4: IN.head := ll.next ;
```

Construction of a min/max depth buffer for DUAL-LL disables depth testing which results in an increase of the number of texture accesses and per pixel shader computations. In the context of storage, one extra screen image is allocated for the head evaluation of the back layer. To avoid a slight increase of contention due to the extensive attempts of accessing the shared memory area from both front and back fragments, an additional address counter variable for back layers is used ($next_{back}$). Conflicts between front and back fragments are avoided by employing an inverse memory mapping strategy for the fragments extracted in the back-to-front direction. Specifically, we route them starting from the end of the node buffer towards the beginning.

The key advantage of these techniques over the rest of the robust methods introduced in this thesis is that they can handle fragment coplanarity of arbitrary length per pixel in

one iteration. This results in a significant decrease of the rendering workload. Practically, contention from all threads trying to retrieve the next memory address for accessing the corresponding data has been reduced since coplanarity occurs only for a small number of cases as compared to the original AB_{LL} algorithm.

Despite the fact that order of thread execution is not guaranteed, list sorting is not necessary since all captured fragments are coplanar. Moreover, F2B-LL rendering pipeline is boosted by hardware optimization components. All these lead to efficiently usage of GPU memory and performance increase. Conversely, random memory accesses and atomic updating of *next* counter(s) from all fragment threads may lead to a critical rasterization stall.

6.2.4 Combining BUN with AB_{LL}

Despite the accurate depth-fighting feature of the above proposed extensions, their performance is rather limited when the depth complexity is high due to their strategy to perform multiple iterations. Furthermore, as mentioned above, AB_{LL} may exhibit some serious performance bottlenecks when (i) the total number of generated fragments (storing process) or (ii) the number of per-pixel fragments (sorting process) increases significantly. To alleviate the above limitations, we propose a single-pass coplanarity-aware depth peeling architecture combining the features of BUN and AB_{LL}. In this variation, we uniformly split the depth range of each scene and assign each subdivision to one bucket. Then, we concurrently (in parallel) store all fragment information in each bucket using linked lists.

A bounding box is initially rendered to approximate the depth range of each pixel. Due to the current shader restrictions, we can divide the depth range into $five$ uniformly consecutive subintervals. A *node* buffer (RGBA_8) is used to store all linked-list fragment data from all buckets. We explore a non-adaptive scheme where all buckets can handle the same number of rasterized fragments. The location of the next available space in the node buffer is managed through $five$ global unsigned int address counters ($[next_{b_0}, \cdots, next_{b_4}]$). Each pixel contains $five$ head pointers (R_32UI), one for each bucket, containing the last node ($[head_{b_0}, \cdots, head_{b_4}]$) it processed. Each incoming fragment is mapped to the bucket corresponding to its depth value. The address counter of the corresponding bucket is incremented to find the next available offset at the node buffer. The head pointer of the bucket is lastly updated to point to the previously stored fragment. After the complete storage of all fragments, a post-sorting mechanism is carried out in each bucket sorting fragments by their depth.

The core advantage of BUN-LL is the superiority in terms of performance over the rest of the proposed methods due to its single-pass nature. BUN-LL is faster than AB_{LL} and exhibits time complexity comparable to SB and AB_{FP}. However, unused allocated memory from empty buckets as well as fragment overflow from overloaded ones may arise for scenes with non-uniform depth distribution.

6.3 Approximate Algorithms

To alleviate the performance downgrade of multi-pass techniques we have explored fixed-sized vectors [6, 29] for capturing a bounded number of coplanar fragments. The core advantage of this class of methods is the superiority of performance in the expense of excessive memory allocation and fragment overflow (see Section 2.4.2).

6.3.1 Combining F2B and DUAL with AB_{FP}/KB

We introduce a solution for combining AB_{FP}/KB with F2B and DUAL (**F2B-B**, **DUAL-B**) to partially treat fragment coplanarity. The idea is to adapt the previously described core methodology of linked lists by exploiting bounded buffer architectures for storage.

Similar to AB_{FP}, constant length per-pixel vectors are allocated to capture the fragment data. In the case of DUAL, we have to allocate two buffer arrays for front and back peeling at the same time. Without loss of generality, we use the same length for both buffers. Per-pixel counters (F2B-FP: R_32UI, DUAL-FP: RG_32UI) are used to indicate the array position of the next incoming fragment (*count*). They are also used to store the number of the total captured coplanar fragments. Atomic operations are only applied for incrementing the counter variables. To support efficiently this approach in hardware, we may employ a KB framework in place of AB_{FP}. While KB is restricted by MRT to peel a maximum of 8 fragments, data packing may be used to increase the output (and reduce memory cost) by a factor of 4. Note that, there is no need for pre-sorting and post-sorting, since we peel fragments placed at same memory space (RMWH-free).

The details of combing F2B with AB_{FP} and KB are shown in Algorithm 6.5, where A.xxx is used to define the fixed-size data array, IN.xxx the input textures (initialized to zero) and TMP.xxx the fragment temporary variables. Only the second pass is provided since the first one is the same as in Algortithm 6.4.

Algorithm 6.5 F2B-B(Array a, Pixel p,Fragment f)

```
  /* using KB: F2B-KB */
1: for i = 0, a.length do
2:     if a[i] == 0 then
3:         a[i] := f.color; break ;
4:     end if
5: end for
  /* using AB_FP: F2B-FP */
1: TMP.counter := IN.counter+1 ;
2: IN.counter ← (TMP.counter == a.length) ? 0 : TMP.counter ;
3: a[TMP.counter−1].color := f.color;
                                  ▷ where ← denotes an atomic store operation
```

The major advantage of this idea is that by updating atomically only per-pixel counters

no access of shared memory is attempted which results in significant performance upgrade. Performance is degraded when KB is used due to concurrent updates, but this is a useful option when advanced APIs are not available. KB_{SR} is a promising option but in this context it is ruled out since it cannot support MSAA, stencil and data packing operations. Note that attribute packing except from extra memory requirements, requires additional shader computations and imposes output precision limitations on fragment data (32bit).

A simplified example that illustrates the peeling behavior of the base-methods and our proposed extensions is shown in Figure 6.1. The scene consists of three objects of different color with the following rendering order: green, coral and blue resulting in the green having the smallest and blue the largest primitive identifiers. A ray starting from an arbitrary pixel hits the scene at three depth layers, where three and two fragments overlap at the first and the third layer, respectively.

6.4 GPU Optimizations for Multi-pass Rendering

The previous sections introduced extensions of the multi-pass depth peeling algorithms to cope with coplanar fragments. In this section, we propose an optimization making use of various features of modern GPUs so as to improve the performance when multi-pass rendering is performed on multiple objects. Inspired by the *occlusion culling* [129] mechanism explained at Section 2.4.1 (where geometry is not rendered when it is hidden by objects closer to the camera), we propose to avoid rendering objects that are completely peeled from previous iterations. By skipping the entire rendering process for a completely peeled object, we reduce the rendering load of the following rendering passes.

Similarly to occlusion culling, we substitute a geometrically complex object with its bounding box. If the bounding box of the object ends up entirely behind the last captured contents of depth buffer, we may cull this object at the geometry level (see Figure 6.2). This is easily realized by hardware occlusion queries. Due to the observation that objects that are culled during a specific iteration, will be always culled in the successive ones, we reuse the results of the occlusion queries from previous iterations [10]. This leads to a reduction of the number of issued queries eliminating CPU stalls and GPU starvation.

Finally, we avoid the synchronization cost between the CPU and GPU required to obtain the occlusion query result, be using *conditional rendering* [128]. Note that conditional rendering can also be used to automatically halt the iterative procedure of multi-pass rendering methods.

6.5 Experimental Study

We present an experiment analysis of our extensions focusing on performance, robustness and memory requirements under different testing scenarios. For the purposes of compari-

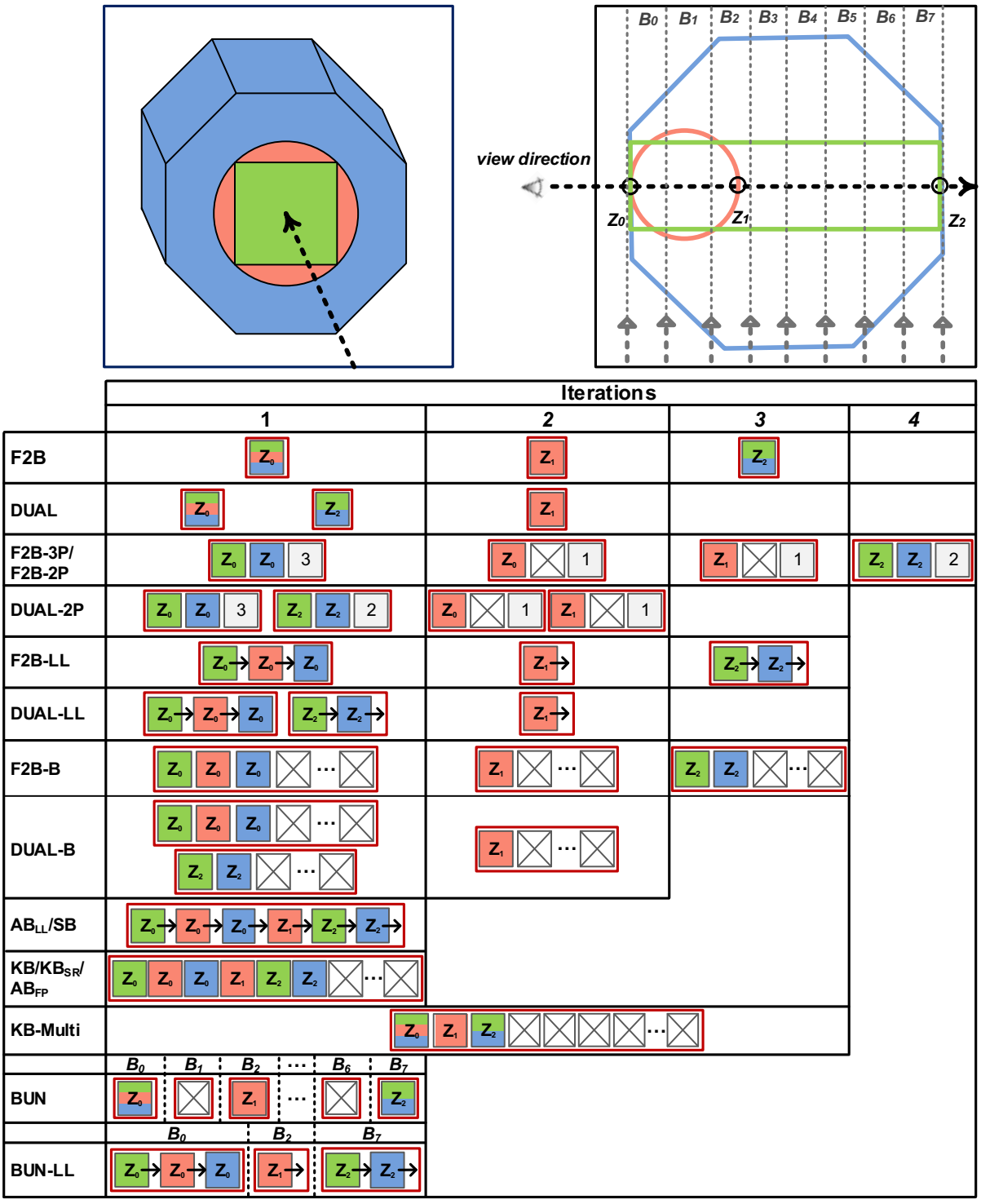


Figure 6.1: Overview of peeling results for our proposed methods and their predecessors. Z_0, Z_1 and Z_2 indicate the depth layers captured by ray casting (black dashed line) and $B_0, B_1, \cdots, B_7$ the uniformly distributed buckets. Each column shows the produced output of each method for the corresponding iteration: extracted fragment(s) painted with the color of an object and coplanarity counters. Squares painted with more than one color demonstrate z-fighting artifacts (it is undefined which fragment might win the z-test). To distinguish between fragments of the same object, we have included their depth value to their associated square.

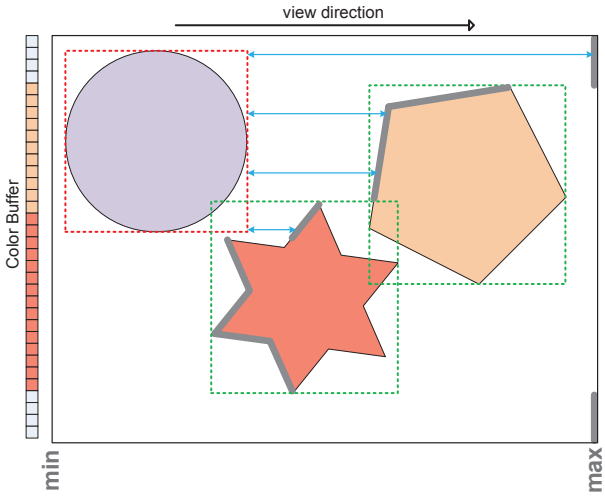


Figure 6.2: A sphere is efficiently culled and thus does not need to be rendered for the remaining iterations since its bounding box lies entirely behind the current depth buffer (*thick gray line strips*).

son, we have developed **F2B2**; a two-pass variation of F2B that uses double speed Z-pass and early Z-culling optimizations. Our methods successfully integrate into the standard graphics pipeline and take advantage of features such as MSAA, GPU-based tessellation and instancing. Methods that do not exploit the AB_{FP} or the LL structures can be used in older hardware. All methods are implemented using OpenGL 4.2 API and performed on an NVIDIA GTX 480 (1.5 GB memory, 35 multiprocessors).

We have applied our coplanarity-aware peeling variants on several depth-sensitive applications (transparency effects, wireframe rendering, CSG operations, self-collision detection, coplanarity detection) demonstrating the importance of accurately handling scenes with z-fighting (see Figures 6.8 and 6.9).

Table 6.1 presents a comparative overview of all multi-fragment raster-based methods with respect to memory requirements, compatibility with commodity and state of the art hardware, rendering complexity, coplanarity accuracy and other features.

6.5.1 Performance Analysis

We have performed an experimental performance evaluation of all our methods against competing techniques using a collection of scenes under four different configurations. Except from the first scene which is evaluated under different image resolutions, the rest of the tests are rendered using a 1280×720 (HD Ready) viewport.

Impact of Screen Resolution

Figure 6.3 shows how the performance scales by increasing the screen dimensions when rendering a crank model (10K primitives) whose layers varies from 2 to 17 and *no* coplanarity exist. In general, we observe that our variants perform slightly slower than their

Acronym	*Description*	*Per iteration*		*Total*	*Conditional rendering*	*Double speed z- pass*	*Early z-culling*	*Old API*	*Modern API*	*Handles coplanarity*	*Robustness ratio*	*on primitives*	*on fragments*	
F2B	*Front-to-back depth peeling*	1	1	D	x	x	x	3		x	D/C(Zall)			
F2B2	*Two-pass F2B depth peeling*		2	2D		√	√							
F2B-2P	*Two-pass Z-fighting free F2B*	2		C(Zall)		x	x	12	10		1			
F2B-3P	*Three-pass Z-fighting free F2B*		3	3C(Zall)/2	√	√	√			√				
F2B-LL	*Z-fighting free F2B using Linked Lists*	C(Z)	2	2D				x	2C+4					
F2B-B	*Z-fighting free F2B using fixed-size Buffers*	K ; 4K						K+3 ; 4K+3	K+4 ; 4K+4		∑{K/C(Zi)} ; 4∑{K/C(Zi)}	x	x	√
DUAL	*Dual Depth Peeling*	2	1	D/2+1	x			6		x	D/C(Zall)			
DUAL-2P	*Two-pass Z-fighting free DUAL*	4	2	C(Zall)/2+1	√			24	20	√	1			
DUAL-LL	*Z-fighting free DUAL using Linked Lists*	C(Zf,Zb)		D				x	2C(Zf,Zb)+6					
DUAL-B	*Z-fighting free DUAL using fixed-size Buffers*	K ; 4K						K+4 ; 4K+4	K+6 ; 4K+6		∑{K/C(Zi)} ; 4∑{K/C(Zi)}			
KB	*K-Buffer*	K ; 2K	1	1 to D/K				2K+2 ; 4K+2			K/C(Zall) to 1 ; 2K/C(Zall) to	√	√	
KB-Multi	*Multipass K-Buffer*		1...K ; 2K	1 to K ; 1 to 2K	x					x				
KB_{SR}	*Stencil Routed K-Buffer*	K	1	1 to D/K	√	x	x	3K+2		√	K/C(Zall) to 1			x
BUN	*Bucket Uniform Peeling*	2K ; 4K	1	D/(2K) to D/2 ; 1				4K+2		x	D/C(Zall)		x	
BUN-LL	*Z-fighting free BUN using Linked Lists*	all	1	1				x	3C(Zall)+8 to		*overflow* to 1		√	
BAD	*Bucket Adaptive Peeling*	4K	4	4	x			6K+3		x	4K/C(Zall)	x	x	√
AB_{FP}	*A-Buffer using fixed-size Arrays*	all	1	1				x	2D+2	√	*overflow* to 1		√	
AB_{LL}	*Linked-list based A-Buffer*								3C(Zall)+3					
SB	*S-Buffer: Sparsity-aware A-Buffer*		2	2					2C(Zall)+3					
K = buffer size (max=8 for all except from AB_{FP}), {color, depth} attribute of fragment = {32bit, 32bit}. Video-memory is measured in *mb(=4 bytes)*.														
D = max{depth}, C(Z) = [# of coplanar fragments at depth Z] , C(Zall) = ∑{C(Zi)}, C = max{C(Zi)}, where C(Z) ≥ 1 and C(Zall) ≥ D. *overflow* = 1 - (max_memory/needed_memory).														
In A ; B, A denotes the layers/memory/ratio/sorting for the basic method and B for the variation using attribute packing.														

Table 6.1: Comprehensive comparison of multilayer rendering methods and our coplanarity-aware variants.

predecessors due to the extra rendering passes (around 30% in average). Our dual variants perform faster at low-resolutions as compared to the corresponding front-to-back ones since they need half the rendering passes. Similar performance behavior moving from low to high screen dimensions is observed between F2B-2P and F2B-3P. GPU optimizations becomes meritorious when image size is increasing rapidly.

AB_{FP} and SB are highly efficient in this scenario due to the low rate of used pixels that require heavy post-sorting of their captured fragments. DUAL-FP has the best performance from all proposed multi-pass variants, which is slightly worst than DUAL (from 6% (low resolution) to 18% (high resolution)). However, it achieves speed regression by a factor of 2 to 4 as compared to the SB and FAB methods, respectively. This is reasonable since we iteratively render the scene up to 18 times to extract all layers. We further observe that DUAL-2P and DUAL-KB perform quite well in low screen resolution but exhibit significant performance downgrade in the higher ones. Finally, rendering bottlenecks appear in all LL-based methods when the resolution is increased due to higher fragment serialization.

Impact of Coplanarity

Figure 6.4 illustrates performance for rendering overlapping instanced Fandisk objects (1.4K primitives). We observe that F2B-3P outperforms F2B-2P and DUAL-2P, enhanced by the full potential of GPU optimizations. Similar behavior is observed for F2B-FP as compared to its corresponding dual variation. Conversely, DUAL-LL performs better than F2B-LL alleviating the increased fragment contention at high instancing.

AB_{FP} extensions exhibit improved performance as compared to constant-pass ones despite of they have to carry out multiple rendering iterations. This is reasonable since

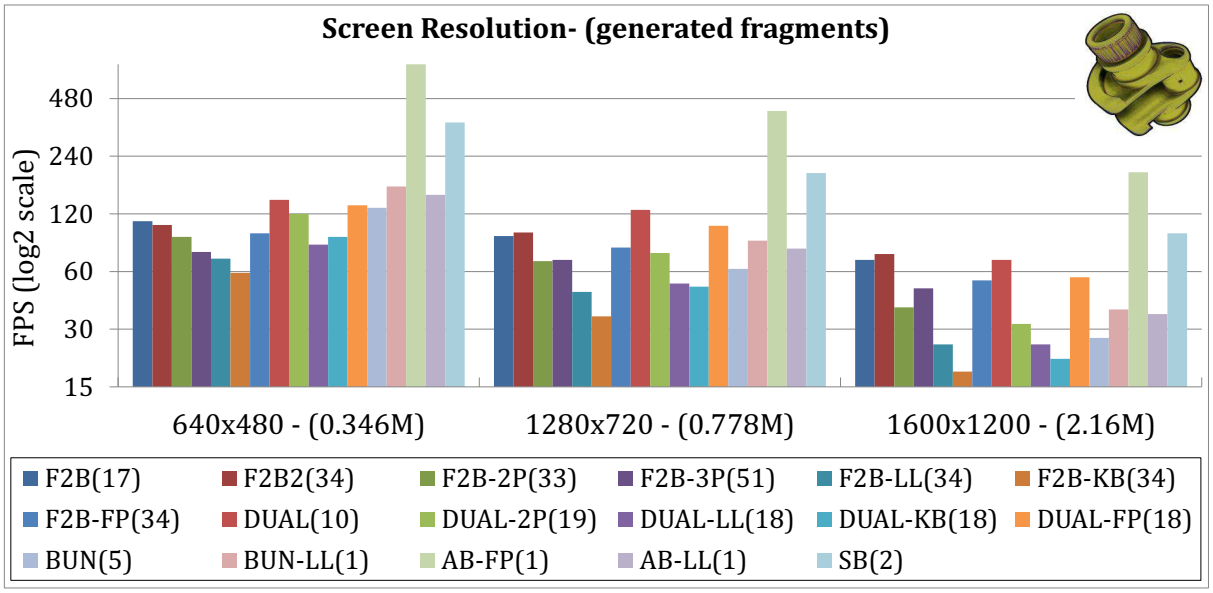


Figure 6.3: Performance evaluation in FPS (log_2 scale) on a scene where no fragment coplanarity is present at different rendering dimensions. Our AB_{FP}-based extensions exhibit slightly worse performance than their base-methods (10% in average). Rendering passes carried out for each method are shown in brackets.

these buffers have to sort the captured fragments resulting in a rendering stall. Finally, BUN-LL is slightly superior than LL and SB, but again is not suitable for scenes with high concentration of fragments in small depth intervals.

Impact of High Depth Complexity

Figure 6.5 illustrates performance comparison of the constant-pass accurate peeling solutions when rendering three uniformly distributed scenes that consists of high depth complexity: Sponza (279K primitives), Engine (203.3K primitives), Hairball (2.85M primitives). We observe the superiority of our BUN-LL over the AB_{LL} and SB methods regardless of the number of generated fragments due to the reduced demands for per-pixel post-sorting of the captured fragments. On the other hand, thread contention in the BUN-LL storing process results at a performance downgrade as compared with AB_{FP} when the rasterized fragments are rapidly increased.

Impact of Geometry Culling

Figure 6.6 illustrates how the performance scales when our geometry-culling is exploited at three representative front-to-back peeling methods under a set of increasing peeling iterations (similar behavior is observed for the rest variations). The scene consists of three non-overlapping, aligned at Z-axis, Dragon models (870K primitives, 10 depth complexity). The scene is rendered from a viewport that the third dragon is occluded by the second one which is similarly hidden by the first. We observe that all front-to-back testing

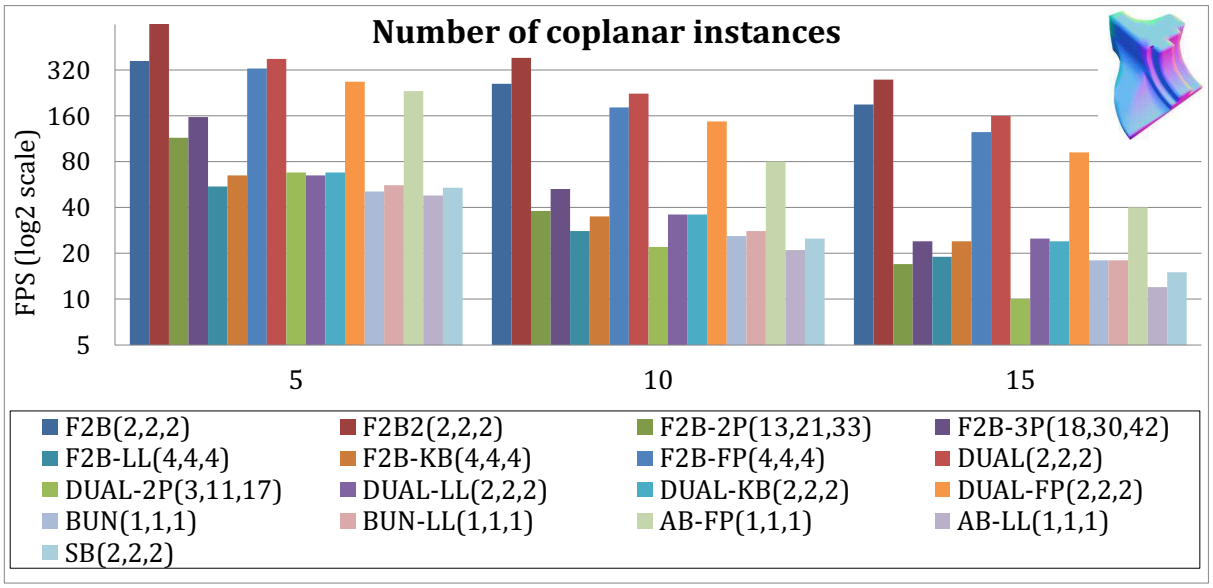


Figure 6.4: Performance evaluation in FPS (log_2 scale) on a scene with varying coplanarity of fragments. AB_{FP} extensions outperform other proposed alternatives and are slightly affected by the number of overlapping fragments. Rendering passes performed for each method are shown in brackets.

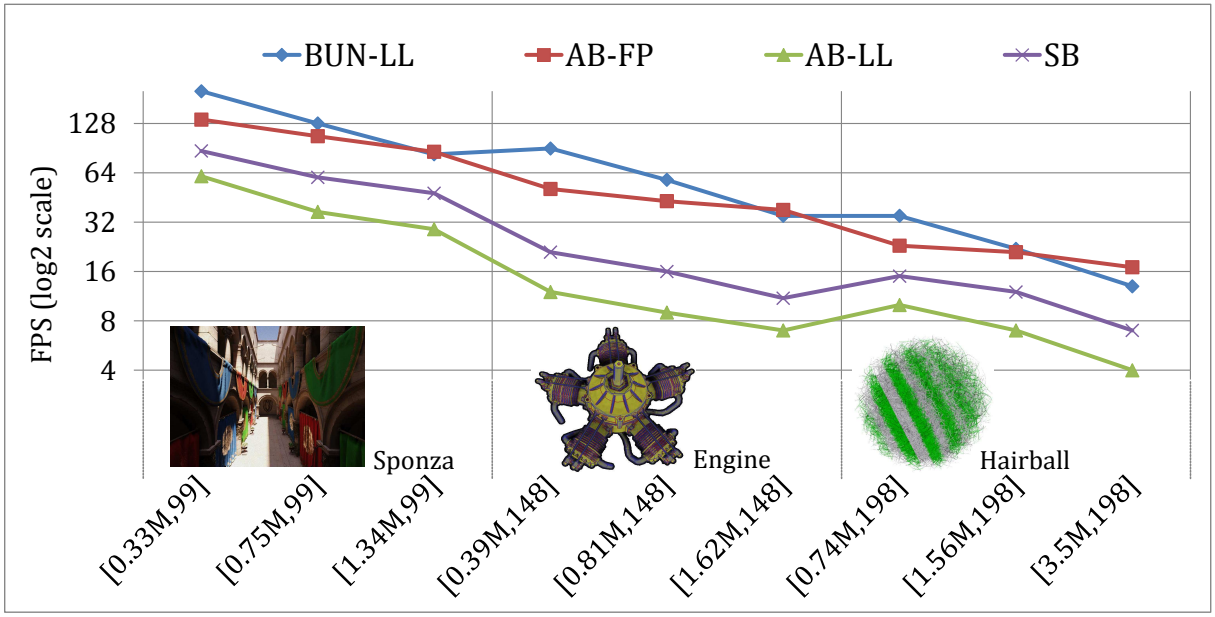


Figure 6.5: Performance evaluation in FPS (log_2 scale) on three uniformly distributed scenes with varying number of fragments and high depth complexity (shown in brackets, respectively). Our BUN-LL outperforms the other buffer-based methods when the fragment capacity remains at low levels.

methods are exponentially enhanced by the use of our early-z geometry culling process when the number of completely peeled objects is increasing. Note that when we have not completely peeled any Dragons, the additional cost of our culling process slightly affects performance (0.01%).

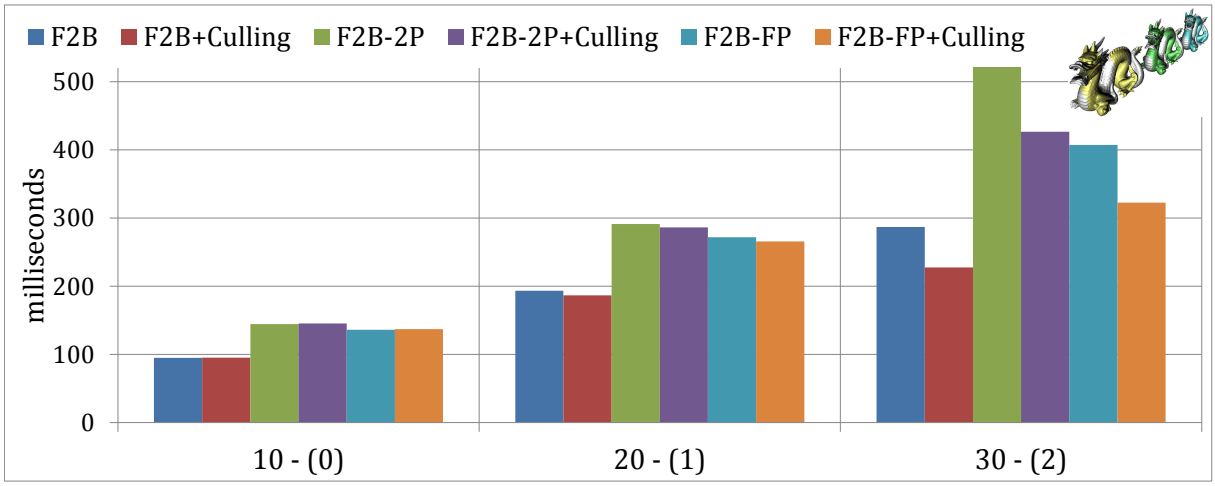


Figure 6.6: Performance evaluation in milliseconds after front-to-back layer peeling a scene without and with enabling our geometry-culling mechanism. The number of completely peeled Dragon models for each peeling iteration is shown in brackets.

6.5.2 Memory Allocation Analysis

Figure 6.7 illustrates evaluation in terms of storage consumption for a scene with varying number of generated fragments (defined by the combination of screen resolution, depth complexity and fragment coplanarity). An interesting observation is the high GPU memory requirements of AB_{FP} due to its strategy to allocate the same memory per pixel. BUN-LL, AB_{LL} and SB require less storage resources by dynamically allocating storage only for fragments that are actually there. However, it will lead at a serious overflow as the number of the generated fragments to be stored increases rapidly.

On the other hand, our multi-pass depth peeling extensions outperform the unbounded buffer-based methods even at high coplanarity scenes. We also observe that robust F2B-2P and F2B-3P methods require slightly less storage than the approximate F2B-KB. Video-memory consumption blasts off to high levels, when data packing is employed for correct capturing high fragment coplanarity. Note that methods that exploit the front-to-back strategy require less memory resources when compared to the dual-direction ones. The same conclusions may be obtained from the formulations of Table 6.1.

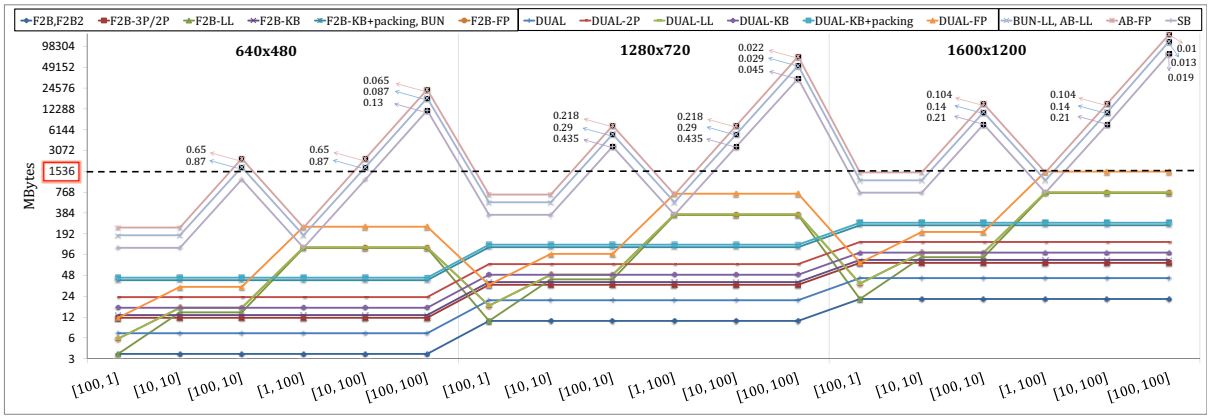


Figure 6.7: Robustness comparison based on memory allocation/overflow (log_2 scale) of a scene with varying resolution and [depth, coplanarity] complexity. Our variants does not consume more than the maximum storage of Nvidia GTX 480 graphics card (dashed line). Note the low robustness ratio of the buffer-based solutions due to the memory overflow.

6.5.3 Robustness Analysis

Impact of Coplanarity

From Table 6.1, we observe that robust variations are able to accurately capture the entire scene regardless of the depth and coplanarity complexity. F2B and DUAL peeling reach their peak when no coplanarity is present. However, robustness is significantly downgraded due to their inability to capture overlapping areas. Multi-pass bucket peeling and its single-pass packed version present similar behavior. Approximate buffer-based alternatives (maximum peeled fragments: without packing ($K = 8$) - with packing ($K = 32$)) are suitable to correctly handle up to 8 or 32 coplanar fragments. Peeling with KB, KB-Multi and KB_{SR} result at memory overflow (hardware restricted to 8 or 16 if attribute packing is used) failing to capture more fragment information. If the scene is pre-sorted by depth, multiple rendering with these buffers will improve robustness. Finally, BUN-LL, AB_{FP}, AB_{LL} and SB perform robustly when fragment storage does not result in memory overflow.

Impact of Memory Overflow

Figure 6.7 shows the needed storage allocated by the memory-unbounded buffer solutions under a scene with varying number of generated fragments. Without loss of generality, we assume that the percentage of pixels covered on the screen is 50% and all pixels have the same depth complexity. Robustness ratio is closely related to memory allocation for these methods (see also Table 6.1). To avoid memory overflow (illustrated by black markers), we have to allocate less storage than we actually need leading at a significant fragment information loss. BUN-LL, AB_{FP}, AB_{LL} and SB robustness is significantly downgraded when the number of generated fragments exceeds a certain point. Conversely, we observe

that our buffer-based extensions perform precisely, allocating less than the maximum storage of the testing graphics card under all rendering scenarios.

6.5.4 Discussion

AB_{FP} has the best performance in conjunction with robust peeling but comes with the cost of extremely large memory requirements. SB alleviates most of the wasteful storage resources running at high speeds, but cannot avoid the unbounded space requirement drawback. Both methods necessitate per-pixel depth sorting resulting at comparable frame rates with BUN-LL when the the number of stored fragments per pixel is high and uniformly distributed.

Multi-pass peeling with primitive identifiers is the best option when accuracy and memory are of utmost importance. AB_{FP} extensions are shown to offer a significant speed up over linked lists variations with satisfactory approximate (or precise when coplanarity is maintained at low levels) results. However, memory limitations should be carefully considered. When modern hardware is not available KB variations might be used to approximate scenes with high coplanarity in the entire depth range.

It is preferred to use front-to-back extensions for handling scenes with low detail under high resolutions. On the other hand, dual extensions performs better when rendering highly tessellated scenes at low screen dimensions.

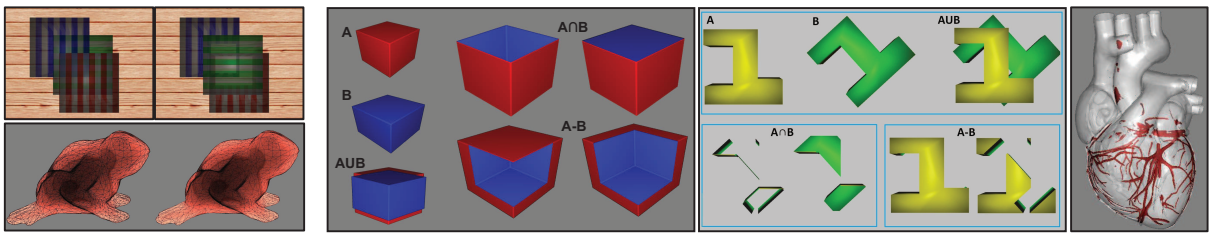


Figure 6.8: Illustrating the image superiority of our extensions over the base-methods in several depth-sensitive applications. (left) (top) Order independent transparency on three partially overlapping *cubes* with and without Z-fighting, (bottom) Wireframe rendering of a translucent *frog* model with and without Z-fighting. (middle) CSG operations rendering without and with coplanarity corrections. (right) Self-collided coplanar areas are visualized with red color.

6.6 Conclusions

Fragment coplanarity is a phenomenon that occurs frequently, unexpectedly and causes various unpleasant and unintuitive results in many applications (from visualization to content creation tools) that are sensitive to robustness. Several (approximate or exact) extensions to conventional multi-pass rendering methods have been introduced accounting for coincident fragments. We have also included extensive comparative results with respect

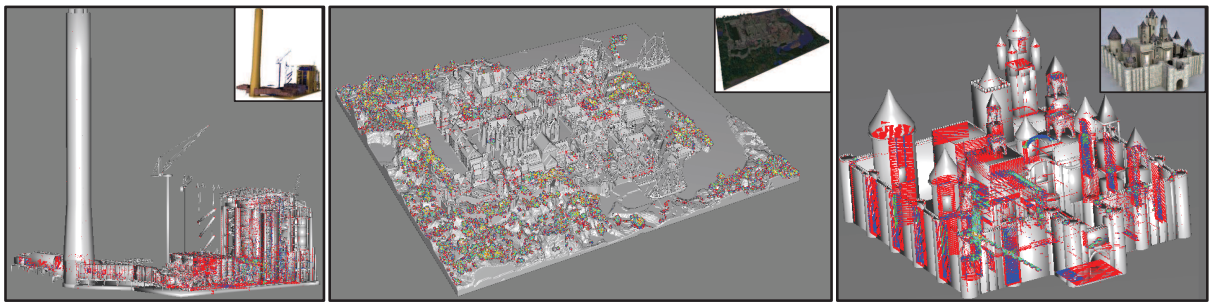


Figure 6.9: Image-based coplanarity detector. (left) *Power plant* ($R_h = 0.98, C_p = 0.285$), (middle) *rungholt* ($R_h = 0.9, C_p = 0.48$) and (right) *castle* ($R_h = 0.88, C_p = 0.81$) scenes are visualized based on the total per-pixel fragment coplanarity: *gray*=none, *red*=2, *blue*=3, *green*=4, *cyan* =5, *aquamarine*=6, *fuchsia*=7, *yellow*=8, *brown*=9. C_p is the average probability for a pixel p to suffer from fragment coplanarity when rendering with the F2B.

to algorithm complexity, memory usage, performance, robustness, and portability. A large spectrum of multi-fragment effects have been considered and used for illustrating the detected differences. We expect that the suite of features and limitations offered for each technique will provide a useful guide for effectively addressing coplanarity artifacts.

Chapter 7

k$^+$-buffer: Fragment Synchronized k-buffer

k-buffer facilitates novel approaches to multi-fragment rendering and visualization for developing interactive applications on the GPU (see Section 2.4.1). Various alternatives have been proposed to alleviate its memory hazards and to avoid completely or partially the necessity of geometry pre-sorting (see Section 1.3.3). However, that came with the cost of excessive memory allocation and depth precision artifacts.

7.1 Framework Overview

In this chapter, we introduce k^+-buffer ($\mathbf{K^+B}$) [151], an efficient k-buffer implementation on the GPU which is free from: (i) geometry sorting prior rasterization, (ii) unbounded memory necessity, (iii) RMW memory-hazards and (iv) depth precision conversion artifacts. Contrary to most of the prior k-buffer alternatives which store and sort the generated fragments on the fly, we follow a faster strategy similar to the one used by the A-buffer construction: The k-nearest fragments are captured in an unsorted sequence, followed by a post-sorting step that reorders them by their depth.

Inspired by [30], we explore a GPU-accelerated *spin-lock* strategy via pixel semaphores to ensure real-time synchronized construction of the unsorted k-front fragments (Section 7.1.1). To alleviate contention (busy-waiting) of distant fragments, we concurrently perform *culling* checks that efficiently discard fragments that are further from all currently maintained fragments (Section 7.1.2). Two array-based data structures are built on the GPU to accurately store the closest per-pixel fragments:

- *max-array*, an array where the maximum element is always stored at the first entry and
- *max-heap*, a complete binary tree in which the value of each internal node is greater than or equal to the values of the children of that node.

Despite its linear complexity, the former performs faster than the latter when the problem size is sufficiently small. For example, order-independent transparency presents high approximation images even with a small core of captured layers ($k \leq 16$). Conversely, plausible photorealistic appearance of hair requires the contribution of a larger set of hair strands ($k > 16$).

To avoid the wasteful pre-allocated storage requirements of pixels that contain less than k fragments (*k-fragmentless* pixels, see Figure 7.1), we have extended our framework by the S-buffer pipeline (Section 7.1.3). An additional geometry pass is performed for counting fragments per pixel, enabling us to allocate the exact amount of memory that we actually need. Memory is linearly organized into *variable contiguous regions* for each pixel, making it feasible to implement both proposed data structures. To our knowledge, this is the first k-buffer implementation with dynamic and precise allocation of the required storage space.

k^+-buffer can also be considered as an unified framework that successfully integrates the functionalities of Z-buffer, k-buffer and A-buffer (Section 7.1.4). The overall framework is described by offering shader-like pseudocode and the fragment processing pipeline. We further highlight features and tradeoffs of our framework, pointing out implementation details and light-weight modifications that can be used to guide the decision of which pipeline alternative to employ in a given setting.

Finally, an extensive experimental evaluation is provided demonstrating the advantages of k^+-buffer over all prior k-buffer variants in terms of memory usage, performance cost and image quality (Section 7.2).

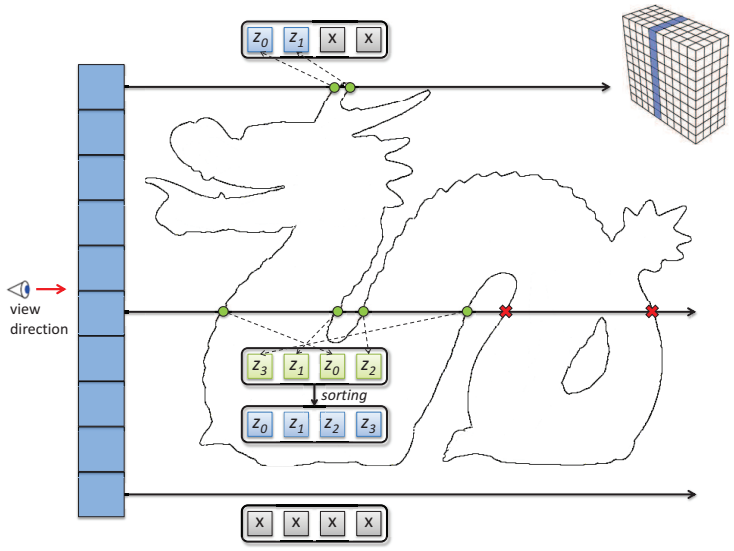


Figure 7.1: Illustrating the construction process of a row of a 4-buffer (highlighted with blue at the top-right thumbnail), when ray casting the dragon model.A significant amount of memory space is wasted at pixels that consist of less than 4 fragments due to the pre-allocation of the same buffer length per pixel.

7.1.1 Spin-lock Strategy

Per-pixel *binary semaphores* are utilized as a synchronization mechanism to ensure fragment exclusive use of the critical storage section. Taking into account the possibility of simultaneous access to the lock, which could cause race conditions, an implementation of an atomic *test-and-set* operation is explored. The calling process obtains the lock if the old value was 0. It spins writing 1 to the variable until this occurs. One way to implement spin-lock strategy employing test-and-set into a pixel shader is shown in the Algorithm 7.6.

Algorithm 7.6 MutualExclusion (Texture s, Pixel p)

```
1: while true do                                   ▷ spin until lock is free
2:     if imageAtomicExchange(s, p, 1) == 0 then
3:         {critical section}                      ▷ exclusive use
4:         imageStore(s, p, 0);                    ▷ release lock when finished
5:         discard;                                ▷ exit shader
6:     end if
7: end while
```

A 32-bit unsigned integer texture with internal pixel format R_32UI is allocated to represent the per-pixel semaphores. At first, a full-screen rendering (**clear pass**) is executed to initialize texture with zeros. Our method is enhanced by the OpenGL's *imageAtomicExchange*(texture lock, ivec2 P, uint V) function which atomically replaces the value V of the atomic object with the argument into texel at coordinate P and returns its original value. Note that there is no need for an atomic operation to perform the lock

release (since the running fragment has exited from the critical section) as opposed to the implementation of [30] where an additional costly atomic exchange is used.

Pixel Synchronization (**PS**) is a graphics extension that Intel has implemented for 4th Generation Intel Core processors with Iris and Iris Pro graphics based on Haswell architecture. PS provides a performance-wise inexpensive mechanism which avoids fragment conflicts in the critical section and ensures that RMW memory operations are performed in submission order [121]. Our framework can be enhanced by the use of PS without remodeling the proposed pipeline. Implementation-wise, a simple call of *beginFragmentShaderOrderingINTEL()* function is necessary to provide fragment serializability. Thus, the per-pixel semaphore-based spin-lock strategy can be omitted (specifically, lines 18-19 and 25-28 in Algorithm 7.8). Avoiding the usage of per-pixel semaphores also results in reduced memory demand. While DirectX11+ and OpenGL extensions are available for Intel graphics cards, we expect that these will be supported in the near future by all manufacturers.

7.1.2 Fragment Capturing

A geometry rendering (**store pass**) is initially carried out to capture the closest fragment data per-pixel in a 64-bit floating point 3D array buffer with internal format of RG_32F, (R for color and G for depth) and k length. Figure 7.1 illustrates a k$^+$-buffer which can hold up to 400 fragments (screen size: 10×10, $k = 4$).

To alleviate the spinning of n generated fragments that do not belong to the closest k, a fast culling mechanism is performed. The idea is to efficiently discard each incoming fragment f_i, $\forall i \in \{0, \ldots, n-1\}$ that has equal or larger depth value ($f_i.z$) from all currently maintained fragments, before trying to acquire the semaphore. Note that i determines the submission order. Let $\mathrm{KB}_i[:] = \{\mathrm{KB}_i[j], j = 0 \ldots k-1\}$ denotes the contents of the k-buffer when fragment f_i has been processed. Initially, we don't discard any incoming fragment until the fragment storage buffer is full ($\forall i < k$). Then, we discard all fragments f_i such that $f_i.z \geq \max\{\mathrm{KB}_{i-1}[:].z\}$. On the other hand, a fragment with $f_i.z < \max\{\mathrm{KB}_{i-1}[:].z\}$ replaces the fragment of the KB with the largest depth value. This strategy guarantees that the k-nearest fragments will always survive since: $\max\{\mathrm{KB}_{n-1}[:].z\} \leq \cdots \leq \max\{\mathrm{KB}_{i-1}[:].z\} \leq \cdots \leq \max\{\mathrm{KB}_{k-1}[:].z\}$.

Note that this process has no impact at the worst case scenario of fragments arriving in descenting depth order. To achieve fragment culling without traversing the entire pixel row for every incoming fragment, we have developed two array-based data structures on the GPU that both store the maximum element at the first array position: (i) *max-array* (**K$^+$B-Array**) and (ii) *max-heap* (**K$^+$B-Heap**). Thus, this operation is performed in constant time. Figure 7.2 illustrates how two incoming fragments are successfully discarded using this formula when the buffer is completely full.

Max-array can be considered as an array where the fragment with the largest depth value is always stored at the first location and the rest are randomly positioned. When an incoming fragment obtains a semaphore, it stores its information in the first empty

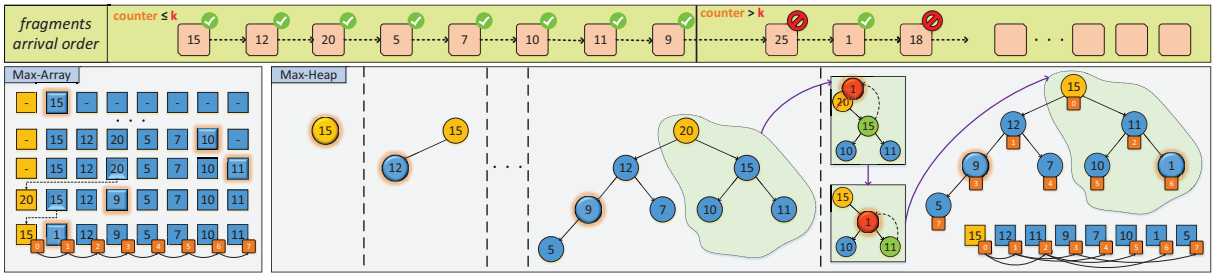


Figure 7.2: Overview of the insertion process of an arbitrary sequence of out-of-order fragments when (left) max-array and (right) max-heap data structures with $k = 8$ are utilized. The incoming fragment in each step is highlighted with a glow effect. When the array is full, fragments with value larger than the maximum captured fragment (yellow-colored) are efficiently discarded ($f_8 = 25$ and $f_{10} = 18$).

entry (O(1)). In this case, a per-pixel *counter* (32-bit unsigned integer texture with internal pixel format R_32UI) is utilized as index and incremented after a successful insertion. Per-pixel counters are initialized to zero during the clear full-screen rendering pass. If the array is full ($counter == k$), it takes the place of the fragment with the larger depth value. Note that since culling mechanism resides outside critical section, an additional checking is mandatory to guarantee correct results. To keep max-array consistent after an insertion on a completely filled array, we find the fragment with the largest depth value (O(k)) and swap it with the newly added fragment (except the latter is the largest one). This process is implemented without the use of any costly atomic memory operations since fragment atomicity is guaranteed.

However when the problem size increases rapidly ($k > 16$), fragment data information is maintained in a max-heap data structure. Max-heap is a complete binary tree (*shape* property) in which all nodes are greater than or equal to each of its children (*heap* property). Max-heap can be implemented using a simple k-sized array without allocating any space for pointers: If the tree root is at index 0, then each element at index $i \in [0, k)$ has children at indices $2i + 1$ and $2i + 2$ and its parent is located at index $\lfloor \frac{(i-1)}{2} \rfloor$. Since the first node contains the largest element, the core pipeline followed by max-array is not altered. Both inserting operations to an empty or a full heap modify the heap to conform to the shape property first, by adding nodes from the end of the heap or replacing the heap root (O(1)). Then, the heap property is restored by traversing up-heap or down-heap (O($\log_2 k$)). Pseudocode for both insertion functions is shown in Algorithm 7.7, where P(f) defines the parent of a fragment f and L(f) and R(f) its left and right children. Figure 7.2 illustrates how both data structures with $k = 8$ are constructed and updated from a number of out-of-order fragment insertions. A representation comparison between max-array and max-heap node pointers is also shown.

Finally, a sorting process is employed to reorder the fragments for each pixel before generating the final image (**resolve pass**). Unsorted fragments are initially copied into a local array before performing the depth sort, as it is relatively faster to perform read-write operations in the register space rather in the global graphics memory. Based on the

Algorithm 7.7 InsertToHeap (Heap h, Pixel p, Fragment f, Int k)

```
1:  procedure UP-HEAP(h, p, f, k)
2:      i := 0;
3:      h[p.counter] := f;                          ▷ Add f to the bottom level of h
4:      while i++ < log_2(k) do                     ▷ Iterate until leaves are reached
5:          if f.z > P(f).z then                    ▷ Compare f with its parent
6:              swap(f, P(f));                      ▷ Swap f with its parent
7:          else
8:              break;                              ▷ Correct depth order, exit
9:          end if
10:     end while
11: end procedure

12: procedure DOWN-HEAP(h, f, k)
13:     i := 0;
14:     h[0] := f;                                  ▷ Replace root with f
15:     while i++ < log_2(k) do                     ▷ Iterate until leaves are reached
16:         C(f) := max{L(f),R(f)};                 ▷ Find f's largest child
17:         if f.z < C(f).z then                    ▷ Compare f with its largest child
18:             swap(f, C(f));                      ▷ Swap f with its largest child
19:         else
20:             break;                              ▷ Correct depth order, exit
21:         end if
22:     end while
23: end procedure
```

number of captured fragments, a mechanism decides which sorting algorithm is applied to the pixel. Despite its quadratic complexity, *insertion sort* is faster for sorting small fragment sequences ($k \leq 16$). When k increases, $O(k \log k)$ sorting algorithms, such as *shell sort*, have better performance [75].

7.1.3 Precise Memory Allocation

Similar to all k-buffer alternatives where k is the same for all pixels, k^+-buffer suffers from potentially large unused memory space allocation of k-fragmentless pixels. For example, Figure 7.1 illustrates the wastefully allocated storage of a 4-buffer for (top) a pixel that consists of 2 fragments and (bottom) an empty-pixel. Note that the value of k is not automatically adjusted based on the rasterized scene and must be carefully set a priori by the user.

Inspired by our S-buffer (see Section 5), we introduce a memory-aware k^+-buffer implementation using two geometry passes (**K$^+$B-SB**). A precise allocation of the required memory space is achieved by performing an initial geometry rendering (**count pass**) which sums up the number of fragments covering each pixel. Contrary to S-buffer where all fragments contribute to the per-pixel aggregation, we bound the number of fragments that affect a pixel by k when $f(p) > k$, where $f(p)$ is the number of generated fragments at pixel p. For each incoming fragment, the per-pixel counter is atomically incremented. When the value of the counter reaches k, the subsequently arriving fragments are discarded. The total size of the k^+-buffer is estimated by accumulating the bounded per-pixel fragments f_k using hardware occlusion queries. Then, the memory offset lookup table (**referencing pass**) is computed in parallel fashion exploiting sparsity in pixel space. Finally, per-pixel counters are reinitialized to zero to guide the subsequent storing phase.

A geometry rasterization is employed to store the most significant fragments to a hybrid buffer scheme starting from the memory offsets computed for each pixel. Knowing its fragment cardinality a priori, each pixel can efficiently choose the fastest way of storing its fragments in either a max-array or a max-heap storage. Note that this is feasible since both data structures are implemented using fixed-arrays. Since max-array structure inserts faster than max-heap when the capacity is not full and k stays small, we apply the following strategy: if $f(p) > k$ and $k > 16$ then we pick max-heap, otherwise we use the max-array data structure as storage buffer (see also Section 7.2).

In terms of performance, accessing global memory for concurrently storing all fragments becomes a significant bottleneck as opposed to the original single-pass k^+-buffer which benefits from the fast operations in the register memory space. Last but not least, the need of an additional geometry rendering step also adds a tessellation-dependent computation cost.

The complete k^+-buffer framework, including the original and its memory-aware version, is shown in Figure 7.3 and Algorithm 7.8, where p.xxx denotes a per-pixel variable, a[i].xxx information located at i position in the buffer array, and f.xxx attributes of each running fragment. insert_empty() and insert_full() are the abstract insertion functions.

Algorithm 7.8 k$^+$-buffer (Array a, Pixel p, Fragment f, Int k)

```
 1: procedure CLEAR(p)                                          ▷ full-screen pass
 2:     p.counter := 0;
 3:     p.semaphore := 0;
 4: end procedure

 5: procedure COUNT(p, k)                                       ▷ geometry pass
 6:     if p.counter < k then
 7:         p.counter ← p.counter+1;                            ▷ bounded fragment accumulation
 8:     else
 9:         discard;
10:     end if
11: end procedure

12: procedure REFERENCING(p)                                    ▷ full-screen pass
13:     compute_pixel_offset(p.counter);
14:     p.counter := 0;
15: end procedure

16: procedure STORE(a, p, f, k)                                 ▷ geometry pass
17:     if p.counter < k or f.z < a[0].z then                   ▷ fragment culling
18:         while true do
19:             if (p.semaphore ↞ 1) == 0 then
20:                 if p.counter < k then                       ▷ enter critical section
21:                     insert_empty(p.counter++);
22:                 else if f.z < a[0].z then
23:                     insert_full();
24:                 end if                                      ▷ exit critical section
25:                 p.semaphore := 0;
26:                 break;
27:             end if
28:         end while
29:     end if
30: end procedure

31: procedure RESOLVE(p)                                        ▷ full-screen pass
32:     if p.counter ≤ 16 then
33:         insertion_sort(p.counter);
34:     else
35:         shell_sort(p.counter);
36:     end if
37:     compute_effect(p.counter);
38: end procedure
                                ▷ where {←, ↞} denote atomic {store, exchange} operations
```

7.1.4 Support of Z-buffer and A-buffer

Without loss of generality, k^+-buffer can also be considered as a unified framework that successfully integrates the functionality of Z-buffer, k-buffer and A-buffer by simply adjusting the value of k. By allocating a single entry per pixel ($k = 1$), our method ensures displaying the closest fragment to the viewer. However, this comes with the additional expense of extra memory requirements and performance downgrade when compared to the hardware depth buffering.

On the other hand, users have to set the value of k large enough to avoid any fragment-overflow ($k = \max_p\{f(p)\}$). More specifically, our framework can be considered as a hybrid scheme that correctly simulates the behavior of AB_{FP} (when K^+B is used) or S-buffer (when K^+B-SB is used). Despite the fact that our framework is not restricted from (i) multiple render targets and (ii) samples of the anti-aliasing buffer, a multi-pass variation may be required to achieve a memory-bounded A-buffer functionality. In this case, $\max_p\{f(p)\}/k$ rendering iterations have to be performed, resulting at a significant workload increase. Performance-wise, max-array structure should naturally be chosen as the fragment storage due to its constant insertion complexity when the array is not full ($\forall p: \ f(p) \leq k$).

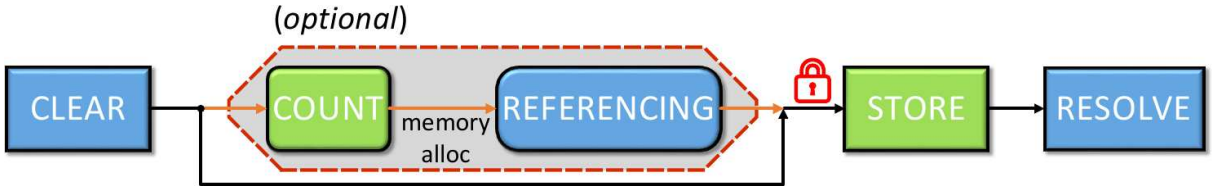


Figure 7.3: Diagram of the k^+-buffer pipeline. Each box represents a shader program. The blue boxes are executed per-pixel using a full-screen rendering pass, while the green ones are executed for each geometry-rasterized fragment.

7.2 Experimental Study

We present an experimental analysis of our k^+-buffer approach against a set of k-buffer and A-buffer realizations focusing on performance, robustness, and memory requirements under different testing conditions. We have measured performance in terms of FPS and ms and memory requirements in terms of MB. For the purposes of comparison, we have developed two variations of $KB\text{-}AB_{LL}$, where instead of using per-pixel linked lists for the A-buffer construction, we have applied either fixed-length (**KB-AB**$_{FP}$) or variable-length (**KB-AB**$_{SB}$) arrays for each pixel. All methods are implemented using OpenGL 4.3 API and performed on a NVIDIA GTX 480 graphics card (1.5 GB memory, 35 multiprocessors).

Figure 7.4 demonstrates the importance of accurately handling multi-fragments for several applications (transparency effects [97], CSG operations [116], and collision detec-

tion [56]). Table 7.1 presents a comparative overview of all k-buffer alternatives with respect to memory requirements, rendering complexity, and other features.

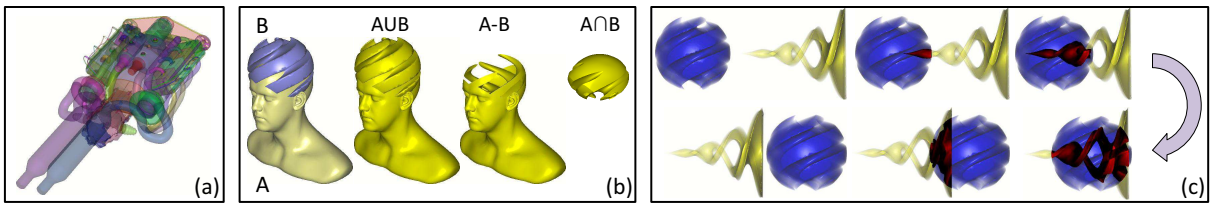


Figure 7.4: A large repertoire of multi-fragment effects can be supported from our framework: (a) Illustrating order-independent transparency of an engine consisting of 195 random-painted components. (b) Rendering boolean operations between a head (A model) and a clipped sphere (B model) surfaces. (c) Detecting collision (highlighted with red color) between a twirl object moving towards a static clipped sphere.

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	1	√	√	8; 16	√	2k; 4k	√
KB-Multi	Multi-pass k-buffer	1 to k	√	√			2k; 4k	
KB-SR	Stencil routed k-buffer	1	√	√	32		3k	
KB-PS	k-buffer using pixel synchronization	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	2	x	x	-	x	2k	
KB-MHA	Memory-hazard-aware k-buffer	1	√	√	8; 16		2k; 4k	
KB-AB$_{FP}$	k-buffer based on A-buffer (fixed-size arrays)	1	x	√	-	√	2n + 1	
KB-AB$_{LL}$	k-buffer based on A-buffer (dynamic linked lists)	1	x	√	-		3f + 1	x
KB-LL	k-buffer based on linked lists	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		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) ≤ k						

Table 7.1: Comprehensive comparison of the prior k-buffer solutions and the introduced k^+-buffer methods.

7.2.1 Performance Analysis

We have performed an experimental performance evaluation of our methods against competing techniques using a collection of scenes under several different configurations. Instead of rendering scenes under different image resolutions, we have used a 854×480 viewport and perform zooming operations defining the percentage of image being rasterized. For a fair comparison, all methods are tested under artificially generated scenes that cover a percentage of screen size (or pixel density: p_d) and produce $n = r \cdot k$ randomly arrived fragments per pixel, where $r \geq 1$.

k-buffer Comparison

Impact of *k*. Figure 7.5 shows how the computation time, for each rendering pass of a set of k-buffer methods, scales by increasing the value of $k = 4, \ldots, 64$ for a scene that

consists of $n = 128$ fragments per-pixel. Except from KB-MDT which needs two passes (namely, Store(Z) and Store), the other memory-bounded methods require to rasterize the scene only once. We observe that our K$^+$B-Array and K$^+$B-Heap perform better than the rest of the techniques for all k values. As expected, K$^+$B-Heap performs better than K$^+$B-Array when moving from low to high k values. Note that the *Resolve* step is more expensive for A-buffer-aware methods, since it has to locate the closest k fragments from all captured ones before sorting. KB-LL and KB-MDT do not perform depth reordering since they both store and sort fragments on the fly. Despite the good performance of KB-MDT for small buffer sizes, the computation cost of storing and sorting depth fragments exhibits a rapid increase for larger k values. This leads to an important conclusion: a future single-pass KB-MDT, which will be enhanced by the expected 64-bit atomic updates on the graphics memory, will present an insignificant performance gain. Note that K$^+$B is up to 25× faster than the current implementation of KB-MDT when $k = 64$. Finally, we observe that *Count* and *Resolve* passes of K$^+$B-SB cost less in terms of computations as compared to the ones of KB-AB$_{SB}$ due to the restricted operations carried out by the former. However, slow fragment storing in global memory from K$^+$B-SB increases fragment spinning which subsequently results at a performance downgrade when the rasterized fragments are significantly increased.

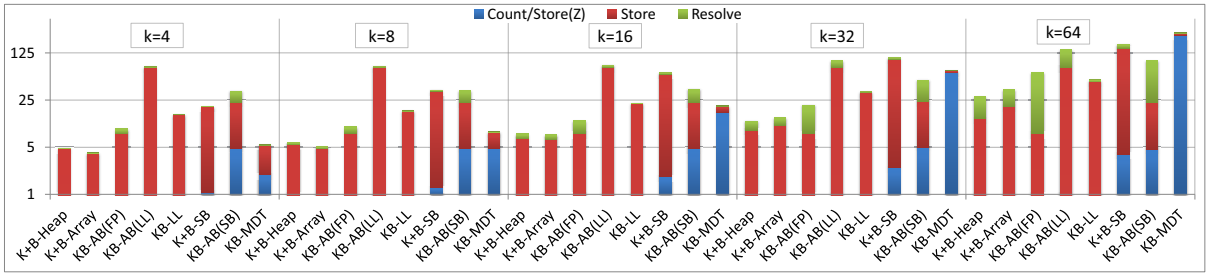


Figure 7.5: Performance evaluation in ms ($\log_5$ scale) of k-buffer variants with increasing k on a scene with constant fragment complexity (128).

Impact of Sorting. Figure 7.6 illustrates performance comparison of our bounded K$^+$B against KB and KB-SR methods for varying k values. All methods are tested under two scenarios where $n = k, \ldots, 1024$ fragments are generated for the $p_d = \{25\%, 75\%\}$ of all pixels. To accurately capture the closest fragments without RMWH from KB and KB-SR, the scene is rasterized in depth order. K$^+$B-Array performs slightly better than K$^+$B-Heap since the first k fragments need O(1) time (compared to O($\log_2 k$)) to be inserted in the array. Due to the sorted arrival fragment order, the remaining fragments do not affect performance since they are successfully culled in both methods. A linear behavior is observed when moving from less ($\{0.21\text{M}, \ldots, 104.93\text{M}\}$) to more ($\{0.61\text{M}, \ldots, 314.82\text{M}\}$) generated fragments. Our methods are superior when compared with KB for all scene configurations. Similar performance conclusions can be made for the not-implemented KB-PS and KB-MHA methods, since they are supposed to perform slightly worse than KB. Despite its efficiency due to the hardware-implemented stencil routing, KB-SR speed

is falling when k reaches higher values. An additional geometry pass must be employed to successfully capture all fragments for KB ($k > 16$) and KB-SR ($k > 32$), resulting in a significant performance downgrade.

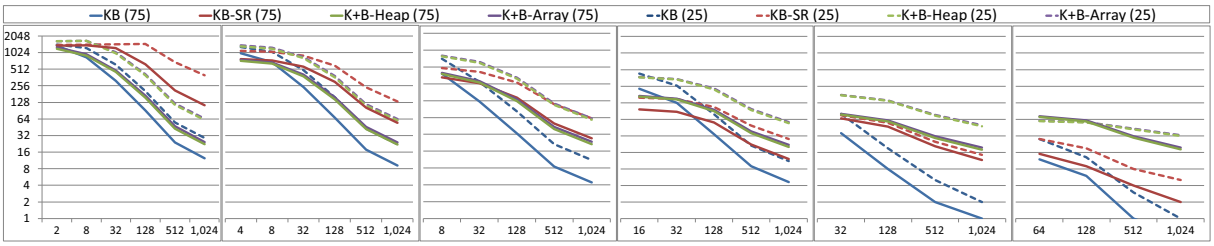


Figure 7.6: Performance evaluation in FPS ($\log_2$ scale) of our bounded K$^+$B methods and the sorting-aware k-buffer methods for a large set of k values. Pixel density is shown in brackets.

Impact of Memory. Figure 7.7 illustrates the performance evaluation in terms of FPS per MB for a testing k-buffer method set when performance and memory are of utmost importance. To construct k-fragmentless pixels, we permit pixels to be influenced by up to $n = 10 \cdot k$ fragments. Thus, we define f_p as the probability of a generated fragment to not be discarded. We observe that K$^+$B-SB is preferred to be used for handling scenes with many empty pixels ($p_d = 25\%$) and small number of rasterized fragments ($f_p = 25\%$). When pixel and fragment densities increase ($p_d = 75\%, f_p = 75\%$), K$^+$B-SB performs better than the rest memory-aware methods. However, K$^+$B-SB behavior is normally worst than the bounded methods since it theoretically performs slower (e.g. one extra pass, storing data at global memory) in conjunction with the small unused memory of the bounded methods. Despite the fast speed of KB-AB$_{SB}$ on sparse scenes, performance is significantly reduced when generated fragments blast off to high levels. Finally, KB-AB$_{FP}$, KB-AB$_{LL}$, KB-AB$_{SB}$ and KB-LL fail to work when fragment allocation results in memory overflow ($k = 64$).

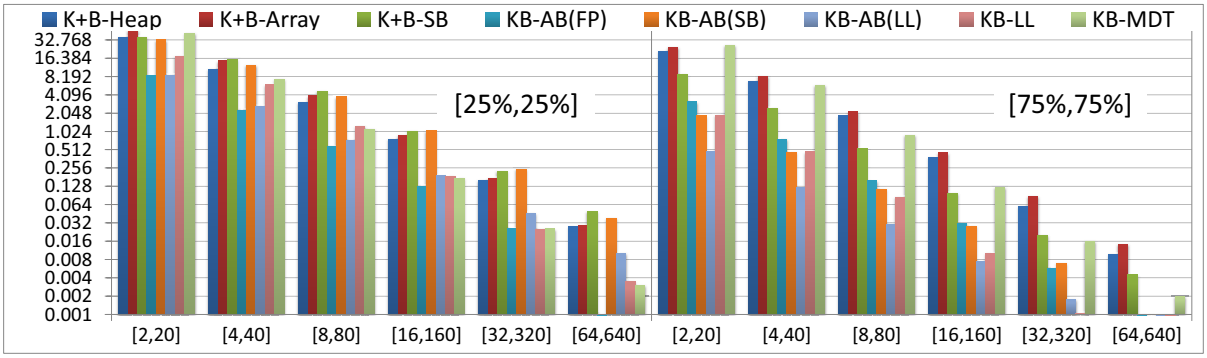


Figure 7.7: Performance evaluation in FPS/MB ($\log_2$ scale) of all k-buffer variants when moving from a scene with $r = 10$ from small number towards a large number of generated fragments.

Impact of Tessellation. Figure 7.8 illustrates how the performance scales by moving from a low (1 level) to a high (64 level) tessellated scene. A representative set of scenes are

used to compare a number of k-buffer methods that aim at capturing 8 fragments. Three scenes are generated where the same number of fragments are rasterized $n = \{16, 40, 80\}$ for $p_d = 50\%$ of the pixels. Different configurations yield similar speed results. In all tests, a small performance impact is observed from the SB-aware methods as opposed to the linear behavior of the rest k-buffers. This is due the fact that the counting geometry pass is not-tessellation dependent.

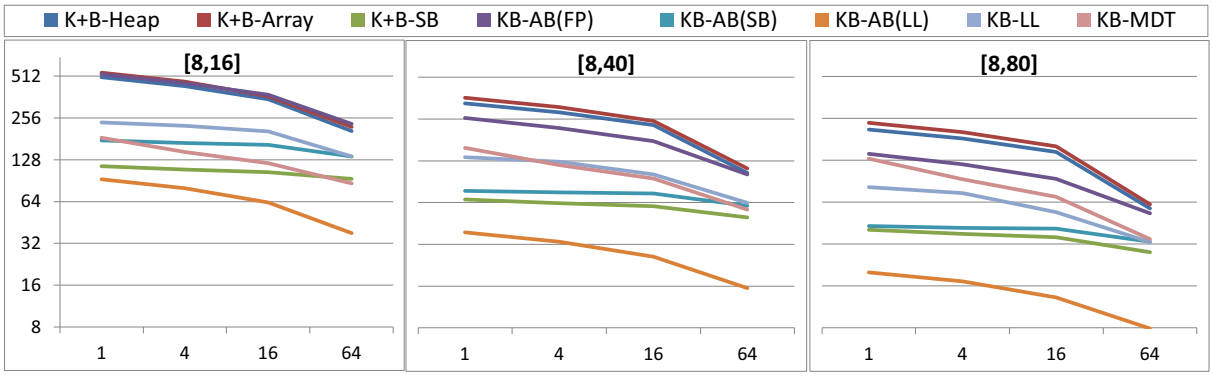


Figure 7.8: Performance evaluation in FPS ($\log_2$ scale) of all k-buffer versions on a scene with varying tessellation resolution $[1, 64]$ and increasing per-pixel fragment complexity n : $r = \{2, 5, 10\}, k = 8$.

A-buffer Comparison

Figure 7.9 illustrates performance comparison of our methods against the A-buffer alternatives for a scene with varying depth complexity. The same scene configurations with Figure 7.7's test have been used, with the difference that k is set to the fragment cardinality, so that K$^+$B methods are able to capture all generated fragments. We initially observe that both bounded K$^+$B methods perform better from all memory-aware A-buffer variants and slightly worse than AB$_{FP}$, the fastest A-buffer implementation so far. The unnecessary culling mechanism is responsible for this cost. Similar to previous tests, K$^+$B-Array outperforms K$^+$B-Heap, enhanced by its constant-time insertion process. On the other hand, K$^+$B-SB despite its smaller computational cost as compared to LL is worse than SB in all cases. Except from the culling cost, the additional condition for each fragment at the count pass significantly affects performance. Finally, note that the performance gap exhibited between K$^+$B-SB and KB-AB$_{SB}$ is alleviated when the pixel density is increased (resulting at more rasterized fragments, from $\{0.21\text{M}, \ldots, 13.12\text{M}\}$ to $\{0.61\text{M}, \ldots, 39.35\text{M}\}$).

7.2.2 Memory Allocation Analysis

Table 7.1 presents complexity in terms of memory consumption for all available methods that more or less simulate the behavior of k-buffer. We initially observe that our K$^+$B methods require slightly more storage (8-byte) per-pixel than the rest of the memory

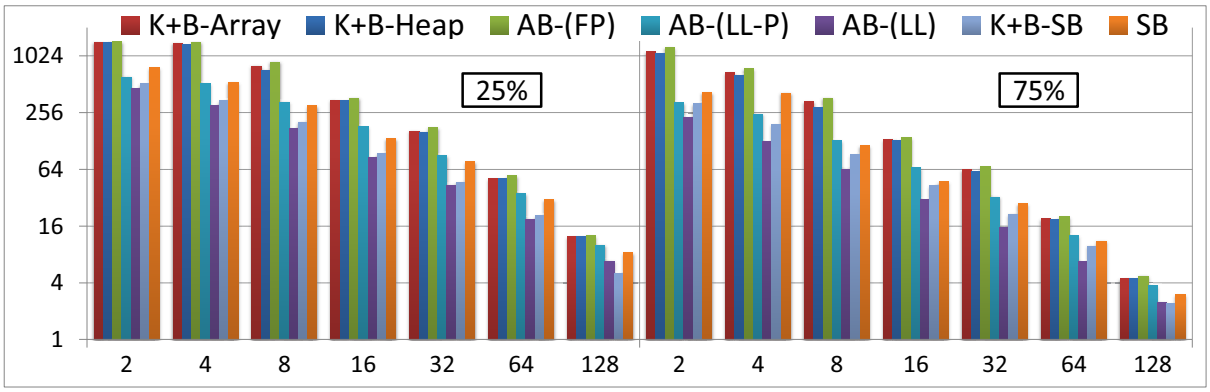


Figure 7.9: Performance evaluation in FPS ($\log_2$ scale) of A-buffer alternatives on a scene with varying maximum depth complexity.

bounded methods due to the additional allocation of the counter and semaphore textures. When moving to extreme screen resolutions this cost is noticeable. However, these methods need more storage when data packing is explored ($\forall k > 1 : 4k > 2k{+}2$). K$^+$B methods require less memory resources when compared to the KB-SR ($\forall k > 2 : 3k > 2k{+}2$). Note that semaphore texture allocation is further avoided when pixel synchronization extension is employed on Haswell hardware. On the other hand, video-memory consumption blasts off to high levels when A-buffer is constructed. Observe the increased memory requirements of KB-AB$_{FP}$ due to its strategy to allocate the maximum memory per pixel p ($n = max_p\{f(p)\} \gg k$). KB-AB$_{LL}$, KB-LL, KB-AB$_{SB}$ require less storage resources by dynamically allocating storage only for non-empty pixels ($f(p) \in [1, n]$). Our memory-aware method K$^+$B-SB requires equal (when $f(p) \leq k$) or less (when $f(p) > k$) storage than the unbounded A-buffer-based methods reducing the risk of a memory-overflow. Finally, an interesting observation is that K$^+$B and K$^+$B-SB when extended to capture all fragments ($k = n$) require the same storage demands when compared with the AB$_{FP}$ and SB methods, respectively.

7.2.3 Image Quality Analysis

Figure 7.10 shows the image differences of KB, KB-MDT and KB-AB$_{FP}$ methods when compared with the ground truth on three different scenarios: (top) Z-buffer: A *radial engine* CAD model is rendered using Gooch shading, (center) k-buffer: a transparent *hairball* model is visualized with red strips, and (bottom) A-buffer: a transparent *temple* model is completely rasterized. Noticeable quality downgrade is observed at the first two image columns due to (center, left) RMW hazards of KB and (center, right) depth conversion artifacts of KB-MDT. To avoid memory overflow of KB-AB$_{FP}$, we have to allocate less storage than we actually need leading at (right) a visually information loss for a small set of pixels. Note that in the last example, K$^+$B and KB-AB$_{FP}$ naturally produce the same image.

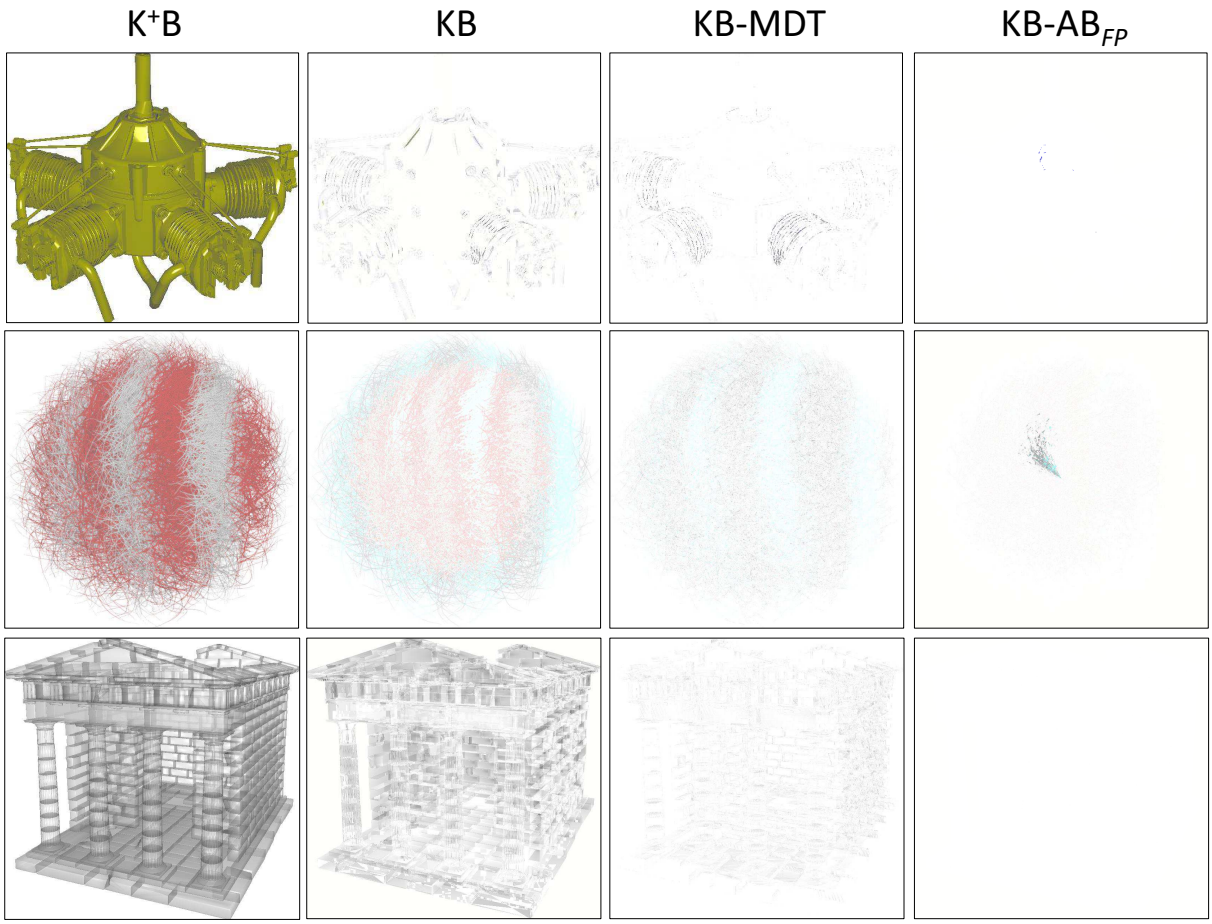


Figure 7.10: Color coded-differences between (left) the images generating using K^{+}B against the outputs of (center-left to right) KB, KB-MDT and KB-AB$_{FP}$.

7.3 Conclusions

We have introduced k^+-buffer, an improved GPU-accelerated k-buffer framework, which handles RMW memory-hazards and depth precision conversion artifacts, and avoids geometry pre-sorting and the requirement for unbounded memory. A consecutive geometry rasterization may be executed to enable precise memory allocation. Implementation details and light-weight changes are offered to enable full support of our framework on Fermi and Haswell GPU architectures. Furthermore, Z-buffer and A-buffer functionalities are successfully integrated under the proposed framework. Extensive experimental comparison demonstrated the superiority of our framework as compared to previous k-buffer alternatives with regards to storage requirements, performance and image quality.

Chapter 8

Direct Rendering of Self-Trimmed Surfaces

Most steps in the geometry processing pipeline, like deformation, smoothing, subdivision and decimation, may create self-intersections. The approximated skinned surface may also result in a self-crossing surface. Various rules have been introduced for the interior/exterior classification of the connected components of the complement of a self-crossing surface produced through a continuous deformation process of an initial embedded manifold. However current semantics are not capable of performing trimming operations automatically in a manner that is coherent over time and that is compliant with the results that would be obtained through possible constructive solid geometry operations. Sections 2.1 and 2.4.1 provide the important preliminaries regarding the problems of trimming self-intersecting surfaces and multi-fragment rendering, respectively.

8.1 Framework Overview

In this section, we introduce a complete framework [116] for treating self-trimmed surfaces as first class citizens allowing us to use them as CSG primitives or to show their cross-sections (intersections with a plane) using capping. In this work, we explore rules that capture application semantics (Section 8.1.1) for the problems formally defined at Section 1.1.5 and further provide efficient algorithms for efficiently rendering the resulting trimmed boundary on the GPU (Section 8.1.2).

More specifically, we initially explore static rules that depend only on the SCS and evaluate at least in simple cases how well the results they produce match what we consider to be plausible intentions of the designer. The second problem corresponds to dynamic rules that depend on the deformation history and the SCS. We are particularly interested in formulations of $T(S_t)$ that correspond to a designer's intuitive expectation of the sequence of results that should be produced by a reasonable deformation D_t that creates several self-crossings. In particular, we propose semantics that mimic locally the natural behavior of incremental Boolean operations, where self-crossings are created in S_t one at a time and each performs a local union or intersection of shapes defined partially by two portions of the previous frame S_{t-1}.

We also introduce practical and efficient GPU-based trimming algorithms that render $T(S_t)$ directly by scan-converting S_t and S_{t-1} without the need for computing self-intersection curves. We do this by testing the rasterized fragments (surfels), to establish whether they lie on the boundary of $I(S_t)$.

We claim three advantages of such a direct rendering and trimming approach. The first advantage is the elimination of the cost of computing self-intersection curves and of identifying the faces (connected components that are cut out by these curves). Such a cost would otherwise make it impossible to render the trim during deformation animations or perform interactive editing. The second advantage is the flexibility of being able to define S_t as the result of a (possibly adaptive) subdivision process to be carried out on the GPU. Finally, we can render the result of combining the interiors of two or more self-crossing surfaces through CSG operations.

8.1.1 Revisiting Interior Exterior Classification Rules

Previously proposed rules for interior classification are static and sometimes depend on the topology of the self-intersecting surface (for example index based classification). Such rules do not capture the process of dynamically extending interior or exterior parts in a consistent and intuitive manner (Problem II) but may yield useful classifications for determining the interior of an SCS of unknown origin (Problem I). In this section, we seek classification rules with predictable, consistent and intuitive behavior.

Static Rules

Static rules are usually based on the point index with respect to the current SCS for classifying points as interior or exterior. We wish for the classification rules to be **intrinsic**, i.e., independent of the choice of coordinate system. In some applications, it is useful to support rules based on global topological or integral properties (such as the genus or volume of a component or its surface area) [104]. We will discuss possible extensions of our work to support such **global rules**, but this section is primarily focused on local rules. A rule is **local** if the classification is based on the intersections of S with a given ray that does not intersect any curve derived by a self-crossing. In this section, we consider only static rules that are based on the index w of the component that we wish to classify.

The popular **parity** (also called *alternating interior*) rule classifies a component as *interior* when the index of its points is *odd*. This corresponds to switching the *interior* status each time one traverses S. Note that the result is not altered by a global change of the orientation of S. Hence, we say that the parity rule is **orientation invariant**. Unfortunately, the parity rule will only trim (remove from S) some of the two-dimensional self-overlapping portions of S. Thus, if S is self-crossing, but does not have self-overlaps, then $T(S) = S$. Although this is a useful rule, it does not allow the designer/user to easily modify the genus of the interior closure of a manifold boundary S by warping S so that it crosses itself. When S overlaps itself (along a full-dimensional portion instead of cleanly self-crossing), the parity of the number of portions that overlap at a point $\boldsymbol{p}$ of S defines whether $\boldsymbol{p}$ is in *T(S)*.

Unfortunately, the alternating interior rule is rarely an acceptable option since

1. Usually it does not trim the surface at all, hence it yields solids with $T(S) = S$ (except in self overlapping parts).

2. It produces non-manifold solids where each self crossing-edge is a non-manifold edge.

Both facts are illustrated in 2D in Figure 8.5(b). The red blurred lines indicate the trim (i.e. the STS which is the surface after trimming). In the simple case of Figure 8.7((left) all three manifold sub-parts are considered inside with the parity rule, even the clearly negative volume in the middle. Finally, they are connected through shared curves resulting in a non-manifold object.

Heisserman [50] has proposed the **positive index rule** *(1-st unary intersection)* that classifies as *in* the components for which w is positive (*the closure of the set of points with winding number greater or equal to* 1). Of course, w is normally 1 inside S and 0 outside when S is free from self-intersections. This rule is not orientation invariant. This semantics is appropriate for applications that involve growing an initial set through offsetting [118], sweeps and Minkowski sums [65]. For example, Figure 8.1 shows a green loop with interior G, a red loop with interior R, and a self crossing blue loop defining an interior B, which is the Minkowski average $\frac{G \oplus R}{2}$ [65], where the interior is defined as the set of points with strictly positive winding number.

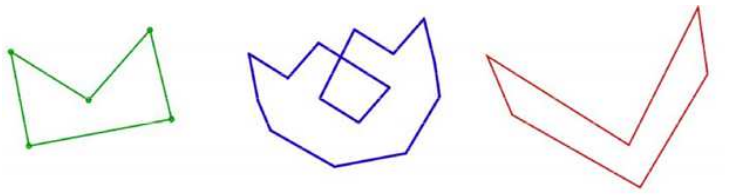


Figure 8.1: An example where the positive index rule works.

Defining the interior by $w \geq 1$ works well for cases such as the ones shown in Figures 8.5(a) and 8.6 but does not yield intuitive results for the case of Figure 8.2. Also it may improperly classify a region, such as component $\boldsymbol{e}$ in Figure 8.7. In the case of Figure 8.5(a) if we change the surface orientation we derive an empty set as $I(S)$. Finally, to derive the complement of the solid under the positive index rule, we should both change the orientation of the surface and adjust the index by adding a surrounding box (i.e. assign index $-w + 1$).

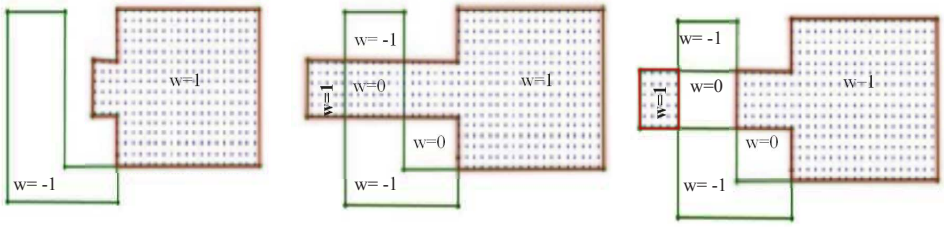


Figure 8.2: The green self-crossing loop (left) defines two regions, one of which has negative index and is discarded. Then, we grow the other region (which is the interior defined by the self-crossing loop) by extruding a portion of its left border so that it overlaps the discarded region. We expect the space conquered by this extrusion be part of the new interior (center). The red line indicates the trim $T(S)$ (the boundary of the interior). Note that (right) using the positive index rule does not produce the expected result.

Here, we propose the **alternating border rule** where a point p is *in* if and only if $\left\lceil \frac{w(p,S)}{2} \right\rceil \%2 = 1$. In simple configurations, it is equivalent to the positive index rule as shown in Figure 8.6. It has however two interesting and intuitive properties.

When the surface is free from self-overlaps, but crosses itself, then the classification of S, as being part of the trim T of $I(S)$ or not, alternates at each crossing edge. This is illustrated in 2D in Figure 8.3, which shows that this simple rule makes it easy to design and represent faces that are simply connected by using a single loop. In other words, the union of the self-crossing curves decomposes S into faces. This is substantiated by the following Theorem.

Theorem 8.1. *Adjacent faces (those incident upon the same self-crossing edge) have opposite classification with respect to the trim T (alternating border).*

Proof. Consider a surface part s crossing a surface part s' and let $f_i(s)$ and $f_{i+1}(s)$ be the two adjacent faces. We shall prove that $f_i(s)$ is part of the trim if and only if $f_{i+1}(s)$

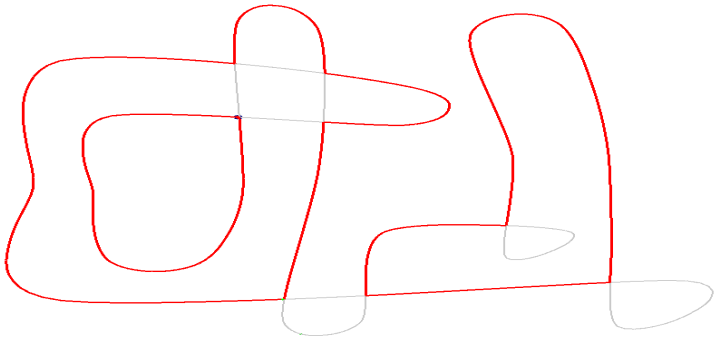


Figure 8.3: An example of creating a genus-1 object by deforming a single loop.

is not part of the trim (see Figure 8.4). For the purposes of the proof we will use an equivalent definition of the alternating border rule: a point p is *in* if and only if $w(p)\%4$ is 1 or 2. Equivalently, p is *in* if and only if there is an integer λ such that either $w(p) = 4\lambda + 1$ or $w(p) = 4\lambda + 2$. Clearly, p is *out* if and only if there is an integer λ such that $w(p) = 4\lambda$ or $w(p) = 4\lambda + 3$. Assume $f_i(s)$ is part of the trim, then the adjacent components $U_i(s)$ and $D_i(s)$ will have classification *in/out* or *out/in*, respectively. Since the components are adjacent to the face their index numbers will differ by one. Let $U_{i+1}(s)$ and $D_{i+1}(s)$ be the adjacent components to face $f_{i+1}(s)$. Without loss of generality assume $w(D_k(s)) = w(U_k(s)) + 1$ for $k \in \{i, i+1\}$. Then for every case of $U_i(s)/D_i(s)$ being *in/out* or *out/in* $U_{i+1}(s)/D_{i+1}(s)$ have classification *out/out* or *in/in* (see Figure 8.4). Therefore, $f_{i+1}(s)$ is not part of the trim. Vice versa, suppose $f_i(s)$ is not part of the trim. Likewise, it follows that $f_{i+1}(S)$ is part of the trim. ∎

Furthermore, the alternating border rule does not generate non-manifold edges unless the surface crosses itself multiple times along the same intersection curve.

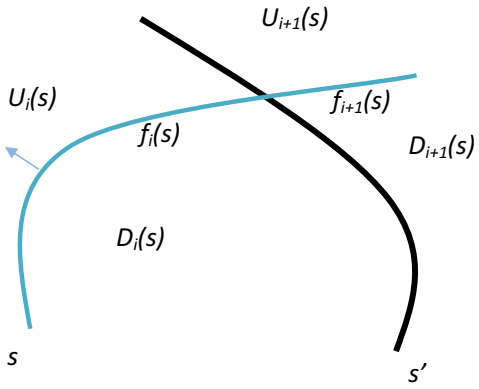
Lemma 8.1. *The alternating border rule does not produce non-manifold solids with simple self-crossings (see Figure 8.5(c))*

Proof. Consider 4 portions of the surface incident upon any segment of a self-crossing intersection curve C. Two of these are trimmed away, because our rule toggles trimmed/retained classification when crossing C. Hence, the segment has two incident portions and is thus manifold. ∎

Finally, the alternating border rule has two more practical characteristics:

1. We may obtain the complement of a solid, simply by adding two bounding boxes with the same surface orientation (index becomes $w - 2$ or $w + 2$). This follows directly from the equivalent definition of the alternating border rule in the proof of Theorem 1.

2. If we reverse the surface orientation we obtain the complement of the trim. This follows immediately from Theorem 1.



adjacent component *classification for face $f_i(s)$. Components $U_i(s)$ / $DL_i(s)$*	index numbers w($U_i(s)$) / w($D_i(s)$) w($U_i(s)$)+1=w($D_i(s)$)	Index numbers w($U_{i+1}(s)$) / w($D_{i+1}(s)$) case w increases by 1 or case w decreases by 1 after crossing s'	adjacent component *classification* for face $f_{i+1}(s)$ (after crossing s'). Components $U_{i+1}(s)$ / $DL_{i+1}(s)$
in/in	4λ+1 / 4λ+2	4λ+2/4λ+3 or 4λ/4λ+1	in/out or out/in
out/out	4(λ-1)+3 / 4λ	4λ/4λ+1 or 4(λ-1)+2/4(λ-1)+3	out/in or in/out
out/in	4λ / 4λ+1	4λ+1/4λ+2 or 4(λ-1)+3/4λ	in/in or out/out
in/out	4λ+2 / 4λ+3	4λ+3/4(λ+1) or 4λ+1/4λ+2	out/out or in/in

Figure 8.4: Establishing that under the alternating border rule, for adjacent faces it holds that exactly one of them will be part of the trim. A face is part of the trim if the two adjacent components have different interior/exterior characterization (*in/out* or *out/in*).

In particular the result of the alternating interior rule in Figure 8.5(c) is a single manifold of maximal genus (genus-3) as derived by a static SCS (Problem I).

Dynamic Rules

Figure 8.7 illustrates the limitations of static rules with respect to Problem II. We use a deformation of an initial self-crossing curve shown on the left that extends the bottom tip upwards. The intuitively correct result is shown in the right. However, both the positive index rule and the alternating boundary rule would classify area ***e*** as exterior since its index is 0.

It has become apparent from the above discussion that we need to maintain some history throughout user-controlled deformations. Besides, it is useful for several applications to render the result of a series of deformations combined with CSG and/or change of LOD operations. We need rules that capture correctly continuous deforming operations on surface parts (Problem II).

Thus, we propose more complex rules, which at each stage, compare the previous and current indices of each region. More formally, we define a characterization of the components C_i' at some stage of the deformation process based on the current index of the component, and the classification of the component and the index with respect to the surface S at the previous stage of the deformation. Components C_i' are formed by a

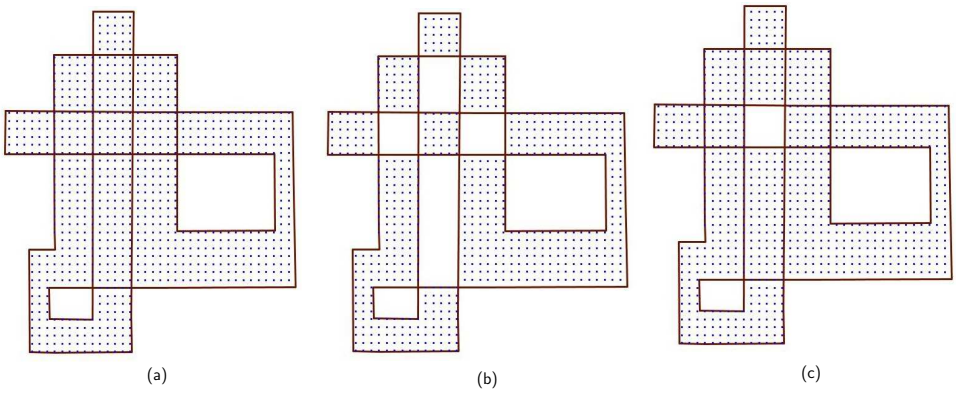


Figure 8.5: Interior classification using (a) the positive index rule, (b) the alternating interior rule and (c) the alternating border rule.

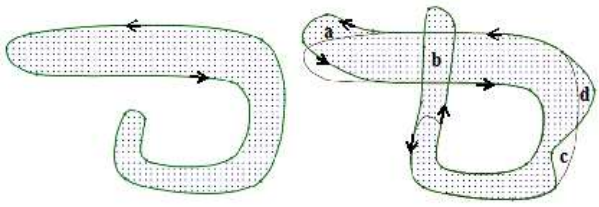


Figure 8.6: An example where the positive index number rule derives intuitive interior/exterior classification: (a) top tip wagging (b) bottom tip extending (c) dent creation (d) bump creation.

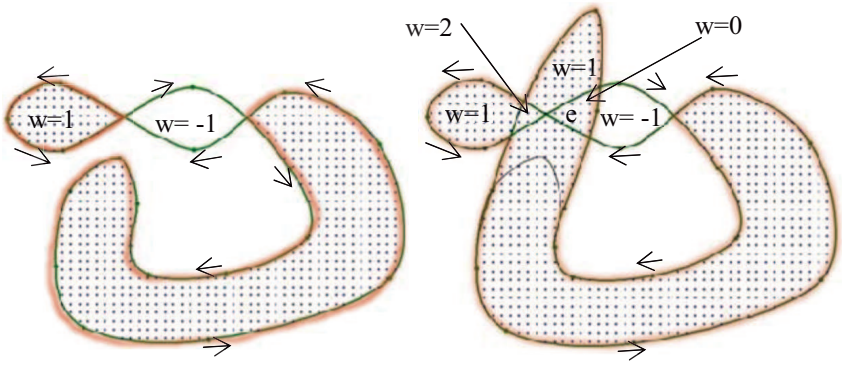


Figure 8.7: A deformation example where static rules fail to derive intuitive interior/exterior classification. The initial self-crossing curve (left) is modified (right) by extending the tip of the bottom part upwards. This change creates a region e where $w = 0$ and which is hence excluded by the static rules. Yet, intuitively, it should be part of the interior, since it corresponds to Boolean union of the initial interior with the extruded region.

deformed surface $S^{'}$ that is derived by performing k **disjoint concurrent deformations** on S, such that $S^{'} = f(S) = f(s_1) \cup f(s_2) \cup \ldots \cup f(s_k)$, where $\{s_1, s_2, \ldots, s_k\}$ is a partition of S and for all surfaces in $\{f(s_i) : f(s_i) \neq s_i\}$ it holds that they do not cross, self-cross, overlap or self-overlap. This restriction ensures that a boundary surface may cross a point only once during each set of concurrent deformations. Determining the interior/exterior is based on the classification and point index obtained for the reference surface S and the new point index with respect to the current surface $S^{'}$.

Properties Characterizing the Behavior of the Dynamic Rules. We will determine whether the following properties hold for post-deformation interior/exterior classification semantics:

Extension-normal confluence property. When deforming a surface by displacing it locally in the direction of the outward pointing normal, the points crossed either become interior or are not affected. When we deform the surface in the opposite direction to the normal, the points crossed either become exterior or are not affected. Points whose index increases are candidates to become interior points and points whose index decreases are candidates to become exterior points. Points whose index does not change preserve the status that they had before the deformation.

Complement symmetry property. If we apply the same deformations on the complement we obtain the complement of the result.

Component homogeneity property. Each component contains only interior or only exterior points. This is a very important property since otherwise the border of the interior parts may not be a subset of the deformed initial surface. This is equivalent to the aforementioned *boundary diminishing* property (see Section 2.1.1).

The following rules are based on the point index variation and the interior/exterior classification of the reference surface S. We denote the previous and the current index at a point p by $w(p, S)$ and $w(p, S^{'})$, where S and $S^{'}$ denote the surface before and after the deformation, and the previous and the current interior/exterior classification by $i(p, S)$ and $i(p, S^{'})$, respectively. Let $i(p, S)$ be the classification of point p with respect to surface S. Then, $i(p, S)$ is 1 when p is *in*, 0 when p is *out*, and *undefined* when p is on $T(S)$.

For the initial classification of the SCS surface S_0 prior to any deformation, we use the alternating border rule: $i(p, S_0) = \left\lceil \frac{w(p,S_0)}{2} \right\rceil \%2$. However, another scheme, such as the positive index rule, could be used if desired.

Constructive Rule. Here we define the constructive rule that emulates CSG behavior among the original solid and the newly created volumes due to deformation. According to CSG we employ *additive* (union), *subtractive* (difference) and *intersection* deformation semantics for interior/exterior classification. After each step of the concurrent deformations, we determine the interior/exterior classification of a point p with respect to the

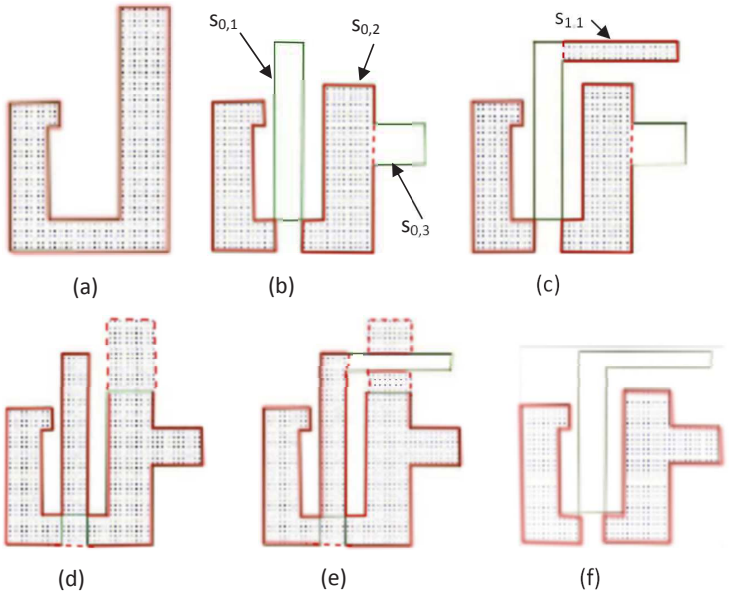


Figure 8.8: Illustrating how the constructive and the confluent deformation rules work. Interior parts are shown as shaded regions. The SCS is depicted by the green polyline. Red blurred lines indicate the trim (i.e the border of the new solid). Dashed red lines indicate parts of the trim that are not part of the SCS. (a) The original object and boundary, (b) after applying a set of 3 concurrent disjoint deformations f on surface parts $s_{0,1}$, $s_{0,2}$ and $s_{0,3}$ on (a) with subtractive semantics (constructive rule), (c) after applying a set of one deformation g of $s_{1,1}$ on (b) with additive semantics (constructive rule), (d) after applying f on (a) with additive semantics (constructive rule), (e) after applying g on (d) with subtractive semantics (constructive rule), and (f) after applying f on (a) and then g using the confluent deformation rule. In cases (a)-(e) the boundary of the shaded regions (trim) is not always a subset of the SCS.

deformed surface $S^{'}$ by performing a union, subtraction or intersection between the original solid S and the newly created volumes. A point belongs to the newly created volume if and only if its index has been modified:

$$i(p, S^{'}) = i(p, S)\, op\, (w(p, S)! = w(p, S^{'}))$$

where *op* depends on the type of deformation. For additive deformation $AopB$ corresponds to logical OR ($A \vee$B), for intersection deformation it corresponds to logical AND ($A \wedge$B) and finally the subtractive operation $AopB$ is realized as $A \wedge \neg B$.

Additive deformation corresponds to adding a part to the interior (set union), subtractive deformation corresponds to subtracting a part from the interior (set difference). These semantics yield results that are symmetric to the complement if we replace each additive with a subtractive deformation and vice versa.

We observe that although this rule captures design intent and has a constructive nature, it does not preserve component homogeneity. Thus, in some cases this rule may yield highly non intuitive results for users not familiar with the CSG process. For such users,

results where the trim is not part of the initial surface S (see Figure 8.8) may look ill-defined. To address this problem, we use the confluent deformation rule (see Figure 8.8(f)).

Confluent Deformation Rule. After each step of concurrent deformations, the interior exterior classification is determined by the following formula:

$$i(p,S') = \begin{cases} i(p,S),\ w(p,S) = w(p,S') \\ \left\lceil \frac{w(p,S')-w(p,S)}{2} \right\rceil \%2,\ otherwise \end{cases}$$

which turns out to be equivalent to:

$$i(p,S') = \begin{cases} i(p,S),\ w(p,S) = w(p,S') \\ 0,\ w(p,S) > w(p,S') \\ 1,\ w(p,S) < w(p,S') \end{cases}$$

Note that, given that if $w(p,S')$ has changed then it differs from $w(p,S)$ only by one. Thus, $i(p,S')$ is 1 if and only if the point index is increased (extending the interior) and 0 if and only if the point index is decreased (extending the exterior). Figure 8.8(f) illustrates the result of applying two sets of concurrent disjoint transformations on the object of Figure 8.8(a).

Homogeneous Confluent Deformation Rule. The confluent deformation rule can be extended so as to enforce component homogeneity by imposing the following restriction:

A part s_{out} of surface S that is not part of the border B cannot be deformed towards the normal if s_{out} is between two exterior components. Likewise a part s_{in} of a surface S that is not part of border B cannot be deformed in a direction opposite to its normal if s_{in} is between two interior components. This restriction improves the confluent deformation rule semantics by enforcing component homogeneity.

To enforce the deformation restriction we need to detect trimmed off parts of the surface, i.e. parts that do not belong to the border and the interior/exterior classification of the adjacent components. To simplify user interaction, we suggest to prohibit deformations of the surface parts that have been trimmed off.

Table 8.1 provides a comparative overview of the interior/exterior classification rules presented in this section based on their principle, the efficiency of their implementation, their properties and their intuitiveness with respect to graphics and CAD designers. Graphics designers expect continuity during deformation sequences and consistency as far as viewing from outside is concerned. On the other hand, CAD designers expect robustness, manifold objects and are better acquainted with solid modeling operations.

Implementing these rules efficiently in hardware is far from trivial. We need to detect fragments that belong to the trim of the surface based on the current index, the reference frame index and the interior/exterior classification. In all cases, we shall maintain in the current scene the fragments of the reference frame as well. By doing so, we can build efficient algorithms for realizing the confluent deformation rules with or without component homogeneity.

Rule	based on	intuitiveness for CAD designers	intuitiveness for graphics designers	Properties					Implementation efficiency
				component homogeneity	extension-normal confluence	complement symmetry	direct reversibility	order invariance	
Static rules	index	low	low	√	NA	NA	-	√	very efficient two pass
Constructive	index change and previous classification (needs reference frame fragments)	high	low	-	-	√	for additive and subtractive	-	efficient multipass or freepipe
Confluent deformation	index change and previous classification (needs reference frame fragments)	moderate	moderate	-	√	√	-	-	efficient multipass or freepipe
Homogeneous confluent deformation	index change, previous classification, and current border information (needs reference frame fragments for efficiency only)	moderate	high	√	√	√	-	-	efficient multipass or freepipe

Table 8.1: Comparative overview of the properties and characteristics of static and dynamic rules

8.1.2 The Rendering Algorithms

In this section, we present rendering algorithms for self-crossing manifolds and in particular we explain how to compute efficiently the point index, and how to perform efficiently trimming, clipping, capping and CSG operations. We discuss how the static and the dynamic rules are realized in this context. For computing and rendering the trim, we have employed multi-pass (sort-independent) and buffer-based peeling techniques. The basic process underneath all these techniques is the same: process all fragments per pixel (in addition to reference frame information for the dynamic rules) to determine the index and the interior/exterior classification. Section 8.2 presents results using all these alternatives.

Rendering using Static Rules

For the purposes of performance analysis, we consider that rendering based on static rules involves two processes:

1. compute the index p.index of a point that lies at depth p.z on a ray starting from the corresponding pixel p. For static rules, this information is sufficient for directly computing the interior/exterior classification p.class.

2. find the first fragment after the clipping depth that lies between two areas with different interior/exterior classification. This step corresponds to trimming, i.e. rendering only the trim.

We have realized this on the GPU using several algorithms. Below we describe these implementation options, their details and their advantages and restrictions.

Point Index and Interior/Exterior Classification. One may use F2B for peeling all front-facing and back-facing intersections of the ray with the SCS (see Figure 8.9). This rendering method requires a buffer for counting front-faces and a separate buffer for counting back-faces. We have employed the technique proposed at Chapter 6 that correctly resolves the z-fighting limitations achieving correct rendering of cracks, thin plates, and thin portions of the solid near sharp silhouette edges. Our objective is to iteratively peel and count the layers until we find the first layer whose depth is greater than the target depth p.depth. We use lock flags to avoid peeling at pixels that have

retrieved the next layer after the target depth. The peeling iterations terminate when we have reached our goal for all pixels. Finally, we classify the surfel to be rendered as interior or exterior by static_rule($O_f - O_b$), where O_f and O_b are the total number of extracted front and back facing fragments, respectively. Here, static_rule($\boldsymbol{w}$) stands for any static classification rule and depends only on the point index $\boldsymbol{w}$ using the static rules explained in Section 8.1.1.

Instead of depth peeling one layer per pass, we apply the dual depth peeling method for peeling a pair of a front facing and a back facing fragment in one geometry pass. The reduction of the number of geometry passes results at a significant performance speed.

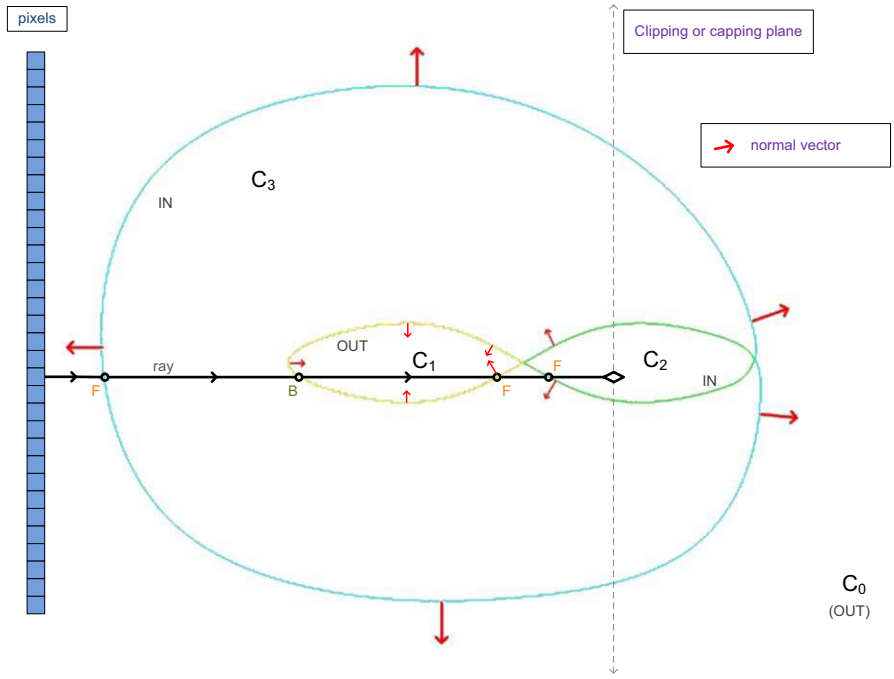


Figure 8.9: A self intersecting orientable surface that partitions space in four components: C_0 is the outside component, components C_2 and C_3 are interior and C_1 is exterior according to the alternating border rule. The boundary surfaces of C_2 and C_1 are shown with green and yellow respectively, whereas the remaining boundary (part of the boundary of C_3) is illustrated in cyan. The green part of the boundary should be trimmed off according to the alternating border rule.

Inspired by sort-independent methods [101, 8] for approximating efficiently transparency rendering effects, we introduce a technique that determines the classification in two geometry passes. The main advantage of this technique is that it does not require sorting of the individual fragment layers of the model. At the first pass, we compute the point index by turning off the hardware depth test and initializing the point index to zero (in this algorithm there is no need for lock flags). Then, for each layer whose depth is less than p.depth, we set its index count to 1 or -1 depending on whether it is front or back facing, respectively. Using the atomic ADD blending operation, we accumulate the final point index result. Then, by using a full screen pass we compute the interior/exterior clas-

sification applying the static rule on the point index. A second geometry pass is needed to retrieve the next layer after the target depth *p.depth*. Overall, the two pass technique works in two steps:

1. *IndexClassificationStatic* - (Algorithm 8.9)
2. *ClosestRender* - (Algorithm 8.10)

Algorithm 8.9 IndexClassificationStatic(Pixel p, Fragment f)

/* compute point index using ADD blending */

1: p.index := 0;
2: **for** f.z $\leq p$.depth **do**
3: p.index:= (f is *front facing*) ? 1 : -1;
4: **end for**

/* classify point as interior or exterior by applying the static rule on the point index */

1: p.class := *static_rule*(p.index);

Algorithm 8.10 ClosestRender(Pixel p, Fragment f)

/* find the closest fragment after the p.depth using Z-Test */

1: **if** f.z $\leq p$.depth **then**
2: discard;
3: **else**
4: p.color:= f.color;
5: **end if**

The same principle can be implemented by **buffer-based peeling** using the AB_{FP} or AB_{LL} techniques by processing all stored fragments with $f.z \leq p.depth$ and adjusting accordingly the signed sum of front and back facing fragments. This is accomplished by performing a depth based presorting of all fragments per pixel.

Clipping and Trimming with Static Rules. For rendering the clipped STS, we process fragments after the clipping plane until we find a fragment that has alternating interior/exterior characterization on its two adjacent sides (interior/exterior or exterior/interior). This is the fragment that we shall render. To determine the classification of the point index of the corresponding point of the clipping plane $\boldsymbol{C}$, we use Algorithm 8.9 with respect to the depth of $\boldsymbol{C}$ (i.e. setting $p.depth = C.z$).

Rendering using Dynamic Rules

The point index computation is the same as in the case of static rules. The interior classification for a certain point is a slightly more complicated process. The trimming process

for dynamic rules is considerably more complex and is outlined below.

Interior/Exterior Classification. In rendering with dynamic rules, we need to pass over to the next animation frame the characterization (interior/exterior) and the index of the corresponding clipping plane point. For this reason, we use one more texture with the index of the previous frame and the interior/exterior characterization. In fact, we may use the index and the interior exterior characterization of the point of the clipping plane that corresponds to any of the previous frames of the current sequence of disjoint deformations. For efficiency, we use the first frame of each such sequence (also called the *reference frame*). The corresponding texture information (index and interior/exterior classification) will be used in all frames of the current disjoint deformation sequence. The correctness of this process is due to the fact that during each sequence of disjoint transformations the index of each point may be altered only once. For this process, we use a variation of the *static* methods. We present a dynamic classification rendering technique that determines the point as interior or exterior by applying the dynamic rule on the point index. Note that p^c.index refers to the index of point in a pixel p in the current frame. Likewise, the p^r.class and p^r.index refer to the interior classification and the index of p in the reference frame and are modified only when we enter a new concurrent disjoint deformation sequence. Note that the algorithm can only work if both reference and current frames are rendered from the same viewport. The details are shown in Algorithm 8.11.

Algorithm 8.11 IndexClassificationDynamic(Pixel p)

/* classify point as interior/exterior by applying the dynamic rule on the point index */

1: p^c.class:= *dynamic_rule*(p^c.index,p^r.index,p^r.class);
2: **if** *current frame == reference frame* **then**
3: p^r.class := p^c.class;
4: p^r.index := p^c.index;
5: **end if**

Clipping and Trimming with Dynamic Rules. For rendering the STS with dynamic rules, we need to have available all previous interior/exterior characterizations and index information from the reference frame. To do this, we need all interior/exterior information, the location of all fragments (i.e. the corresponding depths), and the information to compute the corresponding indices. It is prohibitive to maintain all this information per pixel and pass it over from one shader to the next. A test implementation demonstrated that this is feasible using all available texture memory for 128 layers of fragments but this would also speed down considerably the rendering algorithm and would require powerful state of the art graphics hardware.

We present an algorithm that maintains the geometry information of the reference frame and uses coding for distinguishing whether a fragment has been derived from a primitive of: (i) the reference frame, (ii) the current frame or (iii) both (has not been deformed). This information is compiled through the geometry shaders using the value

of the *frame.class* parameter to store the frame classification of the primitive. This is then conveyed to the corresponding fragments. If $frame$.class= 0 then this fragment has originated from a primitive that is part of the reference frame only. If $frame$.class= 1 then the fragment has originated from a primitive that is part of the current frame only. Finally, $frame$.class= 2 means that the fragment has originated from a primitive that exists in both the current and the reference frames.

This is decided based on whether a primitive has moved as compared to its initial (reference frame) position. Thus, for each pixel we have available all fragment information from the current frame and the reference frame including depth information. We can also compute the corresponding indices for each such fragment of the current or the reference frame. In addition to this information we need a bit vector that will store the in/out information per pixel that corresponds to the characterization of the partitioning of space by the corresponding fragments. This needs to be computed only once for each reference frame. The size used to store this information sets a bound for the number of layers that we can peel. If we use a 4×32 bit vector we can account for 128 fragment layers per frame, a trade off that is quit reasonable even for commodity graphics hardware.

The algorithm at a high level uses the following registers per pixel that are passed on to the next step: the current depth of the fragment we are processing p^c.depth (initialized to the depth of the clipping plane $\boldsymbol{C}$); the final color of the first fragment that will not be trimmed: p.color (initialized to 0, this variable is also used as a lock flag); the p^c.index that is the index initially at the clipping plane and then after each processed fragment at the current frame; the p^r.index that is the index in the context of the reference frame initially at the clipping plane and then after each processed fragment; the bit vector PC^r (reference frame partitioning characterization vector) that stores the interior/exterior characterization of the areas between fragments of the reference frame (computed for each reference frame only); the p^r.count is the fragment index for the fragment classification bit vector of the reference frame and the local registers CL_b, CL_a that maintain the characterization of the points before and after the fragments in the current frame. The algorithm is carried out in two steps:

1. *IndexClassificationDynamic* - (Algorithm 8.12)

2. *TrimRenderDynamic* - (Algorithm 8.13)

IndexClassificationDynamic() is invoked only once and determines the indices of the clipping plane at the reference and the current frames.

During *TrimRenderDynamic*, we process one layer at a time, until we find the first fragment that should be rendered. Each step of the *TrimRenderDynamic* algorithm corresponds to either a separate shader invocation (multi-pass peeling) or processing the next fragment in the sorted fragment list.

Algorithm 8.12 IndexClassificationDynamic (Pixel p, Fragment f)

/* compute indices using ADD blending on $[cIndexP, rIndexP]$ */

1: $[p^c.\text{index}, p^r.\text{index}] := [0, 0]$;
2: **if** $f.\text{z} \leq p.\text{depth}$ **then**
3: $\quad w_f := (f \text{ is } \textit{front facing}) \ ? \ 1 \ : \ -1$;
4: $\quad p^c.\text{index} := (frame.\text{class} > \ 0) \ ? \ w_f \ : \ 0$;
5: $\quad p^r.\text{index} := (frame.\text{class} \neq \ 1) \ ? \ w_f \ : \ 0$;
6: **end if**

/* classify point as interior/exterior by applying the dynamic rule on the point index */

1: $p^c.\text{class} := \textit{dynamic\_rule}(p^c.\text{index}, p^r.\text{index}, p^r.\text{class})$;
2: **if** *current frame == reference frame* **then**
3: $\quad p^r.\text{class} := p^c.\text{class}$;
4: $\quad p^r.\text{index} := p^c.\text{index}$;
5: **end if**

Algorithm 8.13 TrimRenderDynamic(Pixel p, Fragment f)

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

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

Capping and CSG Operations

We have implemented **capping** at no extra cost by subtracting a capping box from our object (see CSG operations below) or by simulating the result of a front facing plane that has extended the exterior towards the capping plane (see Section 8.2). This will clip and cap the target object. We can render this using the algorithms described in the previous sections by setting the clipping plane outside the object since no additional clipping needs to be performed in this case.

Constructive solid geometry can be supported, allowing a modeler to create a complex surface by using Boolean operators even between complicated self-crossing objects. If BC_A^r and BC_B^r are the reference partitioning characterization bit vectors of manifolds A and B respectively, then *union*, *intersection* and *difference* can be implemented by simply performing bitwise operations on these vectors to compute the current point characterization and fragment trimming of the resulting CSG object. In this case, index and dynamic rule computations for each fragment of the current or the reference frame are not needed, thus speeding up the trimming of the CSG operation result.

We may consider as deformed from the reference frame one of the two combined surfaces, say without loss of generality surface B. We maintain the geometry information of the reference frame and we use coding to distinguish the primitives of A and B. As before, $frame$.class= 2 means that the fragment has originated from a primitive that is common in both the current and the reference frames, so that it belongs to surface A. Moreover, $frame$.class= 1 means the fragment has originated from a primitive that is part of the current frame only, so it belongs to surface B. Finally, if $frame$.class= 0 then this fragment has originated from a primitive that is part of the reference frame only, a case that never occurs in our CSG setting.

The algorithm at a high level uses the following registers per pixel of the dynamic trimming and the following registers that are passed on to the next step: the $Count_A^r$ and $Count_B^r$ that are the fragment indices for the fragment classification bit vector of the reference frames of A and B, the $Count_{AopB}^c$, that is the fragment index for the fragment classification bit vector of the current frame of the result of the CSG operation, the bit vectors PC_A^r and PC_B^r that store the interior/exterior characterization of the space partitioning by the fragments of A and B at their reference frames, the bit vector BC_{AopB}^r that stores the interior/exterior characterization of the space partitioning by the fragments of the resulting surface at the current frame (this becomes then the new reference frame), and the local registers: p.color of the latest processed fragment and finally the characterization of the points before and after the fragments in the current frame CL_b, CL_a.

We use the bitwise operation *LOGIC_OP*. Union, intersection and difference are computed using the *OR*, *AND* and *AND_REVERSE* modes of *LOGIC_OP*, respectively. The $A - B$ is realized using the *AND_REVERSE* operation, i.e. using the equivalent $A \cap \neg B$. The algorithm is performed in two steps:

1. *IndexClassificationDynamic* - (Algorithm 8.12)

2. *RenderCSG-* (Algorithm 8.14)

During *RenderCSG()*, we process one layer at a time, until we find the first fragment that should be rendered.

Algorithm 8.14 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**

Figure 8.10 illustrates an example of CSG operations on two self-crossing manifolds A and B. The top (reference) frame shows the A and B manifolds prior to the CSG operation. The current frame is generated transforming B. The processed fragments from an arbitrary ray (yellow line) of the newly created self-crossing manifold surface are highlighted with small circles. Using the bit vectors (middle) with the interior/exterior classification of the areas between the fragments in the reference frame we provide the characterization for both surfaces of A and B surfaces after combining them. The computed bit vectors for union, intersection and difference operations are also shown. Finally, the resulting border (highlighted in orange) is computed using the trim that consist of boundary with alternating interior/exterior characterization on each side. Filled highlighted areas illustrate the interior of the resulting manifold.

8.2 Experimental Study

To demonstrate our technique, we have applied the rendering algorithms to a user controllable animation setting using both static and dynamic rules. We have used two types of concurrent local deformation operations: local influence deformations in conjunction with Laplacian smoothing and control point movement of NURB surfaces combined with

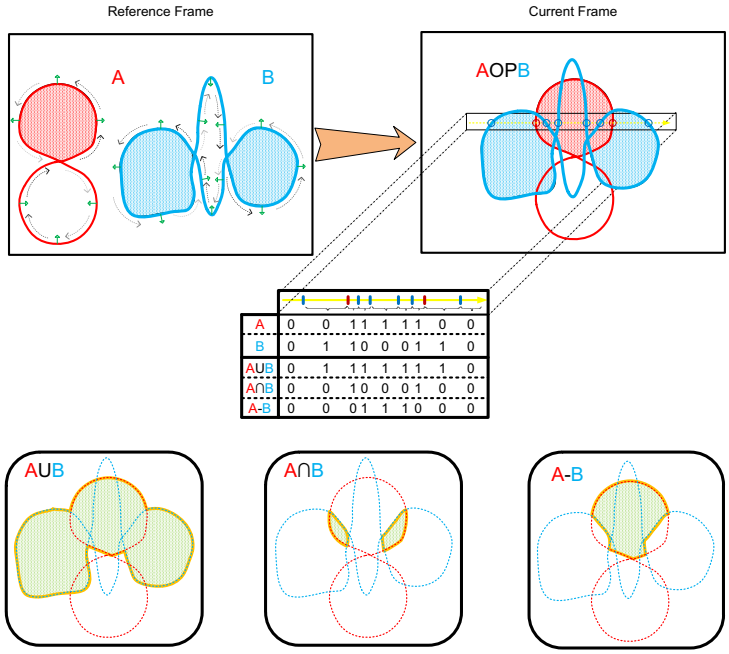


Figure 8.10: Trimming for CSG operations.

mesh subdivision. The animation starts with an orientable self-crossing manifold whose areas are classified and trimmed using any static rule. At each step the user applies a sequence of concurrent disjoint deformations. The user may select different rules for each set of concurrent deformations. We have implemented a GPU-based linear space warp deformation tool to offer users the capability of creating arbitrary animation sequences. This tool works the same way a magnet does; it attracts many vertices at each frame forcing the surface to deform smoothly. Vertices other than the selected vertices are affected within a geodesic distance range. An angle-weighted estimation process computes the attenuated deformation at each vertex. When working with non-dense geometry, it can become difficult to apply extreme stretches to the vertices without causing nasty lumps and creases on the model surface. To correct this effect, we have implemented iterative Laplacian smoothing and weighted vertex normal recomputation as separate GPU steps performed after the deformation process. Alternatively, for Bspline or NURB-based meshes the user may edit the control polygon and adjust the subdivision to derive a set of concurrent deformations.

Resolution 1024x768				*Static Rules*																										
Model				*F2B*							*Dual*							*2 passes*			*FreePipe*			*Linked Lists*			*Dynamic*			*Standard*
Name	**Size**			**Best**		**Average**		**Worst**			**Best**		**Average**		**Worst**			**All**			**All**			**All**			**All**			**All**	
	Vertices	Faces	Fragments	Fps	Passes	Fps	Passes	Fps	Passes	MB	Fps	Passes	Fps	Passes	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes	MB	Fps	Passes
Homer	5103	10202	0.991M	510		212	7	121	16		490		203	5	113	9		940			575		90	111		20,34	895			1355	
Sphere	7478	14952	0.897M	500		263	6	180	10		465		240	4	162	6		1055			620		54	123		19,26	990			1518	
Deformed sphere	7478	14952	0.933M	515	2	219	7	135	14	21,75	485	1	194	5	125	8	31,5	965	2	7,5	608	1	78	115	1	19,67	905	2	9,75	1465	1
Deformed Nurbs	23807	47557	2.57M	330		149	7	88	14		300		120	5	76	8		520			312		78	47		38,4	494			824	
Armadillo	172974	345944	0.8M	213		51	9	31	16		245		73	6	51	9		273			350		90	30		18,2	265			461	
Dragon	437645	871414	1.3M	121		23	12	12	22		162		38	7	22	12		198			258		126	38		23,83	196			286	

Table 8.2: FPS, number of passes and memory needed using the static rule without trimming.

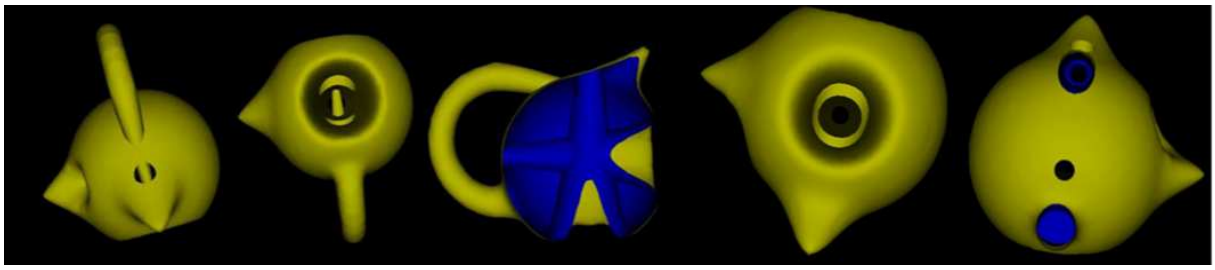


Figure 8.11: Rendering of the trim using clipping after applying several sets of deformations.

We have implemented interior/exterior point classification and rendering using the algorithms presented in Section 8.1.2. The implementation is based on OpenGL 4.2 using framebuffer objects with high precision 8bit integer, 32bit floating internal texture formats and 32bit floating point depth buffer precision. All experiments were carried out on a commodity desktop with Intel Core i7-870 2.93GHz, 4GB DDR3 memory and NVIDIA GeForce GTX 480 graphics hardware. The visual results demonstrated by the figures use resolution of 1024×768 pixels which yields reasonably high quality results. We have implemented classification, trimming and rendering in conjunction with clipping and capping.

Figure 8.11 illustrates the result of rendering a self-trimmed surface that has undergone a series of deformations. In this case, the result is the same using either the static alternating border rule or the dynamic confluent deformation rule. Yellow visualizes front facing geometry whereas blue visualizes back facing geometry. The snapshot in the center uses clipping for inspecting the interior of this complex self-crossing manifold. Figure 8.13 illustrates the result of applying several deformations on the NURB surface (a) by moving the control points. The unintuitive hole and bump that are introduced when rendering the trim using the static rule (d,f), are eliminated through the use of the dynamic rule (e,g). The unintuitive hole is created due to the fact that the extended positive surface meets a previously negative area that is not part of the trim before the deformation. The unintuitive bump is created due to the fact that the extruded hole meets a previously internal highly positive index area that is not part of the trim before the deformation. For a better visual understanding of the unintuitive holes the reader is referred to the supplementary material. Figure 8.14 illustrates the result of using capping (c) for inspecting the interior of model (a). Here we use a striped texture for rendering the capped part.

Figure 8.15 illustrates CSG operations between two copies of a highly deformed sphere. Finally, Figure 8.12 demonstrates the combination of deformed horse (where the original surface has been pushed through the object to create the SPM logo) with several other objects to render a horse-armadillo with dinosaur tail. The resulting object has around 500k faces producing 2 million fragments and is being rendered at 20-100 fps depending on the method used.

Table 8.2 presents a comparative performance evaluation of the proposed algorithms

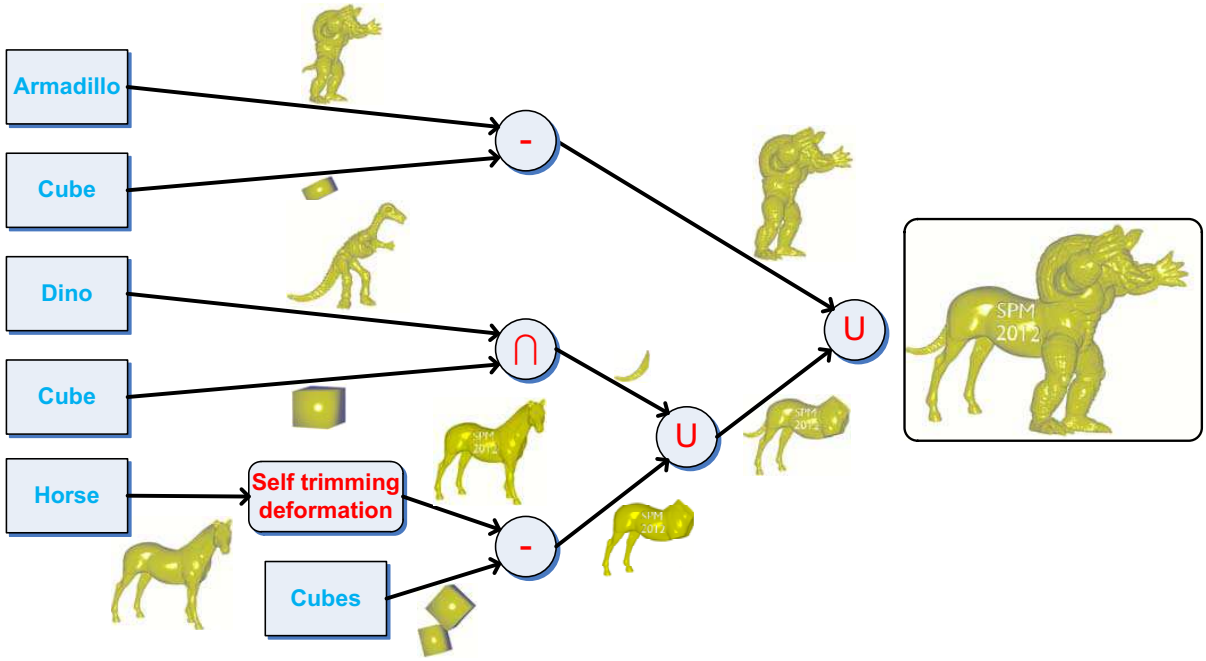


Figure 8.12: Rendering a deformed horse model combined with several other complex object parts with CSG operations.

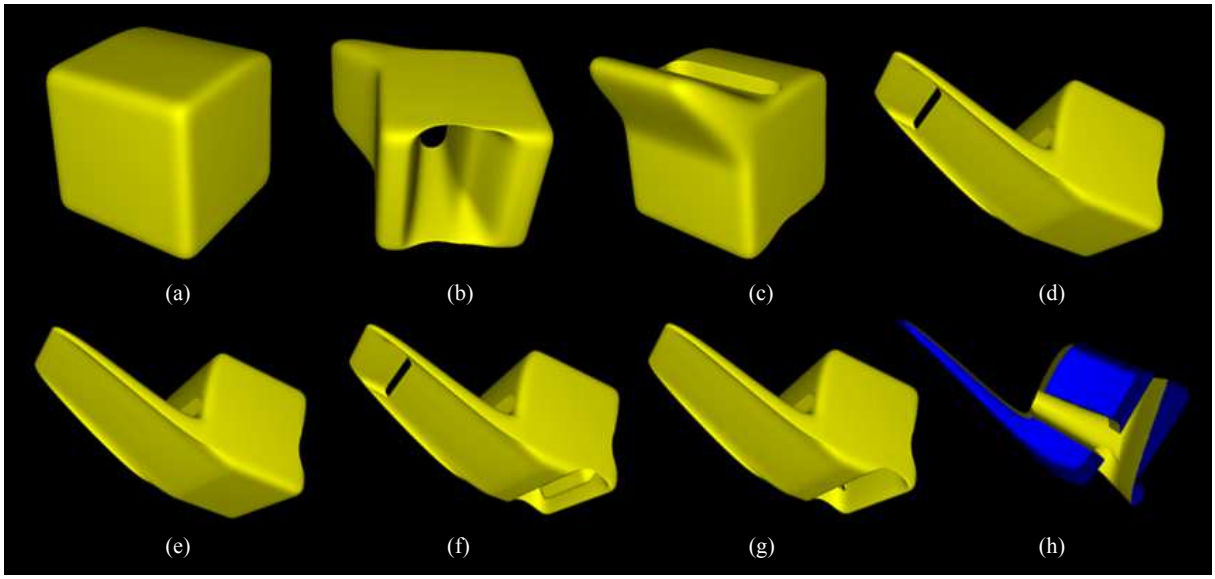


Figure 8.13: (a) The original NURB surface. (b and c) After applying a set of disjoint deformations. (d) Rendering the result of one more deformation with the static rule, where the upper extrusion is deformed so as to cross an extended hole (observe the unintuitive hole in the upper part) and (e) the same using the dynamic rule (no hole is present in the upper part). (f) Rendering the trim with the static rule after one more hole is created at the bottom, observe the unintuitive bump at the bottom and (g) the same with the dynamic rule (no bump is present). (h) g with clipping.

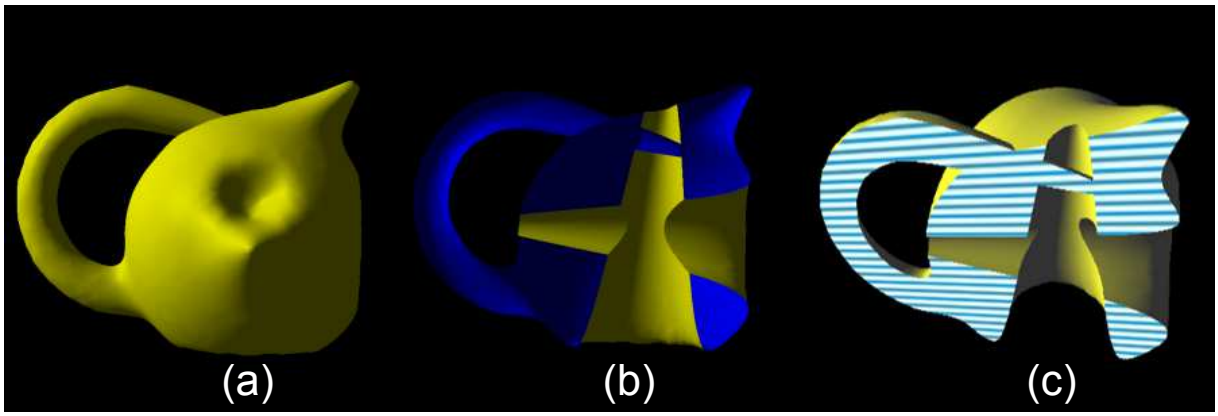


Figure 8.14: Rendering the deformed self-trimmed surface of (a) using (b) clipping and (c) capping.

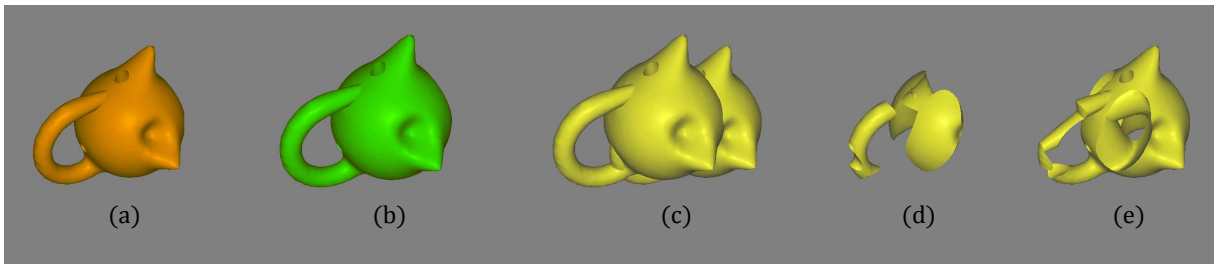


Figure 8.15: (a) Object A. (b) Object B. (c) Rendering $A \cup B$, (d) $A \cap B$ and (e) $B - A$.

for in/out classification and index computation without trimming for models with different characteristics. For all peeling methods, we provide the resulting frames per second and the number of passes needed for point classification. Two geometry pass performs very well as compared to all other methods including the AB_{FP} and the AB_{LL}. For the F2B and DAUL peeling methods, we have provided results for minimum, average and maximum number of rendering passes which depends on the location of the clipping/capping plane. The performance of the rest of the techniques is not affected by the number of depth layers of the model. This is verified by observing that the rendering frame rates are almost identical for models with the same size and different number of depth layers such as the sphere and a deformed self-crossing instance of it. Note that carrying out depth peeling with either the F2B or the Dual technique on models with high level of detail such as the Armadillo and the Dragon results in a significant performance downgrade. We observe that the linked lists approach that creates linked lists of all fragments per pixel yields poor results in terms of efficiency because of extensive memory access contention. Thus, the AB_{LL} implementation depends a lot on the number of the generated fragments. The AB_{FP} based technique achieves reasonably good performance in the expense of increased memory requirements. Moreover, AB_{FP} scales better that Linked-Lists with respect to resolution increase.

Table 8.3 presents performance results for rendering the trim using our technique with (a) multi-pass depth peeling (b) AB_{FP} peeling and (c) peeling using AB_{LL} on a

NURB surface with three tessellation levels, a deformed self-crossing sphere and several non self-crossing models.

In the *Homer, Armadillo* and *Dragon* models, static depth peeling stops after extracting the first layer (using two geometry passes) since these are non self-crossing. On the other hand, the multi-pass algorithm on self-trimmed models depends on their maximum depth complexity since peeling is needed until all pixels find a fragment with alternating in/out classification on its two sides. For the NURBS and the deformed sphere three to five rendering passes are required. For the dynamic case, we choose the worst case scenario using as current and reference model the same model, leading at poor performance due to the large number of layers processed. The AB_{FP} technique outperforms static multi-pass peeling since only one geometry pass is needed to store the entire fragment array. Furthermore, the overhead for the dynamic trimming is considerably smaller (around 50%) as compared to the overhead for the multi-pass peeling ($> 60\%$). Finally, the limitations of the AB_{LL} algorithm discussed in Table 8.2, hold for the trimming version as well.

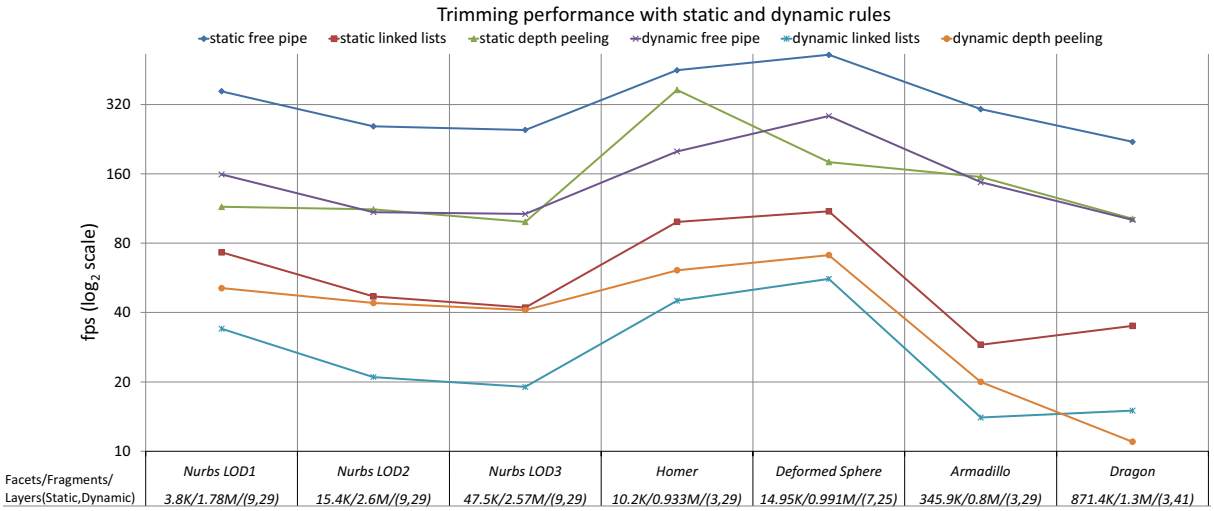


Table 8.3: Performance results for rendering a self-trimmed surface using static and dynamic rules.

8.3 Conclusions

We have explored static and dynamic rules for classifying the interior of self-crossing manifolds and have introduced algorithms for efficiently rendering the resulting trimmed boundary on the GPU. Several well fitted applications, such as previewing the interior of solids using capping or used as a primitive in direct rendering of CSG operations, are further implemented. We have adapted several state of the art multi-fragment techniques to achieve efficient rendering of self-trimmed surfaces and we have provided comparative results in terms of time and memory requirements.

Chapter 9

Conclusions and Future Work

9.1 Summary

The contributions of this dissertation are twofold. First, the motivation of this thesis has been to design efficient methodologies for editable segmentation and skinning representations of arbitrary animated mesh collections that take into account the temporal coherence from a pose-to-pose point of view. Second, we have focused on the problem of accurate and interactive rendering (detecting and trimming) of self-crossing dynamic objects realized through novel multi-fragment rasterization solutions.

More specifically, in Chapter 2 we have offered the required mathematical notation and background regarding the aforementioned problems. We have further provided a detailed overview of the graphics rendering pipeline. Finally, a comprehensive survey of the related prior art is included.

In Chapter 3, we have presented a general approach to efficiently deriving a multi-resolution segmentation of arbitrary deformations. An over-segmentation is initially built by merging the individual partitionings computed for each animation pose based on a deformation feature measured from a reference pose. The desired segmentation resolution is dynamically chosen by the user applying a robust temporal-coherent reduction process which aims at cleaning noisy segments created between successive partitionings. Contrary to prior segmentation methods, our final result can accurately be adjusted when the original mesh sequence is either modified or updated. A perceptually friendly visualization scheme have also been introduced for propagating segment colors between consecutive frames. Finally, we have included extensive comparative results showing the superiority

of our segmentations in the context of skinning quality when compared with a variety of state-of-art methods.

In Chapter 4, we have presented a novel pose-to-pose approach to skinning highly deformed animated meshes by observing that only small deformation variations will normally occur between sequential poses. The idea is to perform the skinning transformations as a new pose is derived by transforming the vertices of the previous pose. Although transformation fitting is performed from pose to pose, a reproduction scheme from the rest pose to an arbitrary pose may efficiently be computed. Several optimization solutions are further given to significantly refining the skinning approximation. The experimental study has shown that this scheme reduced the approximation skinning error and further supported arbitrary propagated pose editing without increasing the storage or the computation requirements.

In Chapter 6, we have considered a key problem in multi-fragment rendering: the Z-fighting, a phenomenon that occurs frequently, unexpectedly and causes various unpleasant and unintuitive results when rendering scenes suffering from coplanar geometry. Approximate and exact extensions to conventional single and multi-pass rendering methods have been introduced accounting for coincident fragments. A large spectrum of rendering effects have been considered and used for illustrating the detected differences. Finally, we expect that the suite of features and trade-offs offered for each technique, pointing out GPU optimizations, portability, and limitations, will provide a useful guide for effectively addressing coplanarity artifacts.

In Chapters 5 and 7, we have investigated efficient GPU-accelerated multi-fragment rendering solutions that simulate the behavior of A-buffer and k-buffer architectures with reduced memory demands, respectively. First of all, we have presented S-buffer, an efficient and memory-aware A-buffer implementation based on real-time concurrent construction of per-pixel variable-length fragment bins on the GPU. S-buffer is designed so as to take advantage of the fragment distribution and the sparsity of the pixel-space using two geometry passes. Experimental analysis have demonstrated that S-buffer exhibits improved combined memory usage and performance behavior even in low pixel sparsity rasterizations. Finally, we have introduced k^+-buffer to accurately maintain the k-foremost fragments per pixel in a single geometry pass avoiding memory-overflow. k^+-buffer alleviates prior k-buffer limitations and bottlenecks by exploiting fragment culling and pixel synchronization. A memory-friendly strategy has also been proposed, extending the proposed pipeline to dynamically lessen the potential wasteful memory allocation. Without any software-redesign, the proposed scheme can be adapted to perform as a Z-buffer or an A-buffer capturing a single or all generated fragments, respectively. Extensive experimental comparison demonstrated the superiority of our framework as compared to previous k-buffer alternatives with respect to storage requirements, performance and image quality.

Finally, in Chapter 8, we have studied the problems of real-time (interior) visualization of self-intersecting deformable solids. The motivation behind this work is to perform

the trimming operation in the image space, eliminating the cost of computing the self-intersection curves and identifying the new faces. The advantage is that self-trimming of animated or interactive edited models can now be supported. To this end, static and dynamic rules for the interior/exterior classification of the object's parts in 3D have been introduced, automatically deciding which region should be visible at each pixel. Based on these semantics, we have introduced practical and efficient GPU-based trimming algorithms by adjusting the classic single and multi-pass rendering frameworks. Being able to render the trim in real time makes it possible to interactively adapt the tessellation of the trim by using view-dependent levels-of-detail or adaptive subdivision. Except of the context of self-trimming, another well-fitted application has been offered previewing the interior of solids using clipping and capping operations prior to performing free form editing. Finally, the fast interior and boundary detection and rendering have been extended to combining the interiors of two or more self-crossing surfaces through CSG operations.

9.2 Limitations and directions for future work

There are a number of research directions that could be explored further to improve our techniques regarding segmentation and skinning of mesh animations. First, our segmentation may easily support time-varying meshes [4] by exploiting vertex mapping techniques to establish pairwise parametrization between successive frames. Further directions may be investigated for tackling the problem of the increased resolution of over-segmentation. For example, we may reorganize similar poses into clusters and pick one of them thereby reducing significantly the number of the merged partitionings. Except from improving performance, this solution will also reduce the additional memory requirements of storing the individual per-pose partitionings. It would be interesting to investigate the possibility of employing a more sophisticated mesh clustering algorithm [130], rectifying the generated boundaries [63], or imposing a confidence criterion on the initial pose partitionings to retain only important boundaries in order to increase the final segmentation quality. However, this will come with the cost of additional parameter tuning and extra computation cost. Finally, visualization coherency is lost between "far-away" poses when there is no mapping between segments of successive poses. An interesting alternative strategy could be to increase the cluster mapping search in a larger time window. Another challenge is to combat the high pre-processing overheads by making use of modern high-performance parallel architectures architectures such as GPUs and multicore CPUs using CUDA or OpenCL [46, 16, 76].

While switching to the pose-to-pose scheme reduces the approximation error and can support editing tools with much more capabilities as compared to the rest-pose scheme, there are some limitations that should be addressed. The proposed scheme cannot be implemented in parallel and approximation flaws that occur in a single pose may be propagated to all subsequent poses due to the sequential nature of the fitting process. Similarly to segmentation, key-frames extraction methods of animation sequences can also be used

to reduce the dimensions of the fitting optimization. Mesh simplification algorithms [163] can also be used for the same reason. Finally, we believe that temporal coherence for deforming objects deserves further investigation. For example, future research should be conducted on adaptive proxy-joint distribution methods and automatically discovering the smallest reasonable number of suitable clusters.

Considering completeness, further directions may be explored for combating with multi-fragment rendering issues in rasterization architectures. Similarly to all previous unbounded A-buffer implementations, GPU memory recourses may be exceeded and per-pixel sorting may stall rendering when the number of generated fragments becomes too high. To reduce bandwidth demand of the rendering operations and increase locality of memory accesses, tiled rendering [144] may be explored.

Despite the low memory requirements of the proposed depth-fighting-free multi-pass techniques, the rendering passes can be a bottleneck for scenes with high depth complexity. Determining the set of elements that are not visible from a particular viewpoint, due to being occluded by elements in front of them [10], may affect the performance of the multi-pass peeling methods. Moreover, a hybrid technique [18] is an interesting option that should further be investigated. To this end, one may seek a modified form of peeling which efficiently captures a sequence of layers when coplanarity is not presented followed by on demand peeling of overlapping fragments.

Regarding the S-buffer method, a direction for future work is to develop a more sophisticated hash function that uniformly divide the non-empty pixels. So far, performance complexity is bounded by the pixel group with the highest fragment capacity. Uniformly distributing workload across individual pixels groups may result at a significant performance gain.

The need of an additional accumulation step of S-buffer and k^+-buffer may result at a performance downgrade when compared to the fixed-size array buffers. A neat idea is to perform the preliminary fragment counting pass rendering a lower-detailed subdivision of the initial 3D scene by employing tessellation shaders. On the other hand, temporal coherence solutions [127] may be exploited to improve performance based on the observation that successive frames typically have the same or very similar number of fragments located at the same screen pixel location.

A dynamic k-buffer technique, where k value is not the same for all pixels, is an interesting problem that should be examined. In cases, where the number of fragments varies when the camera or scene is animated, an interesting alternative is to capture the $k\%$ of the generated fragments per-pixel. The memory-friendly variation of k^+-buffer could easily be adjusted to support this function by taking advantage of its first rendering pass.

Concerning rendering of self-intersecting solids, it is possible to further extend the proposed framework in order to realize fast collision detection in complex scenes. It is worth noting that it is of great value to investigate ways to voxelize the interior and then produce an actual mesh for the self-trimmed surface [170].

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2011.

[2] Rachida Amjoun and Wolfgang Straber. Efficient compression of 3d dynamic mesh sequences. *Journal of WSCG*, 15(1-3), 2007.

[3] Dragomir Anguelov, Praveen Srinivasan, Daphne Koller, Sebastian Thrun, Jim Rodgers, and James Davis. Scape: Shape completion and animation of people. *ACM Trans. Graph.*, 24(3):408–416, July 2005.

[4] Romain Arcila, Cédric Cagniart, Franck Hétroy, Edmond Boyer, and Florent Dupont. Segmentation of temporal mesh sequences into rigidly moving components. *Graphical Models*, 75(1):10–22, 2013.

[5] J. Andreas Baerentzen, Steen Lund Nielsen, Mikkel Gjøl, and Bent D. Larsen. Two methods for antialiased wireframe drawing with hidden line removal. In *Proceedings of the 24th Spring Conference on Computer Graphics*, SCCG '08, pages 171–177, New York, NY, USA, 2010. ACM.

[6] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 97–104, New York, NY, USA, 2007. ACM.

[7] Louis Bavoil and Kevin Myers. Deferred rendering using a stencil routed k-buffer. *ShaderX6: Advanced Rendering Techniques*, pages 189–198, 2008.

[8] Louis Bavoil and Kevin Myers. Order independent transparency with dual depth peeling. *Technical Report, Nvidia Corporation*, 2008.

[9] Curtis Beeson and Kevin Bjorke. Skin in the "dawn" demo. *SIGGRAPH Comput. Graph.*, 38(2):14–19, May 2004.

[10] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[11] Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics*, 14:213–230, January 2008.

[12] Abdullah Bulbul, Cetin Koca, Tolga Capin, and Uğur Güdükbay. Saliency for animated meshes with material properties. In *Proceedings of the 7th Symposium on Applied Perception in Graphics and Visualization*, APGV '10, pages 81–88, New York, NY, USA, 2010. ACM.

[13] Stef Busking, Charl P. Botha, Luca Ferrarini, Julien Milles, and Frits H. Post. Image-based rendering of intersecting surfaces for dynamic comparative visualization. *Vis. Comput.*, 27:347–363, May 2011.

[14] Steven P. Callahan, Milan Ikits, João L. D. Comba, and Cláudio T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[15] Benot Le Callennec and Ronan Boulic. Interactive motion deformation with prioritized constraints. *Graphical Models*, 68(2):175 – 193, 2006. Special Issue on SCA 2004.

[16] Feng Cao, Anthony K. H. Tung, and Aoying Zhou. Scalable clustering using graphics processors. In *Proceedings of the 7th international conference on Advances in Web-Age Information Management*, WAIM '06, pages 372–384. Springer-Verlag, 2006.

[17] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, volume 18, pages 103–108. ACM New York, NY, USA, 1984.

[18] Nathan Carr, Radomír Měch, and Gavin Miller. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 33–40, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[19] Thomas J. Cashman and Kai Hormann. A continuous, editable representation for deforming mesh sequences with separate signals for time, pose and shape. *Computer Graphics Forum*, 31(2pt4):735–744, 2012.

[20] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[21] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 554–560. ACM, 2006.

[22] E. Chaudhry, L. H. You, and Jian J. Zhang. Character Skin Deformation: A Survey. *2010 Seventh International Conference on Computer Graphics, Imaging and Visualization*, pages 41–48, August 2010.

[23] Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. A benchmark for 3d mesh segmentation. *ACM Transactions on Graphics (TOG)*, 28(3):73, 2009.

[24] Daniel Cohen-Or. Space deformations, surface deformations and the opportunities in-between. *J. Comput. Sci. Technol.*, 24:2–5, January 2009.

[25] Forrester Cole and Adam Finkelstein. Partial visibility for stylized lines. In *Proceedings of the 6th international symposium on Non-photorealistic animation and rendering*, NPAR '08, pages 9–13, New York, NY, USA, 2008. ACM.

[26] NVIDIA Corporation. GPU programming guide version for GeForce 8 and later GPUs. Technical report, NVIDIA Corporation, 2008.

[27] NVIDIA Corporation. OpenGL SDK 10 code samples: Simple depth float, 2008.

[28] Trevor F Cox and Michael AA Cox. *Multidimensional scaling*. CRC Press, 2010.

[29] Cyril Crassin. Fast and accurate single-pass a-buffer using opengl 4.0+, icare3d blog, 2010.

[30] Cyril Crassin. Icare3D blog: Linked lists of fragment pages, 2010.

[31] Edilson De Aguiar, Christian Theobalt, Sebastian Thrun, and Hans-Peter Seidel. Automatic conversion of mesh animations into skeleton-based animations. *Computer Graphics Forum*, 27(2):389–397, 2008.

[32] Peng Du, Horace Ho-Shing Ip, Bei Hua, and Jun Feng. Using surface variability characteristics for segmentation of deformable 3d objects with application to piecewise statistical deformable model. *The Visual Computer*, 28(5):493–509, 2012.

[33] Ludovic Dutreve, Alexandre Meyer, and Saïda Bouakaz. Real-time dynamic wrinkles of face for animated skinned mesh. In *Advances in Visual Computing*, pages 25–34. Springer, 2009.

[34] David Eppstein and Elena Mumford. Self-overlapping curves revisited. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 160–169, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[35] David Epstein, Frederik W. Jansen, and Jarek Rossignac. Z-buffer rendering from CSG: The trickle algorithm. *IBM Research Report*, RC15182, 1989.

[36] Cass Everitt. Interactive order-independent transparency. *Technical Report, Nvidia Corporation*, 2(6):7, 2001.

[37] Wei-Wen Feng, Byung-Uck Kim, Yizhou Yu, Liang Peng, and John Hart. Feature-preserving triangular geometry images for level-of-detail representation of static and skinned meshes. *ACM Transactions on Graphics (TOG)*, 29(2):11, 2010.

[38] James Gain and Dominique Bechmann. A survey of spatial deformation from a user-centered perspective. *ACM Transactions on Graphics*, 27(4):1–21, October 2008.

[39] Naga K. Govindaraju, Michael Henson, Ming C. Lin, and Dinesh Manocha. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 49–56, New York, NY, USA, 2005. ACM.

[40] Sudipto Guha, Shankar Krishnan, Kamesh Munagala, and Suresh Venkatasubramanian. Application of the two-sided depth test to CSG rendering. *Proceedings of the 2003 symposium on Interactive 3D graphics - SI3D '03*, page 177, 2003.

[41] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525, 2006.

[42] Igor Guskov and Andrei Khodakovsky. Wavelet compression of parametrically coherent mesh sequences. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 183–192. Eurographics Association, 2004.

[43] John Hable and Jarek Rossignac. Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph.*, 24:1024–1031, July 2005.

[44] John Hable and Jarek Rossignac. CST: constructive solid trimming for rendering BReps and CSG. *IEEE transactions on visualization and computer graphics*, 13(5):1004–14, 2007.

[45] Torgeir Hagland. A fast and simple skinning technique. In *Game Programming Gems*, pages 471–475. Charles River Media, 2000.

[46] Jesse D Hall and John C Hart. GPU acceleration of iterative clustering. In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[47] Andrew J Hanson. *Visualizing Quaternions*. Morgan Kaufmann, 2006.

[48] Nils Hasler, Thorsten Thormählen, Bodo Rosenhahn, and Hans-Peter Seidel. Learning skeletons for shape and pose. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 23–30. ACM, 2010.

[49] Oliver Heim, Carl S. Marshall, and Adam Lake. Fast collision detection for 3d bones-based articulated characters. *Game Programming Gems*, 4:503–514, 2004.

[50] Jeff Alan Heisserman. *Generative geometric design and boundary solid grammars.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-16022.

[51] Jim Hejl. Hardware skinning with quaternions. *Game Programming Gems*, 4:487–495, 2004.

[52] Russ Herrell, Joe Baldwin, and Chris Wilcox. High-quality polygon edging. *IEEE Comput. Graph. Appl.*, 15:68–74, July 1995.

[53] Ruizhen Hu, Lubin Fan, and Ligang Liu. Co-segmentation of 3d shapes via subspace clustering. *Computer Graphics Forum*, 31(5):1703–1713, 2012.

[54] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30:78:1–78:8, August 2011.

[55] Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Transactions on Graphics (TOG)*, 24(3):399–407, 2005.

[56] Hanyoung Jang and JungHyun Han. Fast collision detection using the A-buffer. *Vis. Comput.*, 24(7):659–667, July 2008.

[57] Frederik W. Jansen. Depth-order point classification techniques for CSG display algorithms. *ACM Trans. Graph.*, 10:40–70, January 1991.

[58] Yuntao Jia, Wei-wen Feng, and Yizhou Yu. Transplanting and editing animations on skinned meshes. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, pages 431–434. IEEE, 2007.

[59] Di Jiang and Neil F Stewart. Robustness of boolean operations on subdivision-surface models. In *Numerical Validation in Current Hardware Architectures*, pages 161–174. Springer, 2009.

[60] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. *ACM Trans. Graph.*, 26, July 2007.

[61] N.P. Jouppi and C.F. Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 85–93. ACM New York, NY, USA, 1999.

[62] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, 24:561–566, July 2005.

[63] Dan Julius, Vladislav Kraevoy, and Alla Sheffer. D-charts: Quasi-developable mesh segmentation. *Computer Graphics Forum*, 24(3):581–590, 2005.

[64] Zachi Karni and Craig Gotsman. Compression of soft-body animation sequences. *Computers & Graphics*, 28(1):25–34, 2004.

[65] Anil Kaul and Jarek Rossignac. Solid-interpolating deformations: Construction and animation of PIPs. *Computers & Graphics*, 16(1):107–115, 1992.

[66] Ladislav Kavan. *Real-time Skeletal Animation*. PhD thesis, Faculty of Electrical Engineering Department of Computer Science and Engineering, Czech Technical University, 2007.

[67] Ladislav Kavan, Steven Collins, Jií Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics*, 27(4):1–23, 2008.

[68] Ladislav Kavan, Rachel McDonnell, Simon Dobbyn, Jiří Žára, and Carol O'Sullivan. Skinning arbitrary deformations. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 53–60. ACM, 2007.

[69] Ladislav Kavan, P-P Sloan, and Carol O'Sullivan. Fast and efficient skinning of animated meshes. *Computer Graphics Forum*, 29(2):327–336, 2010.

[70] Ladislav Kavan and J. Žára. Fast collision detection for skeletally deformable models. *Computer Graphics Forum*, 24(3):363–372, 2005.

[71] Isaac Victor Kerlow. *The Art of 3-D Computer Animation and Imaging (2Nd Ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[72] John Kessenich. The OpenGL shading language version: 1.50, document revision: 11, 2009.

[73] Scott Kircher and Michael Garland. Editing arbitrarily deforming surface animations. *ACM Trans. Graph.*, 25(3):1098–1107, July 2006.

[74] Florian Kirsch and Jürgen Döllner. OpenCSG: a library for image-based CSG rendering. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 49–49, Berkeley, CA, USA, 2005. USENIX Association.

[75] Pyarelal Knowles, Geoff Leach, and Fabio Zambetta. Efficient layered fragment buffer techniques. In *OpenGL Insights*, pages 279–292. CRC Press, 2012.

[76] Kai J. Kohlhoff, Marc H. Sosnick, William T. Hsu, Vijay S. Pande, and Russ B. Altman. CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, 27(16):2322–2323, 2011.

[77] Paul G. Kry, Doug L. James, and Dinesh K. Pai. Eigenskin: Real time large deformation character skinning in hardware. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, pages 153–159, New York, NY, USA, 2002. ACM.

[78] J. Lander. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, 5(May):11–16, 1998.

[79] J. Lander. Over my dead, polygonal body. *Game Developer Magazine*, 17(October), 1999.

[80] Binh Huy Le and Zhigang Deng. Smooth skinning decomposition with rigid bones. *ACM Transactions on Graphics (TOG)*, 31(6):199, 2012.

[81] Dongwoon Lee, Michael Glueck, Azam Khan, Eugene Fiume, and Ken Jackson. A survey of modeling and simulation of skeletal muscle. *Foundations and Trends in Computer Graphics and Vision*, 2010.

[82] M. Lee. Seven ways to skin a mesh: Character skinning revisited for modern GPUs. In *Proceedings of GameFest, Microsoft Game Technology Conference*, 2007.

[83] Tong-Yee Lee, Ping-Hsien Lin, Shaur-Uei Yan, and Chun-Hao Lin. Mesh decomposition using motion information from animation sequences. *Computer Animation and Virtual Worlds*, 16(3-4):519–529, 2005.

[84] Tong-Yee Lee, Yu-Shuen Wang, and Tai-Guang Chen. Segmenting a deforming mesh into near-rigid components. *The Visual Computer*, 22(9-11):729–739, 2006.

[85] Bin Liao, Chunxia Xiao, Meng Liu, Zhao Dong, and Qunsheng Peng. Fast hierarchical animated object decomposition using approximately invariant signature. *The Visual Computer*, 28(4):387–399, 2012.

[86] Yaron Lipman, David Levin, and Daniel Cohen-Or. Green coordinates. *ACM Trans. Graph.*, 27:78:1–78:10, August 2008.

[87] Jaroslaw Konrad Lipowski. Multi-layered framebuffer condensation: the l-buffer concept. In *Proceedings of the 2010 international conference on Computer vision and graphics: Part II*, ICCVG'10, pages 89–97, Berlin, Heidelberg, 2010. Springer-Verlag.

[88] Baoquan Liu, Li-Yi Wei, Ying-Qing Xu, and Enhua Wu. Multi-layer depth peeling via fragment sort. In *11th IEEE International Conference on Computer-Aided Design and Computer Graphics, 2009.*, pages 452 –456, 8 2009.

[89] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 51–57, New York, NY, USA, 2009. ACM.

[90] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM.

[91] Zhenbao Liu, Sicong Tang, Shuhui Bu, and Hao Zhang. New evaluation metrics for mesh segmentation. *Computers & Graphics*, 2013.

[92] George Maestri. *Digital Character Animation 3*. New Riders Publishing, Thousand Oaks, CA, USA, 2005.

[93] N. Magnenat-Thalmann, R. Laperriere, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proc. Graphics Interface*, volume 88, pages 26–33, 1988.

[94] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9:43–55, July 1989.

[95] William R. Mark and Kekoa Proudfoot. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, pages 57–64, New York, NY, USA, 2001. ACM.

[96] Marilena Maule, João Comba, Rafael Torchelsen, and Rui Bastos. Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 103–118, New York, NY, USA, 2013. ACM.

[97] Marilena Maule, Joo L.D. Comba, Rafael P. Torchelsen, and Rui Bastos. A survey of raster-based transparency techniques. *Computers & Graphics*, 35(6):1023 – 1034, 2011.

[98] James McCann and Nancy Pollard. Local layering. *ACM Trans. Graph.*, 28:84:1–84:7, July 2009.

[99] J Michael McCarthy. *Introduction to theoretical kinematics*. MIT press, 1990.

[100] Tim McLaughlin, Larry Cutler, and David Coleman. Character rigging, deformations, and simulations in film and game production. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, pages 5:1–5:18, New York, NY, USA, 2011. ACM.

[101] Houman Meshkin. Sort-independent alpha blending. In *Game Developers Conference 2007*, 2007.

[102] Alex Mohr and Michael Gleicher. Building efficient, accurate character skins from examples. *ACM Transactions on Graphics (TOG)*, 22(3):562–568, 2003.

[103] Uddipan Mukherjee, M. Gopi, and Jarek Rossignac. Immersion and embedding of self-crossing loops. In *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling*, SBIM '11, pages 31–38, New York, NY, USA, 2011. ACM.

[104] T. M. Murali and Thomas A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 155–ff., New York, NY, USA, 1997. ACM.

[105] Kevin Myers and Louis Bavoil. Stencil routed A-buffer. *ACM SIGGRAPH 2007 Sketches*, 2007.

[106] Diego Nehab, Joshua Barczak, and Pedro V. Sander. Triangle order optimization for graphics hardware computation culling. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 207–211, New York, NY, USA, 2006. ACM.

[107] F. S. Nooruddin and Greg Turk. Interior/exterior classification of polygonal models. In *Proceedings of the conference on Visualization '00*, VIS '00, pages 415–422, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[108] Anjul Patney, Stanley Tzeng, and John D. Owens. Fragment-parallel composite and filter. *Computer Graphics Forum*, 29(4):1251–1258, 2010.

[109] Frédéric Payan and Marc Antonini. Temporal wavelet-based compression for 3d animated models. *Computers & Graphics*, 31(1):77–88, 2007.

[110] Craig Peeper. Prefix sum pass to linearize A-buffer storage. *U.S. Patent*, (2008/0316214), 2008.

[111] Jingliang Peng, C.S. Kim, and C.C. Jay Kuo. Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6):688–733, 2005.

[112] Emil Persson. Depth in-depth. Technical report, ATI Technologies Inc., 2007.

[113] Simon Pilgrim, Anthony Steed, and Alberto Aguado. Progressive skinning for character animation. *Comput. Animat. Virtual Worlds*, 18(4-5):473–481, September 2007.

[114] Aristides A. G. Requicha and H.B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30–44, Jan 1985.

[115] Jarek Rossignac. Ordered boolean list (OBL): Reducing the footprint for evaluating boolean expressions. *IEEE Transactions onVisualization and Computer Graphics*, 17(9):1337 –1351, sept. 2011.

[116] Jarek Rossignac, Ioannis Fudos, and Andreas A. Vasilakis. Direct rendering of boolean combinations of self-trimmed surfaces. *Comput. Aided Des.*, 45(2):288–300, February 2013.

[117] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: cross-sections and interferences. *SIGGRAPH Comput. Graph.*, 26:353–360, July 1992.

[118] Jarek Rossignac and Aristides A. G. Requicha. Offsetting operations in solid modelling. *Comput. Aided Geom. Des.*, 3:129–148, August 1986.

[119] Jarek R. Rossignac and Maarten van Emmerik. Hidden contours on a frame-buffer. In *Eurographics Workshop on Graphics Hardware*, pages 188–203, Cambridge, UK, 1992. Eurographics Association.

[120] Jarek R. Rossignac and Herbert B. Voelcker. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Trans. Graph.*, 8:51–87, November 1988.

[121] Marco Salvi. Advances in real-time rendering in games: Pixel synchronization: Solving old graphics problems with new data structures. In *ACM SIGGRAPH 2013 courses*, SIGGRAPH '13, New York, NY, USA, 2013. ACM.

[122] Tatiana Samoilov and Gershon Elber. Self-intersection elimination in metamorphosis of two-dimensional curves. *The Visual Computer*, 14:415–428, 1998. 10.1007/s003710050152.

[123] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26(3), July 2007.

[124] Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Simple and efficient compression of animation sequences. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217. ACM, 2005.

[125] Yann Savoye and Alexandre Meyer. Multi-layer level of detail for character animation. In *Workshop in Virtual Reality Interactions and Physical Simulation VRIPHYS (2008)*, 2008.

[126] S Schaefer and C Yuksel. Example-based skeleton extraction. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 153–162. Eurographics Association, 2007.

[127] Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V. Sander, Michael Wimmer, and Elmar Eisemann. Temporal coherence methods in real-time rendering. *Computer Graphics Forum*, 31(8):2378–2408, 2012.

[128] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification of version 3.3 core profile, 2010.

[129] Dean Sekulic. Efficient occlusion culling. In *Gpu Gems*, pages 487–203, Boston, USA, 2004. Addison-Wesley.

[130] Ariel Shamir. A survey on mesh segmentation techniques. *Computer graphics forum*, 27(6):1539–1556, 2008.

[131] Shymon Shlafman, Ayellet Tal, and Sagi Katz. Metamorphosis of polyhedral surfaces using decomposition. *Computer Graphics Forum*, 21(3):219–228, 2002.

[132] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of the Conference on Graphics Interface '92*, pages 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[133] Oana Sidi, Oliver van Kaick, Yanir Kleiman, Hao Zhang, and Daniel Cohen-Or. Unsupervised co-segmentation of a set of shapes via descriptor-space spectral clustering. *ACM Trans. on Graphics (Proc. SIGGRAPH Asia)*, 30(6):126:1–126:10, 2011.

[134] Erik Sintorn and Ulf Assarsson. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 157–162, New York, NY, USA, 2008. ACM.

[135] Erik Sintorn and Ulf Assarsson. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 67–74, New York, NY, USA, 2009. ACM.

[136] Ljiljana Skrba, Lionel Reveret, Franck Htroy, Marie-Paule Cani, and Carol O'Sullivan. Animating quadrupeds: Methods and applications. *Computer Graphics Forum*, 28(6):1541–1560, 2009.

[137] Carsten Stoll, Juergen Gall, Edilson de Aguiar, Sebastian Thrun, and Christian Theobalt. Video-based reconstruction of animatable human characters. *ACM Trans. Graph.*, 29(6):139:1–139:10, December 2010.

[138] Robert W Sumner and Jovan Popović. Deformation transfer for triangle meshes. *ACM Transactions on Graphics (TOG)*, 23(3):399–405, 2004.

[139] Robert W. Sumner, Johannes Schmid, and Mark Pauly. Embedded deformation for shape manipulation. *ACM Transactions on Graphics*, 26(3):80, July 2007.

[140] Robert W. Sumner, Matthias Zwicker, Craig Gotsman, and Jovan Popović. Mesh-based inverse kinematics. *ACM Trans. Graph.*, 24(3):488–495, July 2005.

[141] Tatsuhiko Suzuki, Shigeo Takahashi, and Jason Shepherd. An interior surface generation method for all-hexahedral meshing. *Engineering with Computers*, 26:303–316, 2010. 10.1007/s00366-009-0159-9.

[142] Christian Theobalt, Christian Rössl, Edilson de Aguiar, and Hans-Peter Seidel. Animation collage. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '07, pages 271–280. Eurographics Association, 2007.

[143] R.B. Tilove and Aristides A.G. Requicha. Closure of boolean operations on geometric entities. *Computer-Aided Design*, 12(5):219 – 220, 1980.

[144] Stanley Tzeng, Anjul Patney, and John D Owens. Efficient adaptive tiling for programmable rendering. In *Symposium on Interactive 3D Graphics and Games*, pages 201–201. ACM, 2011.

[145] Andreas A. Vasilakis, George Antonopoulos, and Ioannis Fudos. Pose-to-pose skinning of animated meshes. In *ACM/Eurographics Symposium on Computer Animation Posters*, August 2011.

[146] Andreas A. Vasilakis and Ioannis Fudos. Skeleton-based Rigid Skinning for Character Animation. In *GRAPP 2009*, pages 302–308, Lisboa, 2009.

[147] Andreas A. Vasilakis and Ioannis Fudos. Gpu rigid skinning based on a refined skeletonization method. *Computer Animation and Virtual Worlds*, 22(1):27–46, 2011.

[148] Andreas A. Vasilakis and Ioannis Fudos. Z-fighting Aware Depth Peeling. In *SIGGRAPH Posters*. ACM, 2011.

[149] Andreas A. Vasilakis and Ioannis Fudos. S-buffer: Sparsity-aware multi-fragment rendering. In *Proceedings of Eurographics 2012 Short Papers*, pages 101–104, Cagliari, Sardinia, Italy, 2012.

[150] Andreas A. Vasilakis and Ioannis Fudos. Depth-fighting aware methods for multi-fragment rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):967–977, 2013.

[151] Andreas A. Vasilakis and Ioannis Fudos. k+-buffer: Fragment synchronized k-buffer. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, New York, NY, USA, 2014. ACM.

[152] Andreas A. Vasilakis and Ioannis Fudos. Pose partitioning for multi-resolution segmentation of arbitrary mesh animations. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2014)*, 33(2):?–?, 2014.

[153] Daniel Vlasic, Ilya Baran, Wojciech Matusik, and Jovan Popović. Articulated mesh animation from multi-view silhouettes. *ACM Transactions on Graphics*, 27(3):1, August 2008.

[154] H. Voelcker, A. Requicha, E. Hartquist, W. Fisher, J. Metzger, R. Tilove, N. Birrell, W. Hunt, G. Armstrong, T. Check, R. Moote, and J. McSweeney. The PADL-1.0/2 system for defining and displaying solid objects. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 257–263, New York, NY, USA, 1978. ACM.

[155] Anna Vögele, Max Hermann, Björn Krüger, and Reinhard Klein. Interactive steering of mesh animations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '12, pages 53–58, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[156] Wolfram von Funck, Holger Theisel, and Hans-Peter Seidel. Volume-preserving mesh skinning. In Oliver Deussen, Daniel Keim, and Dietmar Saupe, editors, *13th International Fall Workshop on Vision, Modeling and Visualization*, pages 407–414, Konstanz, Germany, October 2008. Akademische Verlagsgesellschaft AKA.

[157] Robert Y Wang, Kari Pulli, and Jovan Popović. Real-time enveloping with rotational regression. *ACM Transactions on Graphics (TOG)*, 26(3):73, 2007.

[158] Wencheng Wang and Guofu Xie. Memory-efficient single-pass GPU rendering of multifragment effects. *IEEE Transactions on Visualization and Computer Graphics*, 19(8):1307–1316, August 2013.

[159] Yajun Wang, Jiaping Wang, Nicolas Holzschuch, Kartic Subr, Jun-Hai Yong, and Baining Guo. Real-time rendering of heterogeneous translucent objects with arbitrary shapes. *Computer Graphics Forum*, 29(2):497–506, 2010.

[160] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 7–14, New York, NY, USA, 2005. ACM.

[161] C.M. Wittenbrink. R-buffer: a pointerless A-buffer hardware architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 73–80. ACM New York, NY, USA, 2001.

[162] Stefanie Wuhrer and Alan Brunton. Segmenting animated objects into near-rigid components. *The Visual Computer*, 26(2):147–155, 2010.

[163] Xiao Xian, Seah Hock Soon, and Tian Feng. Skinning on progressive decimated models. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1054–1058. IEEE, 2008.

[164] Wei-Wei Xu and Kun Zhou. Gradient domain mesh deformation: A survey. *Journal of Computer Science and Technology*, 24(1):6–18, March 2009.

[165] Weiwei Xu, Kun Zhou, Yizhou Yu, Qifeng Tan, Qunsheng Peng, and Baining Guo. Gradient domain editing of deforming mesh sequences. *ACM Trans. Graph.*, 26(3), July 2007.

[166] Han-Bing Yan, Shi-Min Hu, R.R. Martin, and Yong-Liang Yang. Shape deformation using a skeleton to drive simplex transformations. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):693 –706, may-june 2008.

[167] Jason C. Yang, Justin Hensley, Holger Grn, and Nicolas Thibieroz. Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum*, 29(4):1297–1304, 2010.

[168] Xuan Yu, Jason C. Yang, Justin Hensley, Takahiro Harada, and Jingyi Yu. A framework for rendering complex scattering effects on hair. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 111–118, New York, NY, USA, 2012. ACM.

[169] Nan Zhang. Memory-hazard-aware k-buffer algorithm for order-independent transparency rendering. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints):1, 2013.

[170] Hanli Zhao, CharlieC.L. Wang, Yong Chen, and Xiaogang Jin. Parallel and efficient boolean on polygonal solids. *The Visual Computer*, 27(6-8):507–517, 2011.

Author's Publications

1. **Andreas A. Vasilakis** and Ioannis Fudos, *k^+-buffer: Fragment Synchronized k-buffer*, In Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14), San Francisco, California, USA, 14-16 March, 2014.

2. **Andreas A. Vasilakis** and Ioannis Fudos, *Pose Partitioning for Multi-resolution Segmentation of Arbitrary Mesh Animations*, Computer Graphics Forum (Proceedings of Eurographics 2014), vol. 33 no. 2, pages ?-?, April, 2014.

3. **Andreas A. Vasilakis** and Ioannis Fudos, *Depth-fighting Aware Methods for Multifragment Rendering*, IEEE Transactions on Visualization and Computer Graphics, vol. 19, no. 6, pp. 967-977, June, 2013.

4. Jarek Rossignac, Ioannis Fudos and **Andreas A. Vasilakis**, *Direct Rendering of Boolean Combinations of Self-Trimmed Surfaces*, Computer-Aided Design, Volume 45, Issue 2, February 2013, Pages 288-300, ISSN 0010-4485, 10.1016/j.cad.2012.10.012.

5. **Andreas A. Vasilakis** and Ioannis Fudos, *S-buffer: Sparsity-aware Multi-fragment Rendering*, Proceedings of Eurographics 2012, Short Papers, pages 101-104, Cagliari, Italy, May 13-18, 2012.

6. **Andreas A. Vasilakis** and Ioannis Fudos, *Z-fighting aware Depth Peeling*, Siggraph Posters, Vancouver, Canada, August 7-11, 2011.

7. **Andreas A. Vasilakis**, George Antonopoulos and Ioannis Fudos, *Pose-to-Pose Skinning of Animated Meshes*, ACM/Eurographics Symposium on Computer Animation Posters, Vancouver, Canada, August 5-7, 2011.

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

9. **Andreas A. Vasilakis** and Ioannis Fudos, *Skeleton-based Rigid Skinning for Character Animation*, in Proceedings of the Fourth International Conference on Computer Graphics Theory and Applications, Lisbon, Portugal, February 5-8, 2009.

Short Vita

Andreas-Alexandros Vasilakis was born on October 12, 1983, in Corfu. He graduated from the 2nd High School of the same city and was admitted for pursuing undergraduate and postgraduate studies at the Computer Science and Engineering Department of University of Ioannina. He received his BSc and MSc degrees in 2006 and 2008 and since then he has been a Ph.D. candidate at the same Department under the supervision of Prof. Ioannis Fudos. His PhD studies were supported by a scholarship from the Heraclitus II grant through the operational programme "Education and Lifelong Learning" through the European Social Fund, 2010-2013. He has published posters and papers in top conferences and journals in the domain of character animation and interactive 3D rendering, such as SIGGRAPH, EUROGRAPHICS, SCA, I3D, SPM, GRAPP, TVCG and CAVW. His research interests include skinning algorithms, segmentation, animation compression, gpu programming and multifragment rendering.