Universitat de Girona

PhD Thesis

CONTINUITY AND INTERPOLATION TECHNIQUES FOR COMPUTER GRAPHICS

Francisco González García

2013

delivered by
EG EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

PhD Thesis

CONTINUITY AND INTERPOLATION TECHNIQUES FOR COMPUTER GRAPHICS

Francisco González García

2013

Doctoral Programme in Technology
Computer Science

PhD Advisor
Dr. Gustavo Patow

Memòria presentada per optar al títol de doctor per la Universitat de Girona

Universitat de Girona

El Dr. Gustavo Patow, professor associat del departament d'Informàtica i Matemàtica Aplicada (IMA) de la Universitat de Girona,

CERTIFICO:

Que aquest treball, titulat *Continuity and interpolation techniques for Computer Graphics*, que presenta Francisco González García per a l'obtenció del títol de doctor, ha estat realitzat sota la meva direcció.

Signatura

Girona, 13 de Nomvembre del 2012

ViRVIG
UdG

Continuity and interpolation techniques for Computer Graphics, Francisco González García, PhD in Technology, Universitat de Girona, IMA - Informatica i Matemàtica Aplicada, November 2012

Dedicado a mi padre y a mi pareja,sin vosotros no hubiera sido posible. Os quiero!!!

ACKNOWLEDGMENTS

Me gustaría dar las gracias a un cojunto de personas, sin las cuáles, esta tesis jamás hubiera sido lo mismo.

Querría agradecerte Gus el haber podido realizar este doctorado. Sin ti, tu ayuda, guía, tiempo, esfuerzo y dedicación no hubiera sido posible. Es muy hermoso compartir objetivos, inquietudes e ilusiones con una persona y llevarlas a cabo, verdad? Ha sido una aventura emocionante, llena de aprendizajes constantes, tanto en el terreno profesional como en el personal. Como siempre te he dicho, no siento que sea mi tesis, si no un proyecto común que hemos realizado los dos y que ha durado unos intensos cuatro años. He podido realizar el doctorado junto a un brillante científico. Sin duda alguna, no hubiera podido elegir mejor director!!! ¿Pero sabes qué? Poca gente puede decir que ha realizado el doctorado con un buen y gran amigo.

Me gustaría agradecer, de forma especial, a mi compañero de vida, Fran. Has tenido muchísima paciencia, comprensión y empatía por todo cuanto me ha sucedido durante estos últimos años. Tu amor ha sido esa energía que me ha ayudado en muchos momentos y que sin él todo hubiera sido mucho más complicado. Gracias por comprender mi dedicación por mi trabajo, mi pasión por las cosas que hago, por llegar tarde tantas veces a casa y dedicarte menos tiempo del que sin duda mereces. Tú más que nadie sabes que estos últimos años no han sido fáciles para mi, pero gracias a tu apoyo incondicional he podido tirar hacia delante. Gracias por levantarme cuando no podía. Gracias por iluminarme cuando no veía salida. Gracias por ser quién eres y hacerlo a mi lado. Te quiero.

A mi padre y a mi madre. Gracias mama y papa, porque no hubiera podido desear mejores padres. Sois un ejemplo de superación, fuerza, coraje y valentía. Os quiero y siempre os llevaré en mi corazón. Papa, sé que te sientes orgulloso de mi y ello me llena el corazón. Gracias por siempre estar ahí, vigilando y protegiéndome. Te llevo dentro de mi. Esta tesis te la dedico a ti. Sin duda un ejemplo de superación, bondad, justicia y amor. Mama y papa, os quiero más de lo que las palabras pueden expresar.

A mis hermanas, más que amigas, Sílvia Francia y Sílvia Abril. Cuántos momentos hemos pasado juntos, verdad? Qué bonito es tener personas que sin ser de tu familia las sientas como tal. Para mi sois muy importantes, os necesito en mi vida, ya que me llenáis de todo lo hermoso que sin duda tenéis. Mendita, te quiero muchísimo, lo sabes verdad? Jamás olvidaré cada una de tus palabras, ya que como siempre te dije son especiales. Nacen del corazón. Gracias por cantar a mi lado, fue muy especial!!! Jamás lo olvidaré!!! Siempre juntos, siempre el trío de luz. Vacuneta, tu mejor que nadie sabes el esfuerzo que hay detrás de una tesis. Gracias por ser mi amiga, como dije antes, una hermana. Nos queremos, nos respetamos, nos comprendemos y nos valoramos de forma sincera. Gracias por ser y estar en mi vida. Y a ti Dolors, no me olvidaría jamás. Gracias por todos los masajes!!! No sabes lo bien que me iban después de tantas horas sentado delante del ordenador. Pero más importante aún, gracias por todo tu amor, tu calidez, tu bondad,

siempre dispuesta a ayudarme. Te quiero mucho!!!

Me gustaría agradecer también a TODOS los amigos de Ataraxia, especialmente a Inma y Esther. Un grupo de ayuda al ser humano lleno de valores y mucho amor. Gracias por ayudarme a crecer y a convertirme en el ser que soy hoy. Gracias por ser y por estar siempre a mi lado, por protegerme y valorarme por quién soy. Os quiero con todo mi corazón.

A mis compañeros de despacho y de grupo. Gracias a todos por tantos momentos y cafés que hemos compartido. Especialmente a mis queridos amigos Tere e Isma. Os agradezco de corazón vuestra ayuda, vuestros consejos. Por todos los momentos que hemos vivido soy y me siento afortunado. He aprendido y crecido gracias a vosotros, profesional y personalmente. Os deseo de corazón lo mejor en el futuro que os depara. Sin duda os lo merecéis!!! Y quién sabe si en algún momento nos volveremos a encontrar. Hasta siempre!!!

Xavier, no quería olvidarme de ti. Gracias por tu ayuda y predisposición en todo momento. Gracias, porque si no hubiera sido por ello, probablemente no hubiera disfrutado de la beca que se me concedió. Te estoy muy agradecido. También me gustaría acordarme de Pere Brunet, tus consejos cuando tan solo estábamos empezando fueron de gran ayuda. Gracias de corazón.

No me gustaría acabar estos agradecimientos sin dar las gracias a la vida!!! Sin duda alguna es un regalo y hay que aprovecharla. Gracias por hilar la trama perfecta para que esta obra se pudiera llevar a cabo. La experiencia que ha supuesto hacer el doctorado es prácticamente indescriptible para mi. Sin duda hoy no soy el mismo que hace cuatro años, he evolucionado como profesional y más importante aún, como ser humano. Me siento afortunado porque en estos últimos años de mi vida he podido hacer una de las cosas que más me gustaba y al lado de las personas que quise. Gracias, gracias y gracias.

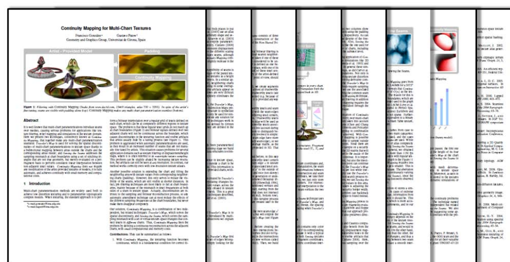
Francisco González García.

Esta tesis ha sido financiada mediante una beca FPU (Ministerio de Educación, Cultura y Deporte), así como de los siguientes proyectos: "Avances en la realidad virtual para aplicaciones punteras" (TIN2010-20590-C02-02) y "Modelado, visualización, animación y anánalisis de entornos 3D altamente complejos en sistemas de realidad virtual interactivos" (TIN2007-67982-C02).

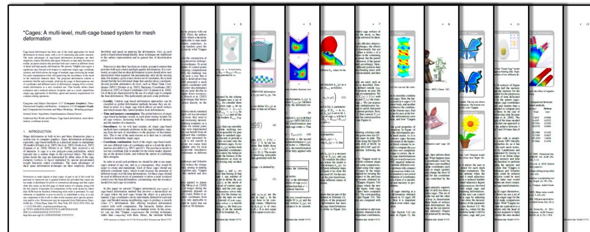
PUBLICATIONS

This thesis has given as a result several publications in some of the most well known journals and conferences in Computer Graphics:

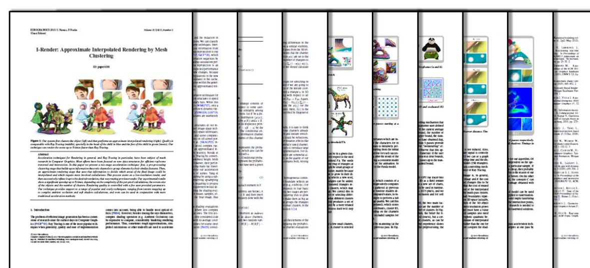
- ***Continuity mapping for multi-chart textures***. González Francisco and Patow Gustavo. SIGGRAPH Asia '09, Yokohama, Japan. ACM Transactions on Graphics, Volume 28, issue 5, pages 109:1–109:8. December 2009. ISSN 0730-0301. Article 109. doi: http://doi.acm.org/10.1145/1618452.1618455.
Publisher ACM. New York, USA.


Continuity Mapping for Multi-Chart Textures

- ****Cages: A multi-level, multi-cage based system for mesh deformation.*** González Francisco, Paradinas Teresa, Coll Narcís and Patow Gustavo. ACM Transactions on Graphics, Volume 32, pages 24:1 –24:13, July 2013. ISSN 0730-0301. Article 24. doi: 10.1145/2487228.2487232.
Publisher ACM. New York, USA. Presented in SIGGRAPH 2013, Anaheim, USA.


*Cages: A multi-level, multi-cage based system for mesh deformation

- ***I-Render: Approximate interpolated rendering by mesh clustering***. González Francisco, García Ismael and Patow Gustavo. *Submitted*.


I-Render: Approximate Interpolated Rendering by Mesh Clustering

LIST OF FIGURES

Figure 1 Modeling, deformation and rendering pipeline in a Computer Graphics application. 8
Figure 2 A possible parameterization (multi-chart) of a surface into texture space. 16
Figure 3 Multi-chart parameterization discontinuities. 17
Figure 4 Parameterization of a surface mesh into object space by the use of another additional surface C. 18
Figure 5 Projection of a surface mesh into screen space. 18
Figure 6 Discontinuities of a surface mesh projected into screen space. 19
Figure 7 Lienar interpolation. 20
Figure 8 Bilinear interpolation. 21
Figure 9 Trilinear interpolation. 21
Figure 10 Mesh parameterization techniques using 2D data structures with one single or several charts. Images from [50], [34], [72], [12], [67] and [99]. 22
Figure 11 Mesh parameterization techniques using 3D data structures. Images from [6], [90], [47] and [96]. 23
Figure 12 Techniques to hide seams by its placement on the 3D mdoel. Images from [80], [43] and [68]. 24
Figure 13 Techniques to hide seams by blending techniques. Images from [66], [55] and [21]. 25
Figure 14 Cage-based deformation methods use a cage to drive the deformation of a model. 26
Figure 15 Different cage-based methods for mesh deformation. From left to right: Mean Value Coordinates (MVC), Positive Mean Value Coordinates (PMVC), Harmonic Coordinates (HC) and Green Coordinates (GC). Images from [39], [38], [53] and [54]. 26
Figure 16 Table showing a summary about coordinates continuity. 27
Figure 17 Cage-based deformation techniques applied to planar domains. Images from [56], [92] and [57]. 28
Figure 18 Several inter-frame acceleration techniques for rendering. Images from [59] and [82]. 29
Figure 19 Several intra-frame acceleration techniques for rendering. Images from [101], [46] and [30]. 29
Figure 20 Screen-space techniques for accelerating Ray-tracing. Images from [88], [95] and [60]. 30
Figure 21 Techniques that use Information Theory and apply it to several Computer Graphics fields. Images from [69], [31], [58], [23] and [71]. 33
Figure 22 Graphics pipeline with all the stages and programable parts. 34

Figure 23 Motivation. Fragments from different sides of a seam are parameterized onto disjoint areas in texture space (different orientation and stretching), leading to discontinuities that are more visible in close-up views. 39
Figure 24 Texture transfer artifacts between different parameterizations. 40
Figure 25 Continuity Mapping overview. 41
Figure 26 Transformation between edges s and s', ... 42
Figure 27 Traveler's Map definitions. Seams s and s' are paired using transformation $T_{s \to s'}$ (left). We create a security border (middle) around each chart storing references to the respective $T_{s \to s'}$ (right). 43
Figure 28 Texels-Seam edges association. Left: Association for trustworthy texels with a single seam edge. Right: Association for trustworthy texels with multiple seam edges. 44
Figure 29 Construction of *Sewing the Seams*. (a) Charts and seams in 3D. (b) Identify and join the trustworthy texel centers in every chart. (c) With *Traveler's Map*, transform the centers to the outside of the corresponding twin seam on the other chart. (d-e) Triangulate both the interior and the exterior centers, taking special care with corners where two seam edges meet. (f) The triangulation mapped back in 3D. 45
Figure 30 *Sewing the Seams* usage and data structures. Fragment f queries the list of triangles associated with the texel (T_1, T_2 and T_3), and point f is found in triangle T_3. 46
Figure 31 Sewing the Seams performance improvement by dividing each texel in four quadrants and storing 1 bit for 47
Figure 32 Seamless Texture Filtering on the bunny model (Atlas 1024^2). First and third column: padding. Second and fourth column: *Sewing the Seams*. 49
Figure 33 Seamless Texture Filtering on the Neptune model (Atlas 2048^2). First and third column: padding. Second and fourth column: *Sewing the Seams*. 50
Figure 34 Seamless Texture Filtering on the snake model (Atlas 1458x3200). First and third column: padding. Second and fourth column: *Sewing the Seams*. 50
Figure 35 *Sewing the seams* at different resolutions on the bunny model. Top row: *Sewing the Seams*. Bottom row: Padding. 51
Figure 36 A parameterized elephant model (13402 triangles) and two different results for continuous reaction-diffusion simulations. Left: padding. Right: *Continuity Mapping*. 51
Figure 37 The bunny model from Figure 32 with different results for continuous reaction-diffusion simulations. 52
Figure 38 Droplet simulations performed on the bunny model from Figure 32. 52
Figure 39 Multi-chart Relief Mapping. Comparison between simple padding, Indirection Maps and Continuity Mapping. 52
Figure 40 Tracing a ray with Multi-Chart Relief Mapping. 53
Figure 41 Multi-chart Relief Mapping on the Loiosh model. Comparison between simple padding, and Continuity Mapping. 53

Figure 42 Multi-chart Relief Mapping on the bunny model. Comparison between simple padding, Indirection Maps and Continuity Mapping. 54
Figure 43 Left: Memory consumption of *Continuity Mapping* depending on the atlas resolution. 55
Figure 44 Dependence of the frame-rate on the distance to the observer (bunny model, viewport: 1024×768). 56
Figure 45 Definition of a point inside a cage by a set of weights respect to the cage vertices. 63
Figure 46 Left: $c_0 = v_0 v_1 v_4 v_3$, $c_1 = v_3 v_4 v_6 v_5$, $c_2 = v_1 v_2 v_7 v_6 v_4$, $B_{c_0} = B_{c_0 c_1} \cup B_{c_0 c_2}$, $B_{c_1} = B_{c_0 c_1} \cup B_{c_1 c_2}$, $B_{c_2} = B_{c_0 c_2} \cup B_{c_1 c_2}$. Middle: Join cage generated by v_3. Right: Join cage generated by v_4. 63
Figure 47 Cube model at binding time and its correspondence influence map. 64
Figure 48 A comparison between piecewise deformations. (a) MVC/PMVC/HC deformation. (b) *Cages with MVC deformation, both for $J_{c_i}(p)$ and $T_{c_i}(p)$. (c) GC deformation. (d) *Cages with GC deformation, both for $J_{c_i}(p)$ and $T_{c_i}(p)$. The second row shows close views of the deformed model. Notice that only (b) and (d) are C^1. 64
Figure 49 Variation of weight $W(v, p)$ on different borders between two cages. 68
Figure 50 Influence map variation on the scheme shown in Figure 46. 70
Figure 51 Influence map variation on the chinchilla model. Left: Original model and initial cages. Right: Results obtained by using different h_{c_i} values for the left ear cage. Red and blue regions mean transformations $T_{c_i}(p)$ and $J_{c_i}(p)$ respectively are fully applied. 70
Figure 52 Multi-level deformation for coordinates not defined outside the cage. (a) Initial cages. (b) Direct cage vertex movement. (c) Parent transformation. 72
Figure 53 Camel model at binding time (left) and its corresponding influence map (right). 72
Figure 54 Twisting a prism using MVC (left) and GC (right). See the similarity between a single cage (a,d) and *Cages (b,e). (c,f) show the corresponding difference maps. Red means higher error and blue means lower error. 73
Figure 55 Two diferent deformations on the camel model with MVC (first column) and *Cages (second column). Third column shows the difference maps. Red means higher error and blue means lower error. . . 74
Figure 56 Influence map of the train model at binding time. 74
Figure 57 Locality: Comparison of single cage approaches and *Cages(a) single cage MVC, (b) *Cages whith MVC, (c) single cage GC, (d) *Cages with GC. 75
Figure 58 From left to right: the cages at binding time on the butterfly model, two different deformations using MVC and GC with *Cagesand a *Cagescombined GC/MVC deformation. 75
Figure 59 Combined deformations on the elk model. Left: Elk model at binding time with its influence map. Middle: *Cages combined MVC/HC deformation. Right: *Cages combined GC/HC deformation. 76

Figure 60 Deformations on the hand model. Columns from left to right: Deformation using MVC (observe the effect of negative coordinates of non-convex cages), and *Cages using MVC (third column) and HC (fourth column) as join transformations, respectively. 77
Figure 61 Fairness of the resulting MVC (left), GC (middle) and MVC/GC with *Cages (right) deformations. The top row shows the influence map and the cages at binding time. 78
Figure 62 Multiple cages meeting at a cage vertex. Left: Original model with 13 cages using different coordinates. Middle: close view. Right: Deformation with *Cages. 79
Figure 63 Multi-level deformation of the squirrel model. (a) Multi-level cages. (b) Leaf deformation: teeth cage. (c) Internal deformation: ears' cage. (d) Internal deformation: head cage. (e) Leaf deformation: face cage. 79
Figure 64 Deformation of the squirrel model using *Cages˙Left: The model and its multi-level cages at binding time. Right: Composition of a pose. . 80
Figure 65 Deformation involving interior points of the "Easter egg" model using *Cages˙Left: The model and the grid of cages at binding time. Highlighted vertices are interior points. Right: Composition of two different deformations. 80
Figure 66 Scene showing the multi-level deformations on the Squirrel and Chinchilla models. Left: Cages at binding time with different coordinates (Blue - MVC, Green - GC, Red - HC, and pink cage boundaries). Right: Composition of different poses. 81
Figure 67 Deformations of the Sintel model (66845 triangles) using *Cages˙Left: Cages at binding time with different coordinates (Blue - MVC, Green - GC, Red - HC, and pink cage boundaries). Right: Composition of different poses. 83
Figure 68 Deformation of the frog model. Left: Cages at binding time and original model. Right: Deformed cages and model using *Cages. . . 84
Figure 69 Iterative application of channel clustering: visibility, orientation and texture stretch. 92
Figure 70 Variation of clustering threshold Th. 96
Figure 71 Cluster smoothing. 97
Figure 72 A band between two clusters, C_i and C_j, defined around a boundary from vertex V_i to vertex V_e. 97
Figure 73 Iterative clustering refining for different model keyframes (left and right). 98
Figure 74 Rendering pattern for three passes starting at a 2×2-pixel image. . . 99
Figure 75 Re-used (R), interpolated (I) and evaluated (E) samples in the successive passes. 100
Figure 76 Hard shadow boundary preservation. Center: I-Render with no boundary preservation. Right: I-Render with boundary preservation. 101
Figure 77 Quality depending on the number of clusters. Left: the clusters, Right: renderings showing details and error values in false color (inset). 102

Figure 78 Quality depending on the number of passes. Top row: Ray Tracing reference and 2, 3 and 4 passes respectively. Bottom row: the clusters and the respective error images. The insets show details of the texturing and soft shadows. . . . 103
Figure 79 Quality depending on the number of passes. Top row: Ray Tracing reference and 2, 3 and 4 passes respectively. Bottom row: the clusters and the respective error images. The insets show details of the texturing and soft shadows. . . . 104
Figure 80 Quality depending on the observer distance. Timings in Figure 84. . . 105
Figure 81 Correct texture rendering. Left: clusters, Middle: Ray Tracing, Right: I-render. The lower insets show the smooth interpolation of the texture coordinates. . . . 106
Figure 82 Our system first performs a feature-based clustering of the object (left) and then reconstructs the final image by an approximate interpolated approach (right). Quality is comparable with Ray Tracing (middle), observe the texture details, shadow boundaries and correct visibility. Our technique can render the scene up to 12 times faster than Ray Tracing. . . . 107
Figure 83 Animation sequence of the panda model using our technique. . . . 108
Figure 84 Graph showing the relation between rendering time (in ms) and distance to the observer (in arbitrary units), for the model in Figure 80. . . . 108
Figure 85 Graph showing the relation between rendering time (in ms) and the number of passes for different resolutions, see Figure 78. . . . 109
Figure 86 Graph comparing the mean rendering time (in ms) between Ray Tracing and I-Render for several views and models. . . . 110

LIST OF TABLES

Table 1 Memory and time requirements for several h_{c_i} values for the chinchilla model. . . . 85
Table 2 Memory and time requirements for the Sintel and Squirrel models using *Cages and standard single cage approaches. . . . 85
Table 3 Table showing the differences between the cost of evaluated and interpolated pixels for Ray-Tracing and I-Render for several models shown in the paper. . . . 106

CONTENTS

1 INTRODUCTION 7
1.1 Contributions . . . 10
1.2 Thesis overview . . . 11

2 PREVIOUS WORK 13
2.1 Parameterization and continuity . . . 15
2.1.1 Texture space . . . 15
2.1.2 Object space . . . 17
2.1.3 Screen space . . . 17
2.2 Interpolation . . . 19
2.3 Continuity and interpolation techniques . . . 21
2.3.1 Texture space . . . 21
2.3.2 Object space . . . 25
2.3.3 Screen space . . . 28
2.4 Mesh clustering . . . 31
2.5 Information Theory . . . 32
2.5.1 Basic concepts . . . 32
2.5.2 Information Theory in Computer Graphics . . . 33
2.6 Graphics hardware pipeline . . . 34

3 CONTINUITY AND INTERPOLATION IN TEXTURE SPACE 37
3.1 Introduction . . . 39
3.2 Overview . . . 41
3.3 Traveler's Map . . . 42
3.3.1 Construction . . . 42
3.3.2 Usage . . . 43
3.4 Sewing the Seams . . . 43
3.4.1 Construction . . . 43
3.4.2 Storage details . . . 47
3.4.3 Filtering with Sewing the Seams . . . 47
3.5 Mip Mapping and Shader LoD . . . 48
3.6 Applications . . . 49
3.7 Results and Discussion . . . 54
3.8 Conclusions . . . 57

4 CONTINUITY AND INTERPOLATION IN OBJECT SPACE 59
4.1 Introduction . . . 61
4.2 *Cages . . . 62
4.2.1 Join transformation $J_{c_i}(p)$ . . . 64
4.2.2 Boundary weight function $b_{c_i}(p)$ . . . 69
4.2.3 Smooth Transformation $S_{c_i}(p)$ . . . 70
4.2.4 Multi-level deformations . . . 71

4.3 Results and discussion 72
4.4 Conlusions 86

5 CONTINUITY AND INTERPOLATION IN SCREEN SPACE 87
5.1 Introduction 89
5.2 Clustering 90
5.2.1 Information Theoretic Channels 90
5.2.2 Clustering a Single Channel 94
5.2.3 Smoothing 96
5.2.4 Animated Scenes 98
5.2.5 Threshold Selection 98
5.3 Rendering by Upsampling 99
5.3.1 Hard Shadows and Higher-Frequency Signals 100
5.3.2 Automatic Pass-Controlling Mechanism 101
5.4 Results and Discussion 101
5.5 Conclusions 111

6 CONCLUSIONS AND FUTURE WORK 113
6.1 Conclusions 115
6.2 Future work 115

I BIBLIOGRAPHY 119

BIBLIOGRAPHY 121

II APPENDIX 129

A *continuity mapping* PIXEL SHADER CODE 131

B *i-render* CUDA KERNELS 141

RESUM

En el camp dels gràfics per ordinador és una pràctica molt comuna utilitzar textures sobre els models 3D per a poder-los aplicar materials. Una vegada els models han estat texturats, aquests solen patir un procés de deformació, amb l'objectiu de poder crear noves postures que puguin ser més apropiades per a una escena determinada. A continuació, l'escena que conté tals models es visualitza mitjançant la utilització d'un algorisme de visualització. Així doncs, és evident que el texturat, deformació i la visualització són parts molt importants dintre dels gràfics per ordinador. En aquests camps s'ha portat a terme molta investigació, la qual ha donat com a resultat mètodes que permeten crear imatges per ordinador de forma més flexible, robusta i eficient. Però tot i així, existeixen moltes millores per a portar a terme, degut a que moltes d'aquestes tècniques pateixen problemes de continuïtat que dificulta la posterior aplicació de mètodes d'interpolació. Per lo tant, en aquesta tesis doctoral presentem una sèrie d'algorismes que aporten continuïtat en àrees estratègiques i importants en els gràfics per ordinador.

En el camp del texturat de models 3D proposem un nou algoritme que permet realitzar un mapeig continu de models texturats amb textures multi-pedaç. Aquest tipus de parametritzacions divideixen un model continu en un conjunt de pedaços discontinus en espai de textura, provocant discontinuïtats i com a conseqüència, problemes en aplicacions tan comunes com poden ser el filtrat de textures i les simulacions en espai de textura. El nostre mètode converteix qualsevol parametrització multi-pedaç en una parametrització sense discontinuïtats, gràcies tant a la utilització d'una correspondència entre àrees que es troben fora dels pedaços i àrees que estan dintre, així com a l'ús d'un conjunt de triangles virtuals que literalment *"cusen"* els pedaçoss per a solucionar les diferències existents en el mostreig dels mateixos. *Continuity Mapping* no requereix modificar la textura prèviament generada per l'artista, és completament automàtic i fa un ús eficient dels recursos, ja que requereix poca memòria i té un cost computacional baix.

Per a deformar un model tridimensional i crear així noves postures, proposem un nou mètode de deformació basat en caixes o gàbies. Fins ara, les tècniques de deformació de caixes estaven limitades a la utilització d'una única caixa degut a l'existència de problemes de continuïtat a les fronteres de les mateixes. Com a conseqüència, aquests mètodes no poden deformar localment una regió d'un model i a la vegada, el consum de temps i memòria es veuen incrementats. Per això introduïm *Cages, una tècnica que permet la utilització de múltiples caixes englobant el model a múltiples nivells de detall, per així poder portar a terme deformacions de forma més senzilla i ràpida. El mètode proposat soluciona les discontinuïtats dels mètodes anteriors mitjançant la combinació suau de la deformació de cadascuna de les caixes a la vegada que permet utilitzar conjunts heterogenis de coordenades, donant així més flexibilitat a l'usuari final.

Per a finalitzar, proposem una tècnica d'acceleració de la visualització d'escenes 3D, anomenada *I-Render*, amb l'objectiu d'obtenir un Ray Tracing aproximat, però a la vegada més ràpid i eficient. Primer portem a terme un procés de d'agrupació dels triangles que

formen la malla d'entrada, el qual es basa en la utilització de la Teoria de la Informació per a agrupar triangles amb característiques similars. Aquests conjunts de triangles són els encarregats de definir tant regions amb poques variacions (suaus), així com zones de transicions més brusques (discontinuïtats). A continuació, introduïm un nou algorisme de visualització multi-passada, que utilitza la informació dels grups, prèviament generada, per a poder decidir quines àrees de la imatge final poden ser interpolades i quines necessiten càlculs més costosos. Tot aquest procés es porta a terme, completament, en espai de pantalla i, como a conseqüència, el nostre mètode es pot utilitzar conjuntament a d'altres tècniques d'acceleració més habituals basades en estructures de dades espacials. A més, *I-Render* també suporta models animats.

RESUMEN

En el campo de los gráficos por ordenador es una práctica muy común texturar los modelos 3D para poderles aplicar materiales. Una vez los modelos están texturados, éstos suelen sufrir un proceso de deformación con el objetivo de poder crear nuevas poses que puedan ser más apropiadas para una escena determinada. A continuación, la escena que contiene tales modelos, se visualiza mediante el uso de un algoritmo de renderizado. Así pues, es evidente que el texturado, deformación y la visualización son, aún hoy, partes muy importantes de los gráficos por ordenador. En estos campos se ha llevado a cabo mucha investigación, la cuál ha dado como resultado métodos que permiten crear imágenes por ordenador de forma más flexible, robusta y eficiente. Pero aún existen mejoras por hacer, debido a que muchas de estas técnicas poseen problemas de continuidad que dificulta la posterior aplicación de métodos de interpolación. Por lo tanto, en esta tesis presentamos una serie de algoritmos que aportan continuidad en áreas estratégicas e importantes de los gráficos por ordenador.

En el campo del texturado de modelos 3D proponemos un nuevo algoritmo, llamado *Continuity Mapping*, que permite llevar a cabo un mapping continuo de modelos texturados con texturas multi-chart. Éste tipo de parametrizaciones particionan un modelo continuo en un conjunto de charts discontinuos, en espacio de textura, provocando discontinuidades y, como consecuencia, problemas en aplicaciones tan comunes como el filtrado de texturas y las simulaciones continuas en espacio de textura. Nuestro método convierte cualquier parametrización multi-chart en una parametrización sin discontinuidades gracias tant al uso de una correspondencia entre áreas que están fuera de los charts y áreas que están dentro, así como a la utilización de un conjunto de triángulos virtuales que literalmente *"cosen"* los charts para solucionar las diferencias existentes en el sampleado. *Continuity Mapping* no requiere la modificación de la textura previamente generada por el artista, es completamente automático y hace un uso eficiente de los recursos, ya que consume poca memoria y tiene un coste computacional bajo.

Para deformar un modelo tridimensional y crear así nuevas poses, proponemos un novedoso método de deformación basado en cajas. Hasta ahora, las técnicas de deformación de cajas estaban limitadas al uso de una única caja, debido a la presencia de problemas de continuidad presentes en las fronteras de las mismas. Como consecuencia, estos métodos no pueden deformar, de forma local, una región de un modelo y a su vez, el consumo de tiempo y memoria se ven incrementados. Por ello introducimos *Cages, una técnica que permite el uso de múltiples cajas englobando el modelo a múltiples niveles de detalle, para así poder llevar a cabo deformaciones de forma más sencilla y rápida. El método propuesto soluciona las discontinuidades de los métodos anteriores mediante la combinación suave de la deformación de cada caja y a su vez permitiendo el uso de conjuntos heterogéneos de coordenadas, dando así más flexibilidad al usuario final.

Para finalizar, proponemos una técnica de aceleración de rendering, llamada *I-Render*, con el objetivo de obtener un Ray Tracing aproximado pero a su vez más rápido y eficiente.

Primero realizamos un proceso de clusterizado de la malla de entrada, el cuál se basa en el uso de la Teoría de la Información para agrupar triángulos con características similares. Estos clusters son los encargados de definir, tanto regiones con pocas variaciones (suaves), así como zonas de transiciones bruscas (discontinuidades). A continuación, introducimos un nuevo algoritmo de rendering multi-pasada, que utiliza la información de los clusters para poder decidir qué áreas de la imagen final deben ser interpoladas y cuáles necesitan de cálculos más costosos. Todo este proceso se lleva a cabo, completamente, en espacio de pantalla y, como consecuencia, nuestro método puede ser utilizado en conjunto con técnicas de aceleración clásocas basadas en estructuras de datos espaciales. Además, *I-Render* también soporta modelos animados.

ABSTRACT

In Computer Graphics applications, it is a common practice to texture 3D models to apply material properties to them. Then, once the models are textured, they are deformed to create new poses that can be more appropriate for the needs of a certain scene and finally, those models are visualized with a rendering algorithm. So, it is evident that mesh texturing, mesh deformation and rendering are still key parts of Computer Graphics. In these fields much research has been done, resulting in methods that allow to create a computer-aided images in a more flexible, robust and efficient way. Despite this, there exist improvements to be done, as many of those approaches suffer from continuity problems that dumper interpolation procedures. Thus, in this thesis we present algorithms that address continuity in key areas of Computer Graphics.

In he field of mesh texturing, we introduce a new algorithm, called *Continuity Mapping*, that allows a continuous mapping of multi-chart textures on 3D models. This type of parameterizations break a continuous model into a set of disconnected charts in texture space, making discontinuities appear and causing serious problems for common applications like texture filtering and continuous simulations in texture space. Our approach makes any multi-chart parameterization seamless by the use of a bidirectional mapping between areas outside the charts and areas inside, as well as the usage of a set of virtual triangles that sew the charts for addressing the sampling mismatch produced at chart boundaries. *Continuity Mapping* does not require any modification of the artist-provided textures, it is fully automatic and has small memory and computational costs.

To deform a model and create new poses, we propose a novel cage-based deformation approach. Up to now, cage-based deformation techniques were limited to the usage of single cages because of the presence of continuity problems existing at cage boundaries. As a consequence, they cannot locally deform a region of a model and the time and memory consumption is increased. We introduce `*Cages` a technique which allows the usage of multiple cages enclosing the model, at multiple levels of details for easier and faster mesh deformation. The proposed approach solves the discontinuities of previous approaches by smoothly blending each cage deformation and allowing the usage of heterogeneous sets of coordinates, giving more flexibility to the final user.

Finally, we propose a new rendering acceleration technique, called *I-Render*, for fast and approximate Ray Tracing. First, we perform a pre-processing clustering on the input mesh, that builds upon information theoretic tools to group triangles by their similar features. These clusters define regions of smooth variation, as well as regions of sharp transitions (discontinuities). Then, we introduce a new multi-pass rendering algorithm that uses that information to decide which areas of the final image could be interpolated and which require more involved calculations. All this process is carried out completely in screen space and, as a consequence, our approach can be used in addition to common acceleration spatial data structures. *I-Render* also supports animated models.

1 INTRODUCTION

Todos aquellos logros que nazcan del corazón y tengan la voluntad de alimentarlos con amor, se cumplirán.

The ultimate goal of a Computer Graphics-oriented application is to generate a synthetic image from a given scene by means of a computer. A scene file contains 3D models in a strictly defined language or data structure and it is usually described by geometry, camera, texture, lighting, and shading information. All this data is then passed to a rendering program to be processed and output to a digital image or raster graphics image file. 3D models are usually obtained by a scanning processes or modeled from scratch by modelers, that are aided by many existing powerful sculpting tools that mimic real life actions. So, we can consider 3D models as our primary elements to create a computer-generated image. Once we have obtained our 3D models, it is a common practice to assign them material properties to enhance their visual quality and final shading. Usually, materials are applied to the surface represented by the models through the usage of texture maps (texturing), which are plain images storing per-pixel colors (see the top-left image in Figure 1). This texturing process is carried out by the usage of parameterization techniques [25], which assign to every point on the mesh surface a texture coordinate (position) from the texture map. This method of texturing has got really popular trough the years mainly because parameterization techniques allow the usage of regular textures, which can be stored in a way that suits current graphics processor units (GPU) architecture. Given the usual complexity of these models, they usually cannot be flattened bijectively in texture space and, as a consequence, parameterization techniques tend to break objects into several developable and disconnected patches, called *charts* (see the top-left image in Figure 1). This type of parameterizations are commonly known as *multi-chart parameterizations* [50, 34, 72, 12, 67, 99] although they allow the usage of common texture maps, they introduce a set of cuts over the mesh, known as *seams*, which produce discontinuities in texture domain. These discontinuities bring several problems such as visible rendering artifacts, difficulties for computing continuous simulations in texture space, or even the presence of cracks and holes when displacement mapping techniques are used. As a consequence, other seamless parameterization approaches [6, 90, 47, 96] emerged during the last years, but, even they produce a continuous mapping over the meshes, they faile on displacing multi-chart approaches as the most popular texturing techniques, mainly because of two reasons: First, artists are used to directly paint over the textures and some of those new seamless methods force artists to paint directly over the 3D mesh. Second, most of the models created were textured by the usage of multi-chart parameterization techniques and, as a consequence, if we want to use them a process of texture transfer between the multi-chart and the seamless parameterization would be necessary. This process is not free of artifacts as the source and target parameterizations do not share the same sampling on the texture domain, which produces a lost in the texture

details. Also, seams in the source parameterization are present in the target parameterization as texture artifacts. So, it is evident that a continuous mapping of mutli-chart textures is still an important need for texturing of 3D models and an interesting field of research.

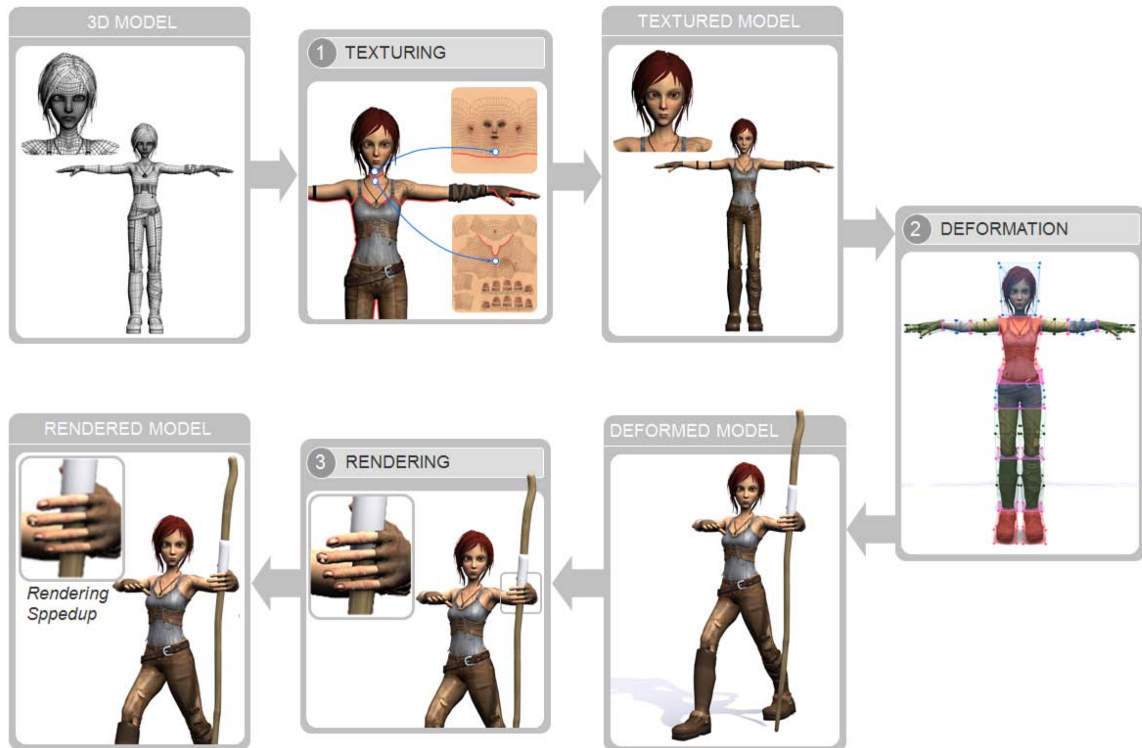


Figure 1: Modeling, deformation and rendering pipeline in a Computer Graphics application.

Once we have a model ready, i.e., modeled and textured, it is a common procedure to modify its initial pose by deforming it and, as a consequence, being able to generate poses that can be more appropriate for the needs of a certain scene (see the bottom-right image in Figure 1). For that purpose, many techniques have been presented in the literature that allow not only a fine degree of control over the deformations, but also a high level of flexibility and usability for the final user [16]. Recently, cage-based deformation techniques have gained a lo of interest [39, 54, 38, 53]. These techniques are characterized using a polyhedron called *cage* to deform an enclosed mesh, which is in fact the model to be modified. This cage is usually of similar shape but simpler than the mesh to be deformed. Those cage-based methods define coordinates for every mesh vertex of the enclosed object with respect to the vertices of the cage. This binding allows to transfer any deformation applied on the cage to the mesh in a fast and flexible way. Several coordinate types exist, each of them providing different deformation results with different properties over the final deformed mesh. However, these cage-based methods are not perfect as they still have some weak points: The usage of a single cage produces deformations that are not local to the region of interest. Instead, a deformation over a region of a cage has impact over the whole mesh. That increases the number of evaluations as well as the required computational costs. Even more, when deforming a model the user has to decide which coordinates to use without being able to combine the benefits of each of them, when needed. So, it could seem a natural procedure for a modeler to have a multi-cage system, instead of a

single cage one, and use them in a hierarchical framework to have a finer degree of control over the final deformation. Moreover, the choice and combination of different coordinates would improve the final results and benefit the modeler as a way to obtain more flexible results with no extra effort. It is clear that all of this properties would improve the features of any deformation package, but problems appear when many cages are used in combination with current cage-based techniques: They produce important discontinuities at cage boundaries and, as a consequence, non-smooth deformations and even cracks and holes will appear in the resulting poses. As a result, modern deformation pipelines still lack the possibility to perform smooth and continuous deformations using a multi-cage and multi-level system. This would increase the productivity of modelers because of the new features: locality of the deformation and performance improvement, as well as a reduction in the computational resources. However, a continuous solution for multi-cage based mesh deformation has still to be proposed.

Finally, the last step is to render the textured and deformed model or scene with a rendering algorithm to generate the final image (see the bottom-left image in Figure 1). Many rendering techniques have been researched and proposed, and software used for rendering may employ a number of these techniques. Two of the most used and well known rendering techniques over the years have been *rasterization* and *Ray Tracing*. Rasterization geometrically projects objects in the scene to an image plane (screen space) converting them into a raster format. This rendering approach can be considered the most popoular technique for real-time 3D computer graphics, because it is cheaper enough to compute for time-demanding applications and most of its stages are currently implemented in GPU's. Instead, Ray Tracing processes the geometry pixel by pixel casting rays from the point of view. When an object is intersected, the color value at the point is evaluated using an illumination model which can take into account, for instance, the color value from a texture-map. Even that its theoretical cost can be considered similar, Ray Tracing is much more expensive to compute than rasterization simply because the lack of standardized specific graphics hardware support [76, 75, 94]. Most methods to improve the performance of Ray Tracing relay in the usage of spatial data structures in object space, as well as in the parallelism capability of modern GPU's [101, 46, 30]. But, for being able to use Ray Tracing in a more interactive environment more efforts need to be done. Thus, Ray Tracing can be still considered an open area of research. It would be interesting to propose alternative solutions that would work with any of the previous object-space acceleration data structures. We can observe that, in every single scene used in production, usually there are many regions that can be considered homogenous or similar, and also other that represent an abrupt change in a certain feature producing, this way, areas of noticeable discontinuities. In contrast to previous discontinuities in the texture and mesh deformation domains, we could take advantage of them as a way to highlight pixels representing places where we could avoid costly computations and, for instance, interpolate values coming from homogenous regions to cheaply generate new pixel values in between. All of these possibilities, represent not only an exciting, but also a difficult challenge, as image quality must remain the same as if we were fully evaluating the visualization.

In conclusion, one can observe that continuity and interpolation problems are still present in many areas of Computer Graphics and in different spaces (e.g., texture space, object space and screen space), and they need to be addressed to simplify and improve

the quality and creation of computer-generated content.

1.1 CONTRIBUTIONS

In this thesis we present several new techniques that represent an improvement over the existing state of the art related to continuity and interpolation techniques in texture space, object space and screen sapce. Each of these techniques solve existing continuity and interpolation problems in each of the stages of the previously presented pipeline (see Figure 1): texturing, mesh deformation and finally rendering. Following the same pipeline, we can summarize our contributions in three main areas:

- **Mesh texturing**. In this field, we propose an approach that solves the continuity problems that commonly appear in all the models that have been textured by using a multi-chart based parameterization. Up to now, no method was focused on providing a continuous mapping over these types of textured models. Thus, we present and approach called *Continuity Mapping* that produces a continuous mapping in boundaries between charts, which are the places where discontinuities are located. This continuous mapping is achieved by a set of virtual triangles that sew discontinuous charts into continuous regions in texture space. The proposed method allows for several applications that where not possible before with muclti-chart textured meshes, like seamless texture mapping, continuous simulations in texture space, and multi-chart relief mapping, among others. As it works completely in texture space, it is fully compatible with any mesh deformation technique. Moreover, *Continuity Mapping* achieves continuity with small memory and computational resources. The results presented in this work have been published in

  > ***Continuity mapping for multi-chart textures***.
  > González, Francisco and Patow, Gustavo.
  > SIGGRAPH Asia '09, Yokohama, Japan.
  > ACM Transactions on Graphics, Volume 28, issue 5, pages 109:1–109:8.
  > December 2009. ISSN 0730-0301. Article 109.
  > doi: http://doi.acm.org/10.1145/1618452.1618455.
  > URL http://doi.acm.org/10.1145/1618452.1618455.
  > Publisher ACM. New York, USA.

- **Mesh deformation**. A new cage-based approach for mesh deformation called `*Cages` is presented. In contrast to previous approaches, it allows the usage of many cages into multiple levels of details (hierarchical) to deform an enclosed mesh. This is an important feature as it allows to produce not only more localized deformations, but also consumes less memory resources and performs fewer evaluations. To use many cages instead of a single one and, as a consequence, be able to obtain all the commented benefits, we solve the discontinuities that appear in boundaries between cages, which produces non-smooth deformations. We blend the associated deformation at cage boundaries with each cage own deformation, thus producing continuous and smooth results. `*Cages` also allows the usage of any of the current cage coordinates in any of the cages, giving the user the freedom to change the deformation

results with the same cage configuration. This research has been accepted for publication in

> ***Cages: A multi-level, multi-cage based system for mesh deformation.**
> González, Francisco; Paradinas, Teresa; Coll, Narcís; and Patow, Gustavo.
> ACM Transactions on Graphics.
> *Accepted for publication*.

- **Rendering**. We present a rendering technique called *I-Render*, that *introduces* discontinuities and takes profit of them at rendering time to improve the visualization performance. First, we propose an information-theoretic framework for mesh clustering that groups triangles of a scene by their degree of similarity with respect to a user pre-defined set of features. Then, we present a new multi-resolution/multi-pass rendering algorithm. This algorithm computes an initial image at a low resolution (which is less expensive) and then, successively increase the image size using previously computed clusters to avoid costly computations in places where discontinuities do not exist (similar or homogeneous regions). As a consequence, interpolation from previous low-resolution images can be performed while guaranteing image quality. We show that our technique can be used for animated characters as well as for static scenes achieving an improvement up to 8 times in the rendering performance while keeping memory consumption low. This work has been submitted and is currently under review:

> ***I-Render: Approximate interpolated rendering by mesh clustering***.
> González, Francisco; García, Ismael and Patow, Gustavo.
> *Submitted*.

1.2 THESIS OVERVIEW

The content of the thesis is organized as follows:

- **Chapter 2: Previous Work**. In the second chapter we are going to introduce the concept of parameterization and continuity related to Computer Graphics in several spaces: in texture space (2D), in object space (3D) as well as in screen space. We will also establish the basics to understand the continuity problems that appear when we parameterize a 3D model in each of those spaces, and how we can solve or even take profit of them. Then, we will review the notion behind interpolation and the most important types. We will also comment the current state of the art in continuity and interpolation in Computer Graphics in 2D, 3D and screen space. We will introduce basic concepts about information theory and its application to Computer Graphics and finally, we will explain the graphics hardware pipeline including all its programmable stages.
- **Chapter 3: Continuity and interpolation in texture space**. In this chapter we are going to introduce a technique called *Continuity Mapping* that will allow to solve

discontinuities in the texture space domain and, as a consequence, we will be able to texture a 3D model without any kind of discontinuity.

- **Chapter 4: Continuity and interpolation in object space**. In the fourth chapter we will face the problem of continuity and interpolation in object space, more precisely, in the field of cage-based mesh deformation. We will introduce a technique called *Cages that will allow to use a more general and flexible approach for cage-based mesh deformation: it will use many cages in a multi-level setup, performing much more localized deformations with less computational and memory resources while preserving the continuity (smoothness) of the mesh trough the set of cages.
- **Chapter 5: Continuity and interpolation in screen space**. In this fifth chapter we are going to propose a technique called *I-Render*, which explicitly introduces discontinuities as a mechanism to detect places in a model where large changes in a set of user-predefined features are located. Then, in screen space, we will use discontinuity projections to improve the speed of the visualization by taking advantage of the low-variability parts in which interpolation of low-resolution information can be performed, without significant impact in the final image quality.
- **Chapter 6: Conclusions and future work**. In this final chapter we are going to present the main conclusions that we have obtained along the whole document, we will highlight all the contributions of this thesis into the field of continuity and interpolation in Computer Graphics and finally, we will talk about future work that could be proposed from the methods presented in this thesis.

2 PREVIOUS WORK

Los obstáculos os ayudan a ser más fuertes, a definir vuestras propias ilusiones.

In this chapter we are going to present some important concepts that will be used throughout the whole document, as well as some techniques involving continuity and interpolation methods for Computer Graphics. We first introduce a generic framework for parameterization and continuity on surfaces in texture sapce (2D), object space (3D) and screen space, as a way to understand the problems and challenges that we need to solve. Then, we introduce the concept of interpolation and we explain several approaches for it. After that, we explain the state of the art for continuity and interpolation in the previously commented spaces. Also, some basic concepts about mesh clustering and Information Theory and how they have been applied to Computer Graphics are explained. Finally, we introduce the graphics hardware pipeline and all its programmable stages.

2.1 PARAMETERIZATION AND CONTINUITY

A mapping or parameterization of a surface can be viewed as a one-to-one correspondence from the surface to a suitable domain [25]. In general, as the mapping domain itself is a surface, to construct a parameterization means to map one surface into another one. Usually surfaces are represented (or approximated) by triangular meshes and in this case the mappings are piecewise linear. Surface parameterization was first introduced in Computer Graphics as a way to map textures into surfaces, but it has been also successfully and widely used in other areas: repair of CAD models, mesh editing and compression, remeshing, surface fitting, detail mapping, etc. In the following subsections we are going to describe parameterizations in several different spaces and we will explain the discontinuities they introduce in the mapping function, which will be later solved by the techniques presented in this thesis.

2.1.1 *Texture space*

Given a surface $S \subset \mathbb{R}^3$ and a point $p \in S$, let's define a domain $T \subset \mathbb{R}^2$ and a parameterization function $m_T : S \to T$ such that $m_T(p) = t$, $t \in T$ (see Figure 2). As our surfaces are represented by triangular meshes, the objective of m_T is to map each polygon of the mesh in S into a polygon of the two-dimensional space T. Usually, a good parameterization m_T should be one that transforms each triangle of the mesh without any kind of distortion from S to T. So, depending on the measure that m_T minimizes, we can classify parameterizations by:

- **Conformal parameterizations**. A parameterization m_T is conformal or angle preserving if the angle of intersection between every pair of edges of each triangle of a mesh is the same in S than in T. Thus, a conformal parameterization m_T will try to minimize the distortion produced between the angles of each pair of edges when they are mapped to T.

- **Equiareal parameterizations**. A parameterization m_T can be considered equiareal if every triangle of the mesh is mapped onto T with the same area. So, an equiareal parameterization m_T will have as its main objective the minimization of the distortion produced between the areas of the triangles in S when they are mapped to T.

- **Isometric parameterizations**. We can define a parameterization or mapping m_T of a triangular surface as isometric when it preserves both the angle of the triangles and their areas. We can consider isometric mappings as ideal as they preserve everything we could ask for: areas, angles, and, as a consequence, lengths. However, isometric mappings only exist for very special cases. So, what many approaches try to do is to find a parameterization which either is conformal, or is equiareal or minimizes some combination of angle and area distortion.

Let us define a function $a : S \to \mathbb{R}^n$ that allows to retrieve all the attributes associated to a surface point. So, $a(p) =< \mathtt{attribute}_1, \mathtt{attribute}_2, ..., \mathtt{attribute}_n >$ is a n-tuple storing all the attributes of point p (e.g., the normal, the texture coordinates, the color, etc.). In practice, in Computer Graphics the domain T is usually discretized in a grid of cells $G \subset \mathbb{R}^2$ by a function $m_G : T \to G$, such that, for each parameterized point $t \in T$ that

falls in a given cell $g \in G$, we store a single representative value $a(m_T^{-1}(t))$ in g. This new space G is usually known as texture space (see Figure 2).

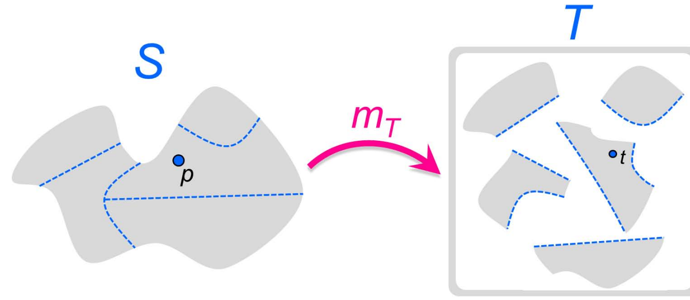


Figure 2: A possible parameterization (multi-chart) of a surface into texture space.

When performing a parameterization operation onto a generic object, it is a common practice to partition the object surface S into multiple *charts* C_i, verifying that $S = \bigcup C_i$ and $C_i \bigcap C_j = \emptyset$ for $i \neq j$. When dividing the mesh surface S into charts, *seams* are introduced, which are cuts applied over S. Then, these charts are mapped by the parameterization function m_T into disjoint parts and, as a consequence, they are mapped into disjoint parts in G by m_G, too. Unwrapping an entire mesh as a single chart can create parametrizations with large distortion and less uniform sampling than what can be achieved with multiple local charts, particularly for surfaces of high genus. Parameterizations that create multiple charts are called *multi-chart parameterizations*.

For mutli-chart parameterizations, it may happen that discontinuities are introduced at the boundaries between charts, where the seams are located. For these parameterizations we can distinguish two type of discontinuities:

- **Spatial discontinuities**. This type of discontinuities appears because neighboring triangles in S (in 3D) are not neighbors in T (in 2D) and, as a consequence, in G. This way, a continuous and smooth mesh S, is transformed by m_T (and m_G) into a discontinuous picewise mesh in T (and G). As an example, in Figure 3 we can observe two neighboring charts C_i and C_j in S. Those charts are parameterized by m_T into disconnected regions in T.

- **Sampling discontinuities**. Sampling discontinuities appear because different charts that are neighbors in S (in 3D) are mapped with different distortion into T by m_T, and as a consequence their sampling in G produced by m_G is different, too. This produces that cells from G do not match up at chart boundaries when back projected to S. As an example, in Figure 3 we show two charts C_i and C_j, the initial and ending points of the boundary between them p_0 and p_1, and a point p lying on the boundary between them. A color attribute is associated for points p_0 (green) and p_1 (red). In a continuous and smooth surface, the color gradation through a chart boundary should look like the one shown in left image of Figure 3. Given the fact that the mapping produced by m_T may differ for triangles of charts C_i and C_j, it can happen that the sampling of the color gradation in T differs, too (see right image in Figure 3). Thus, when the texture is mapped on S, both the cells and the colors stored in them will not match at both sides of the boundary between the charts.

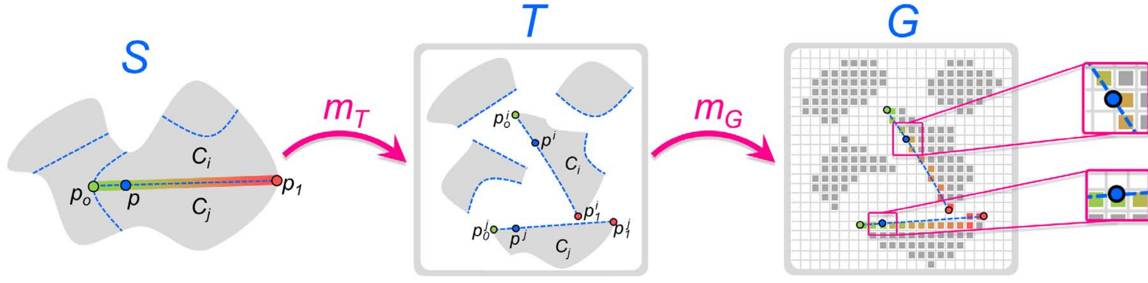


Figure 3: Multi-chart parameterization discontinuities.

2.1.2 *Object space*

The previously defined 2D parameterizations transform a surface $S \subset \mathbb{R}^3$ to a small domain $T \subset \mathbb{R}^2$, where we can easily operate. Now, let us define a parameterization or mapping function m that does not reduce the degree of the output domain but instead represents every point p from S with respect to another mesh C that encloses S. The mesh C is picewise defined by a set of connected submeshes C_i, such that $C = \bigcup C_i$ and $C_i \bigcap C_j = \emptyset$ for $i \neq j$. C is characterized by the fact that it usually has a similar, but simpler shape than S. This is an important point, as by using m we will be able to perform any operation in C and then transfer it to S. As a consequence, we can reduce the complexity of the involved calculations. For that purpose, we first define a function $w : S \times V(C_i) \rightarrow \mathbb{R}^n$ that binds each surface point p with the set of vertices of its enclosing submesh $V(C_i)$ by a set of weights. As we will show in the following sections, there are several ways to define such a binding through the usage of coordinates (see Section 2.3.2). Each type of coordinate has its own properties, but all of them have in common two key features: They are completely defined and smooth (C^∞) inside a cage C_i, but on the contrary they have continuity problems at cage boundaries (C^0 or even discontinuities).

Now, let's specify the parameterization function $m : \mathbb{R}^n \times V(C_i) \rightarrow \mathbb{R}^3$ where a point can be represented by $p = m(w(p, V(C_i)), V(C_i))$. That way, when C is not deformed, the surface S is smooth and continuous using the mapping function m. However, if we apply a deformation to the mesh C, which is transformed into C', transforming each point of the surface by $p' = m(w(p, V(C_i)), V(C_i{}'))$, discontinuities appear in regions near the cages boundaries, producing non-smooth deformations of S. That is, as we get closer to a boundary between two deformed controlling submeshes C'_i and C'_j from C', discontinuities may appear when transferring the deformation to the enclosed surface when the function m is used. As a consequence, all the attributes associated to the enclosed surface could also be discontinuous (see right image in Figure 4).

2.1.3 *Screen space*

Usually, after we have a triangular mesh positioned in 3D space, it is common to visualize it. For that propose, the users define, among other, scene properties, the lighting settings, the cameras to render the model from a given viewpoint, etc. To display the resulting image rendered from the camera point of view, the scene experiences a set of transforma-

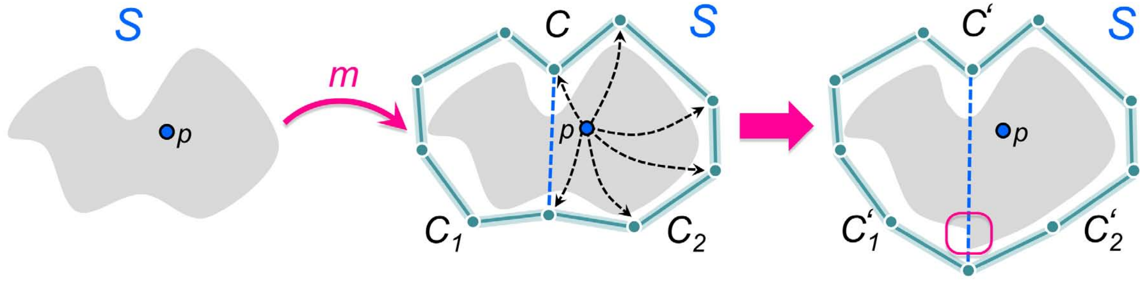


Figure 4: Parameterization of a surface mesh into object space by the use of another additional surface C.

tions that map each point p of a surface in S to the screen by a mapping function m_C (see Figure 5). The transformed model, once it is mapped to the screen, lies in a space commonly known as *screen space*. That space is merely a discretization of the image projected by the camera to be displayed into a physical device.

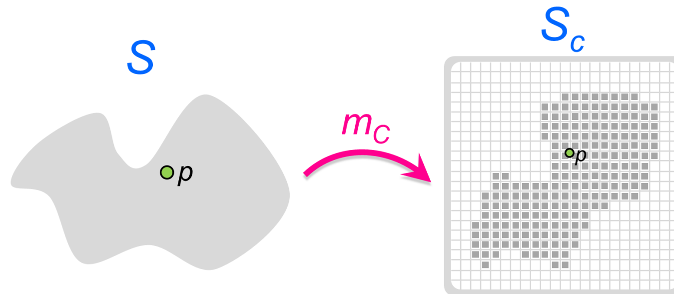


Figure 5: Projection of a surface mesh into screen space.

As we have previously commented, given a point p, we can retrieve all its attributes with $a(p)$. Even if a is a continuous smooth function, it may happen that, when projecting $a(p)$ to screen space S_C by the mapping function $m_C(a(p))$, discontinuities appear as a consequence of the sampling performed to represent the model in the image. This can happen because our sampling rate is not enough to represent the signal defined by the projection of the 3D model. This is simply a consequence of the *Nyquist sampling theorem*, which states that a given bandlimited analog signal can be perfectly reconstructed from an infinite sequence of samples if the sampling rate exceeds 2B samples, where B is the highest frequency of the original signal. An example of the problems that appear due to the incorrect sampling is the loss of detail when projecting in screen space sharp features of a model. On the contrary, a side effect of sampling is that smooth areas of $a(p)$ can be over-sampled and, as a consequence, we are unnecessarily increasing the computational time of the image generation.

Thus, let us consider a partition of a mesh m_S in 3D space by grouping nearby points that have similar features or attributes. Each group will be called *cluster*. So, given two points $p_1 \in S$ and $p_2 \in S$, they will belong to the same cluster if they are similar enough, that is, if the difference between their features is smaller than a given threshold $|a(p_1) - a(p_2)| \leqslant Th$.

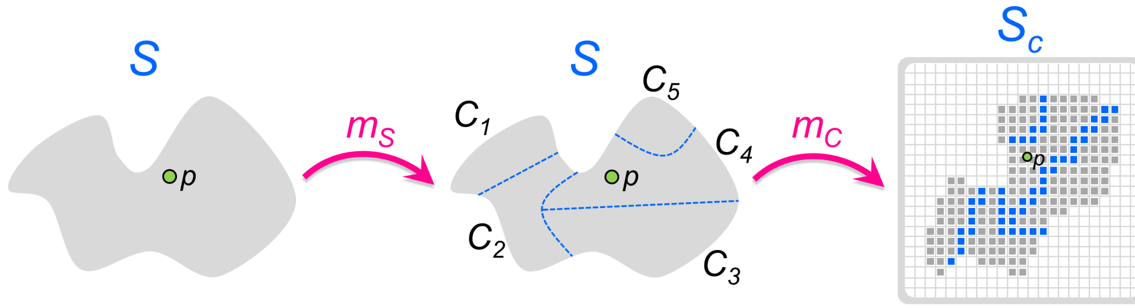


Figure 6: Discontinuities of a surface mesh projected into screen space.

Finally, consider two neighboring clusters C_i and C_j. For every point p lying in the boundary of C_i and C_j the features or attributes of p given by $a(p)$ could represent an abrupt change. As we already said, this is because two main reasons: the high frequencies that can exist in the signal of any of features of the mesh and the discretization performed to project it on the screen. When this happens, a discontinuity or an area of non-homogeneity can be considered there. So, if we project the value $a(p)$ in screen space by $m_C(a(p))$, the discontinuity detected in S will still be reflected in screen space S_C. This way, we are able to easily introduce information about places where discontinuities exist with the objective to perform a better and faster reconstruction of the final image in our physical device.

2.2 INTERPOLATION

In the mathematical field of numerical analysis, interpolation is a method of constructing new data points within the range of a discrete set of known data points. In engineering and science, users often have a number of data points, obtained by sampling or experimentation, which represent the values of a function for a limited number of values of the independent variable. It is often required to interpolate (i.e. estimate) the value of that function for an intermediate value of the independent variable. This may be achieved by curve fitting or regression analysis. Similarly, in Computer Graphics the concept of interpolation is defined as the creation of new values that lay between a set of other known values. These known values usually represent a smooth signal, and as a consequence the new created values represent an smooth signal too (although some detail can be lost). Nobody can expect a smooth interpolation if the initial values contain some sort of discontinuities within the functions. As an example of interpolation, when a triangle is rasterized into a two-dimensional image from its vertices, all the pixels between those vertices are filled in by an interpolation algorithm, which determines their attributes (color, normal, texture coordinates, etc). Another example of interpolation happens when an image generated in a videogame is created in sub-HD resolution and then it is upscaled to be displayed in a monitor that supports full HD mode. There, the missing information is, again, obtained by interpolation.

When performing an interpolation over a set of values, there are a number of interpolation functions that one can use. The following interpolation approaches are the most

used in Computer Graphics, as they are easy to compute, stable and most of them are implemented in current graphics hardware. Let us review the types of interpolation:

- **Linear interpolation.** Linear interpolation (see Figure 7) is the simplest and fastest method of interpolation. Given two known points x_0 and x_1 in a one dimensional space, the linear interpolant is the straight line that joins both points. For a point p that lies in the interval $[x_0, x_1]$, we can define mathematically the term linear interpolation as $(1 - \lambda) * x_0 + \lambda * x_1$, with $\lambda \in [0, 1]$. When λ is equal to 0, $p = x_0$ and when λ is equal to 1, then $p = x_1$. So, this can be understood as the weighted average of p from x_0 and x_1. Values of λ outside this range result in extrapolation. Let us note that linear interpolation produces discontinuities at points x_0 and x_1, i.e., when points x_1 and x_2 are not in the same line of x_0 and x_1.

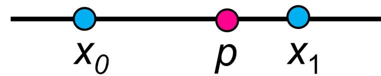


Figure 7: Lienar interpolation.

- **Bilinear interpolation.** In mathematics, bilinear interpolation (see Figure 8) is considered an extension of the linear interpolation for functions of two variables (X and Y) on a regular 2D grid. So, we could consider bilinear interpolation as a linear interpolation of the first variable X followed by a linear interpolation of the second variable Y. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location. In computer vision and image processing, bilinear interpolation is one of the basic resampling techniques, while in texture mapping it is also known as bilinear texture mapping. Bilinear interpolation considers the closest 2x2 neighborhood of known point values surrounding the unknown point's location. It then takes a weighted average of these 4 samples to arrive at its final, interpolated value. The weight on each of the 4 point values is based on the computed point's distance (in 2D space) from each of the known points. So, suppose that we want to find a value of a function f at a point $p = (x, y)$. It is assumed that we know the value of f at the four points (x_0, y_0), (x_0, y_1), (x_1, y_0) and (x_1, y_1). We first perform a linear interpolation in the x-direction and then we proceed by interpolation in the y-direction:

$$
\begin{aligned}
f(x, y) = &1/((x_1 - x_0)(y_1 - y_0)) \\
&(f(x_0, y_0)(x_1 - x)(y_1 - y) + f(x_1, y_0)(x - x_0)(y_1 - y) + \\
&f(x_0, y_1)(x_1 - x)(y - y_0) + f(x_1, y_1)(x - x_0)(y - y_0))
\end{aligned}
\tag{1}
$$

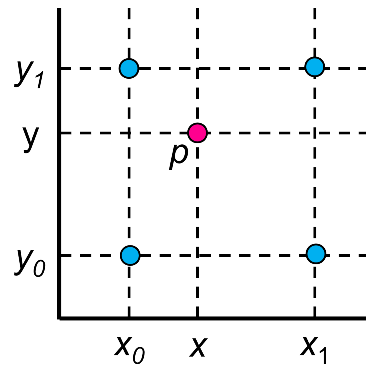


Figure 8: Bilinear interpolation.

- **Trilinear interpolation**. As bilinear interpolation is the extension of linear interpolation to 2D spaces (e.g, texture space), we can refer to trilineal interpolation (see Figure 9) as the extension of linear interpolation to 3D space. Trilinear requires 8 adjacent pre-defined values surrounding the interpolation point, so it is the most expensive interpolation mode. Suppose we have a periodic cubic lattice with spacing 1, all we have to do is follow the same procedure as before but with one more dimension added. So, first of all we interpolate along the x axis, then we interpolate the values obtained along y axis, and finally we perform the same along the z axis. Each interpolation can be seen as we were pushing the corresponding face of the cube each time and perform a linear interpolation.

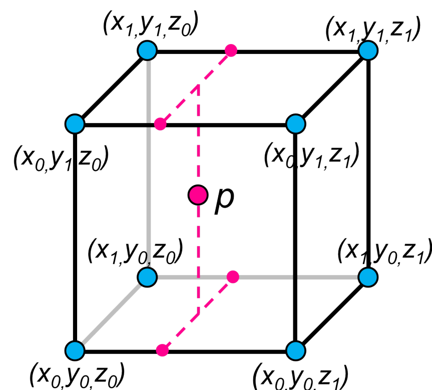


Figure 9: Trilinear interpolation.

2.3 CONTINUITY AND INTERPOLATION TECHNIQUES

In this section we are going to present some of the most well-known techniques related to continuity and interpolation in texture space, object sapce and screen space.

2.3.1 *Texture space*

Continuity and interpolation in texture space domain is strongly related to texturing parameterization [35], as it intends to solve the problem of continuity in texture space [25]. In

order to achieve this, several approaches have been proposed. Some of these approaches rely on 2D data structures for texturing purposes, while others use 3D data structures:

- **Parameterization techniques with 2D data structures**. Levy et al. [50] presented a multi-chart parameterization technique with a new quasi-conformal mapping, based on a least-squares approximation of the Cauchy-Riemann equations, as well as a new packing method for the generated charts. Geometry Images [34] unwrap an entire mesh into a single chart, creating parameterizations with greater distortion and less uniform sampling than can be achieved with multiple local charts, particularly for surfaces of high genus. Carr et al. [12] and Sander et al. [72] present extensions to parameterize them into multiple charts, and Purnomo et al. [67] describe a new type of seamless quadrilateral chart-based atlas. There, neighboring chart's texels were copied into a one-pixel gutter on the boundary of the original chart. This work was the extension of the one by Carr et al. [11], where mip-mappable atlases were created with one chart per triangle, also using a gutter to sample across seams. Finnaly, Zhou et al. [99] presented a fully automatic method to create texture atlases on arbitrary meshes was presented. Seams are still present for creating a chartification of the mesh, but it allows the user to balance the number of charts against the resulting stretch (see Figure 10).

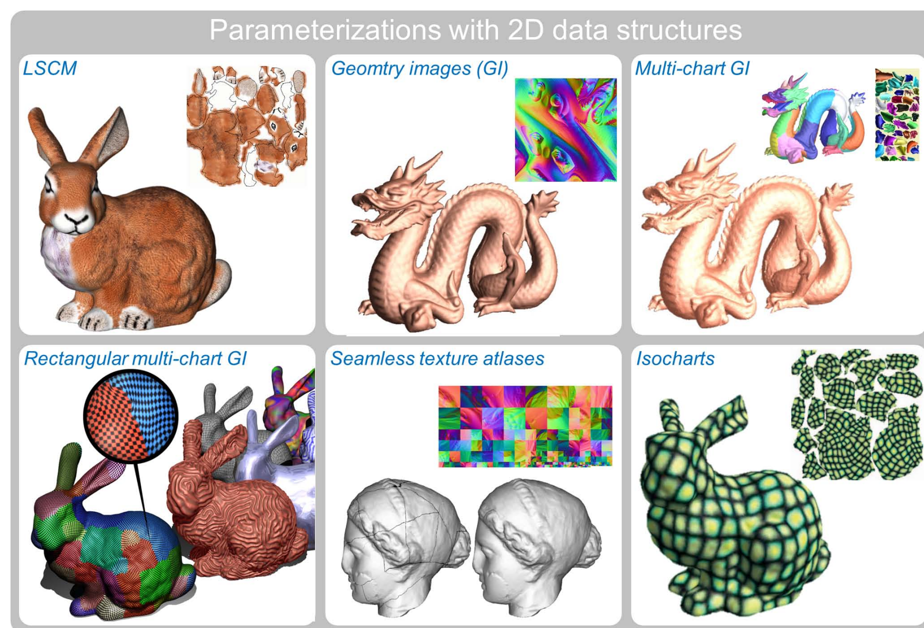


Figure 10: Mesh parameterization techniques using 2D data structures with one single or several charts. Images from [50], [34], [72], [12], [67] and [99].

- **Parameterization techniques with 3D data structures**. Benson et.al [6] presented in 2002 a technique called *Octree Textures* showing how a 3D hierarchical data structure could be used to efficiently store color information along a mesh surface without texture coordinates. Several applications as surface painting and simulations were shown. However, it had several drawbacks: First, any filtering method needs to be implemented by fragment shaders, which increases the cost of the final render, mainly because the access to the data structure is more costly than classic 2D parameterizations, as GPU (Graphics Processor Unit) are extremely efficient at displaying

filtered standard 2D textures. Second, it produces a memory overhead in contrast to common 2D texture maps and, as a consequence, the resolution of the octrees is lower, producing lower quality renderings. Tarini et al. [90] proposed a seamless parameterization of a mesh by using the surface of a *polycube*, whose shape is rawly similar to that of the given model, as texture domain. Even if the parameterization is seamless by construction, the user has to build the polycube manually. Once it is generated, if the geometry or topology of the model is too complex, the technique can have problems handling it, as it increases the complexity of the polycube too, and as a consequence the memory consumption. Moreover, if the polycube is too simple so that it does not capture all the features of the model, visible rendering artifacts may appear. Lefebvre et al. [47] presented another 3D-like parameterization called *Tiletrees* with the objective of seamless texturing of a surface without wasted space (empty regions between charts) and adaptive resolution. This is done by storing square texture tiles into the leaves of an octree surrounding the 3D model. Then, at rendering time the surface is projected into the tiles, and the color is retrieve by common 2D texture fetches. Mesh Colors [96] is a technique that stores colors on vertices defined over the mesh, with the parameterization defined *directly* by the mesh itself. Even all these techniques are seamless, they are not free of problems: Usually artist paint 3D models with mutli-chart parameterizations. So, if we want to use some of these 3D parameterization approaches, a texture transfer process must be applied between the 3D method and the atlas and, as a consequence, some blurring and loss of detail may appear. This will happen no matter how many samples are used for the evaluation during the process. Even more, seams in the original textures will be transferred as artifacts to the newer ones, producing an incorrect visualization of the models.

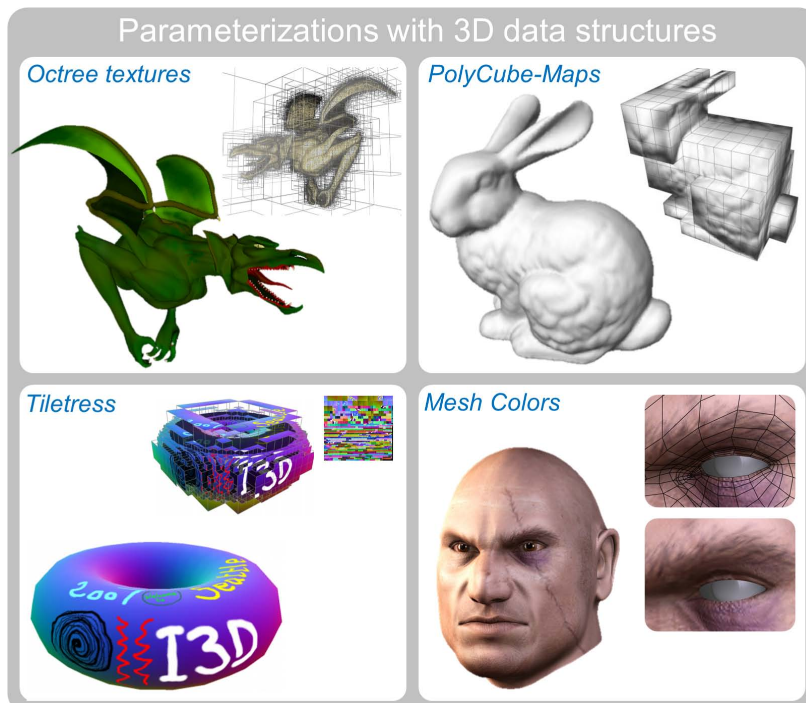


Figure 11: Mesh parameterization techniques using 3D data structures. Images from [6], [90], [47] and [96].

There exist some methods (see Figure 12) that focus on where to place seams to make them less visible: Sheffer and Hart [80] present a method that finds places to put seams, without eliminating them. Sander et al. [72] use an atlas to map the piecewise surface onto charts of arbitrary shape and average values to reduce seam visibility. Also, Kraevoy et al. [43] and Zhou et al. [100] present methods that employ parameterization constraints to hide seams. Castano [13] suggested using patch ownership to assign consistent displacement along seams for subdivision surfaces. Due to the different scaling and orientation of the charts, this still produces seams, although they are less noticeable. More recently, Ray et al. [68], proposed a solution to make multi-chart parameterization seams invisibles, while still outputting a standard texture atlas. They use global parameterization to produce a set of texture coordinates that align the texel grid across boundaries introducing, at the same time, some constrains in the color information of the boundaries.

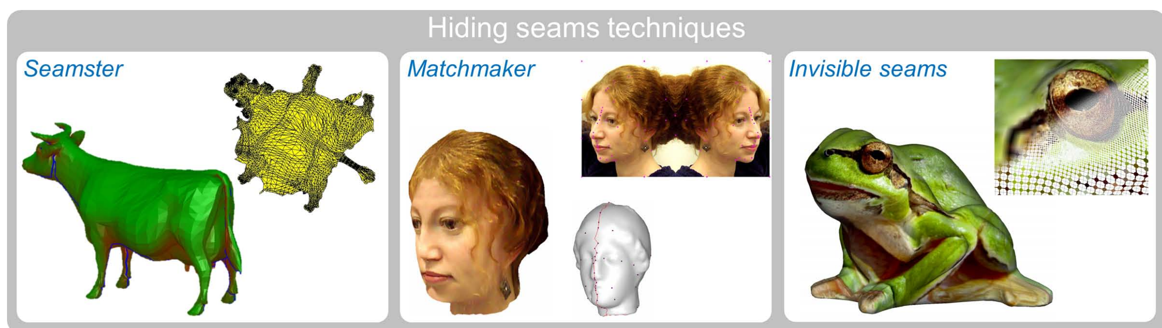


Figure 12: Techniques to hide seams by its placement on the 3D mdoel. Images from [80], [43] and [68].

Other techniques (see Figure 13) use some type of blending approach to dissimulate the seams: In lapped textures [66], the perceptibility of seams is reduced by applying alpha-blending at the edges of the pasted texture patches. Losasso and Hoppe [55] used textures as a height field and blended heights between mip-map levels. In a similar approach, presented by De Toledo et al. [21], neighboring charts must share boundaries with each other, resulting in some overlapping between them. The overlapping area is necessary to avoid cracks in rendering time, and depth information is correctly generated to achieve a seamless reconstruction. However, some undesirable artifacts appear on regions with strong curvature, and self-shadows are very difficult to compute. Even seams can be alleviated in some form, none of these techniques completely eliminates them from the parameterizations.

Lefebvre and Hoppe [49] presented a method to solve the spatial discontinuities caused by multi-chart parameterization to synthesize a texture over a discontinuous atlas. Their technique can only be used for simple simulations in texture space and do not provide any solution for the seam visibility problem in 3D space (sampling discontinuities). Their method work in a similar way to the atlas transition functions defined by Grimm and Hughes [33], but they differ in that they are defined in the surrounding area outside the charts.

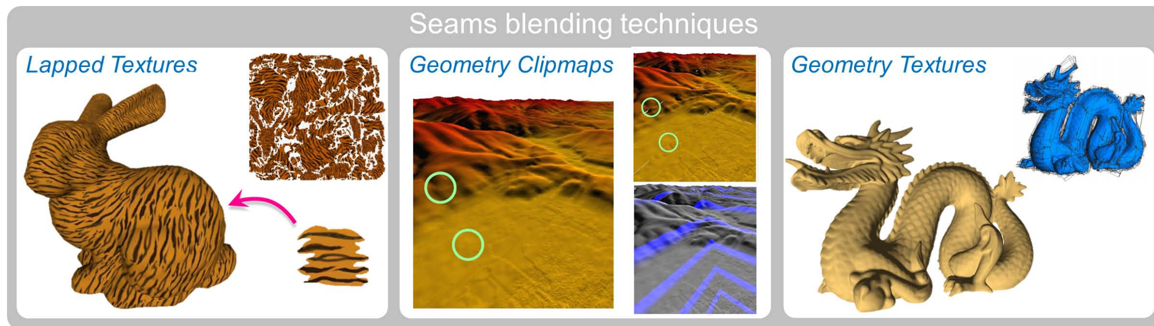


Figure 13: Techniques to hide seams by blending techniques. Images from [66], [55] and [21].

2.3.2 *Object space*

Since 2004, many research effort has been made in Computer Graphics for the deformation and manipulation of triangular meshes. In mesh deformation we can distinguish between two main groups depending on how they apply the deformation on the surface:

- **Surface deformation techniques**. Surface deformation methods modify the surface of the mesh directly and, as a consequence, they can preserve de details and shape of the mesh quite easily. In contrast, these approaches usually need more computational resources.
- **Space deformation techniques**. In comparison, space deformation techniques have received much less attention than surface deformation methods. Here the deformation is applied over a volume or some defined space. The basic space deformation technique defines a lattice with a rather small number of control points that encloses the subject model to be deformed. Then, by manipulating the control points, a smooth deformation is induced. The main advantages of these techniques over the surface methods are their simplicity and speed. However, these methods are oblivious to the surface representation and free of discretization errors.

In recent years, cage-based deformation methods have gained interest and are considered one of the most important mesh deformation approaches. They are characterized by the use of a cage to drive the deformation of an enclosed model (see Figure 14). A cage is a rather low polygon-count polyhedron, which typically has a similar topology and geometry as the enclosed object. The first method based on three dimensional regular lattices was introduced by Sederberg and Parry [77]. Later, this method was extended to handle general lattices [18] and LOD management [78]. In recent years, new deformation methods have been proposed based on the use of coordinates computed with respect to the vertices of a single enclosing cage. Floater and co-workers [24] [25] [39] introduced Mean Value Coordinates (MVC) (see first image in Figure 15) as a method for constructing an interpolant for closed triangular meshes with a closed-form formulation, which is able to reproduce linear functions. MVC are well defined both inside and outside the control mesh (C^∞ continuous) but they are only C^0 continuous across the cage faces.

Later, Joshi et al. [38] proposed Harmonic Coordinates (HC) (see second image in Figure 15) for character articulation, which are positive and C^∞ continuous inside the cage, C^0 continuous on the boundary and have no definition outside the cage. In contrast to MVC, HC guarantee to be positive everywhere in the cage interior, while its influence decreases

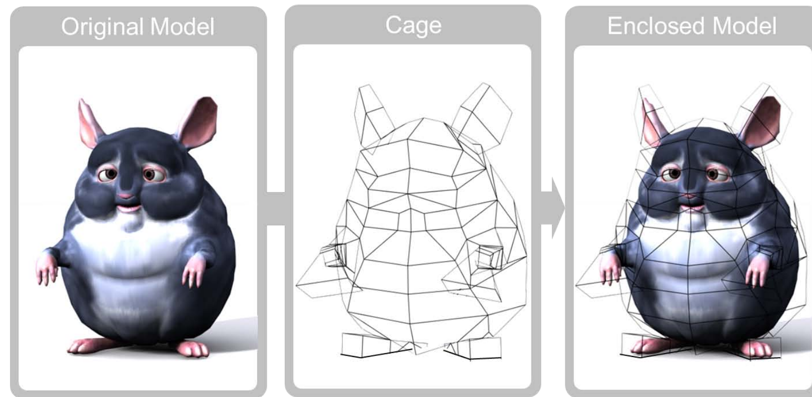


Figure 14: Cage-based deformation methods use a cage to drive the deformation of a model.

with distance as measured within the control mesh. However, as they do not have an explicit expression, they force the usage of a multi-grid finite difference to compute the coordinates. Lipman et al. [53] presented an alternative non-negative coordinate definition to MVC (PMVC) (see third image in Figure 15). The coordinates are computed numerically by using a GPU-friendly approach.

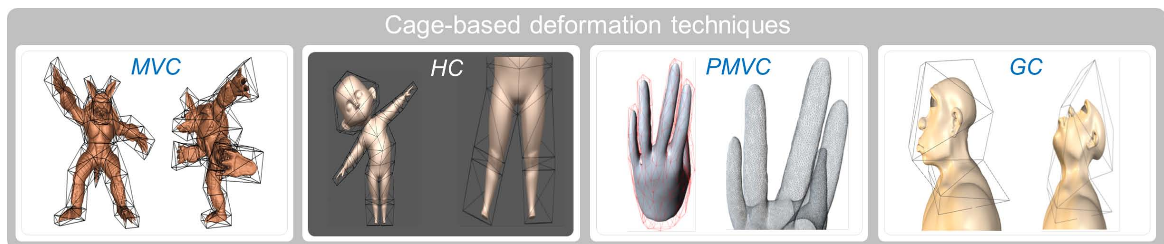


Figure 15: Different cage-based methods for mesh deformation. From left to right: Mean Value Coordinates (MVC), Positive Mean Value Coordinates (PMVC), Harmonic Coordinates (HC) and Green Coordinates (GC). Images from [39], [38], [53] and [54].

Later, Lipman et al. [54] proposed a new shape-preserving space deformation approach called Green Coordinates (GC) (see fourth image in Figure 15). The work, motivated by Green's third integral identity, produces conformal mappings, and extends naturally to quasi-conformal mappings in 3D by using both the vertex positions and face orientations. GC are C^∞ continuous inside and outside the cage but discontinuous at the boundary, although some restrictive extension mechanism can be applied.

A table summarizing the continuity properties of each coordinate is shown in Figure 16: All the cage-based approaches for mesh deformation present some kind of discontinuity at cage boundaries, and even some of them are not defined everywhere. This is why current state of the art cage-based techniques are based only in single monolitic cages to drive the deformation of an enclosed model.

Jacobson et al. [37] proposed bounded biharmonic weights, a linear blending scheme that is able to produce smooth, intuitive and flexible deformations for 2D and 3D shapes using handles of different topology (points, bones and cages). Contrary to single cage-based approaches, they can naturally use partial cages to locally deform a mesh without

LOCATION	MVC	PMVC	HC	GC
CAGE INTERIOR	🟢	🟢	🟢	🟢
CAGE BOUNDARY	🔴	🔴	🔴	🔴
CAGE EXTERIOR	🟢	🟢	🔴	🟢

Figure 16: Table showing a summary about coordinates continuity.

any special restriction or consideration.

A hierarchical approach based on a set of predefined cages was introduced by Zheng et al. [98], where user could group a set of controllers to obtain a hierarchy to deform a mesh. This approach is only applicable to man-made models and uses a small set of representative controllers. As the authors mention, they discarded cages as handlers given the difficulties at cage boundaries.

Langer et al. [45] developed a criteria for the construction of smooth maps, called Bézier maps, that are a piecewise homogeneous polynomial in generalized barycentric coordinates. To avoid discontinuities, they had to increase the number of control points and the order of the polynomials, thus increasing computational costs. In the work by Ben-Chen et al. [5], the challenge was to find a harmonic map from a domain in such a way that it satisfies constraints specified by the user, is detail-preserving, and intuitive to control. Huang et al. [36] presented a mesh deformation technique using modified barycentric coordinates with a tetrahedron control mesh that avoids first order discontinuities across the cage boundaries. Finally, even though it does not use cages, Botsch et al. [9] proposed a real-time freeform shape editing technique that allow to pose user-defined modeling constraints directly on the surface.

Another GC-based technique to locally deform a mesh contained by an automatically generated umbrella-shaped cell was presented by Li et al. [51]. Although their cage is local, they need to bind coordinates for all mesh vertices, thus increasing memory consumption. Ju et al. [40] introduced skinning templates as a solution to share and reuse skinning behaviors for similar joints and similar characters. The skinning templates were implemented using cage-based deformations, and thus they can benefit from all the features of our approach. A hybrid approach that combines surface-based and cage-based deformations were presented by Borosan et al. [8], but, as they note, it is not smooth at the boundary of the cage and meshes that are too coarse limit its effectiveness making their approach suitable only for local deformations.

Recently, Landreneau and Schaefer [44] introduced a Poisson-based method to reduce the storage needs of the coordinates for animated meshes, aiming at making a coordinate system local while still using a global cage. This technique is based on a set of initial poses and it guarantees smooth deformations if a new pose is similar enough to one of the principal ones. This limits the usage of this method only for previously known deformations and not for industrial modeling packages.

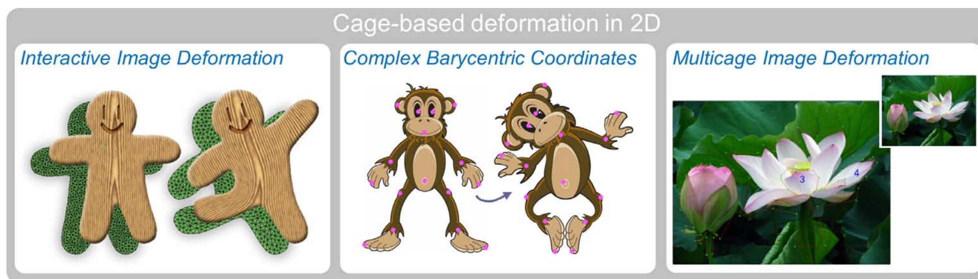


Figure 17: Cage-based deformation techniques applied to planar domains. Images from [56], [92] and [57].

Cage-based deformations have been also applied to planar domains. One related work was introduced by Meng et al. [56] (see the first image in Figure 17) who designed a method to keep the shape of images during the deformation of a region of interest, but continuity depends on the cage coordinates used (MVC, HC or GC). Later, Weber et al. [92] generalized the concept of barycentric coordinates from real numbers to complex numbers, but this is only applicable to two dimensional shape deformations (see the second image in Figure 17). In [57] the authors show how to use cage coordinates (MVC/HC/GC) to deform a 2D image while keeping its original shape. First, they unify the existing deformation coordinates into the concept of Cage Coordinates (CC) providing a generalized formulation and, finally they propose a GPU friendly implementation for interactive deformation (see the third image in Figure 17).

2.3.3 *Screen space*

State of the art Ray Tracing methods rely on acceleration structures to optimize and reduce the number of ray-traversal operations. These structures allow greater flexibility at the expense of inducing storage overhead. Rendering performance depends on the trade-off between the time for querying and updating the data structure, and the reduction in the number and accuracy of the operations. We can classify these techniques as inter- and intra-frame techniques:

- **Inter-frame techniques**. Inter-frame acceleration techniques (see Figure 19) try to use information from previous frames for the current one. Data reprojection is one of the most common strategies [59, 82], which exploits the temporal coherence in animation sequences by caching the expensive intermediate shading calculations performed at each frame. However, data reprojection is an inter-frame method unable to avoid unexpected performance drops in fast movements or drastic view changes, because of the cache misses and shading recomputations in the new view. However, these results are not updated in the cache, forbidding further reuse in nearby samples within the generation of the same frame.

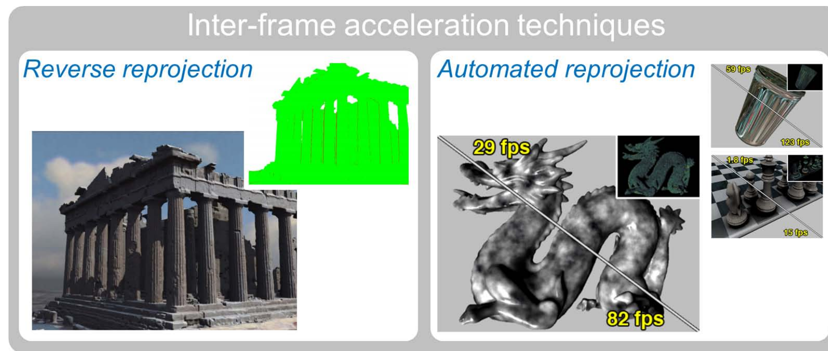


Figure 18: Several inter-frame acceleration techniques for rendering. Images from [59] and [82].

- **Intra-frame techniques**. On the other hand, Intra-frame acceleration techniques for Ray Tracing usually rely on acceleration structures to speed up the traversal of primary and secondary rays. While this problem has been well studied for CPUs [91], only a few recent approaches provided GPU-efficient dynamic ray-traversal acceleration data structures [101, 46, 30].

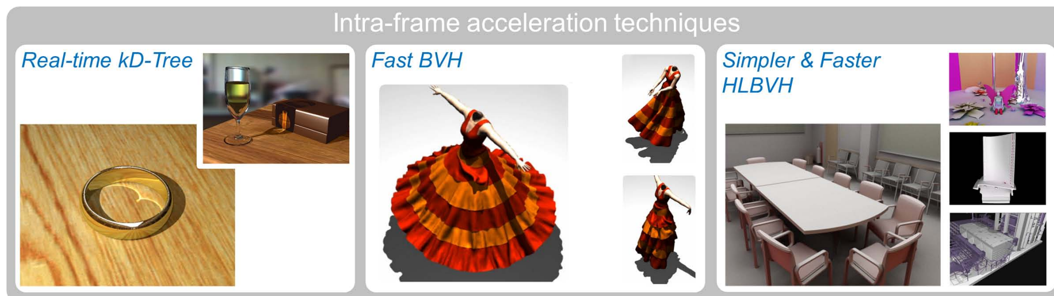


Figure 19: Several intra-frame acceleration techniques for rendering. Images from [101], [46] and [30].

In general, most ray-traversal data structures do not reduce the number of traced rays as they improve the speed of Ray Tracing and the scene hits. However, image-space techniques often show advantages over object-space techniques, because they can divorce algorithmic and scene complexities, which avoids wasting computations on off-screen portions of the scene. Szirmay-Kalos et al. [88] (see Figure 20, left) introduced an image-based structure to avoid complex ray-traversal evaluations, by looking up an approximated result from an environment map. Yang et al. [95] (see Figure 20, middle) proposed accelerating rendering by using a subsampled image and an edge-preserving (differences in depth and normal) upsampling approach to obtain the final resolution. The main drawback of this method is that it requires to process the scene twice, making it suitable only for pixel-bounded scenes. For Ray Tracing visualization this requirement would decrease the possible gain when upsampling the shading evaluations, making it suitable only for rasterized visualizations or very simple ray-traced scenes. It may also show some artifacts when reconstructing the rendering of highly tesselated models, because the large difference in normals and depths from one pixel to its neighbour. More recently, Novák et al. [60] (see Figure 20, right) suggested to accelerate Ray Tracing by converting complex meshes into a set of rasterized height fields intersected by simple ray marching. Their performance improvement is more noticeable when highly tesselated meshes are involved (from thou-

sands to millions of triangles), making this method less suitable when simpler models are used, which normally are the ones used for interactive or real-time applications. Moreover, their boost in performance is more important for primary rays, as they have a higher degree of coherence than for secondary rays-a given ray and its neighbors are going to hit the same region of the scene. Secondary rays loose most of the coherence between them, giving as a result less impressive gain over classic Ray-Tracing.

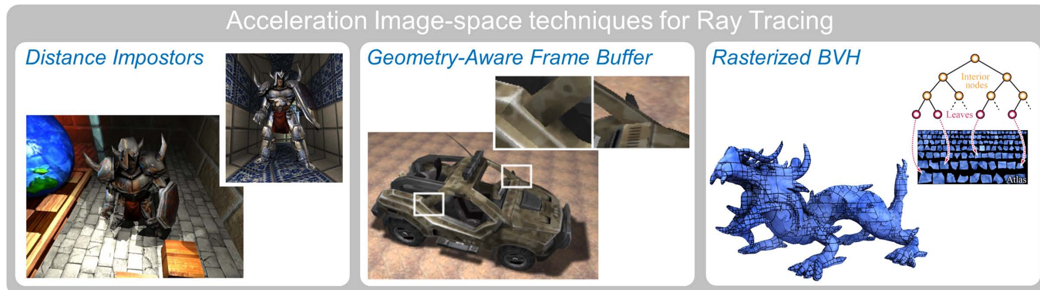


Figure 20: Screen-space techniques for accelerating Ray-tracing. Images from [88], [95] and [60].

Automatically simplifying complex shading evaluations improves the rendering performance, where less complex shaders are used in place of the original ones. The first system proposed by Olano et al. [61], only considered code transformations that replace a texture with its average color. This overlooked many possible opportunities for source-level modifications. In a similar basis, Pellacini [63] considered arbitrary source-level simplifications by a brute-force optimization strategy that can easily miss profitable areas of the shader code. Recently Sitthi-Amorn et al. [83] proposed a more effective shader optimization based on genetic programming optimization. However, while shader simplification provides a clear trade-off between performance and accuracy without additional memory requirements, it cannot reduce the number of ray-traversals. Furthermore, shader simplification requires a manual user selection of representative rendering sequences, in order to estimate performance and their error approximation, while data reuse strategies can be more easily fine-tuned automatically.

There are a set of approaches that reduce the time needed to render a scene mainly based on the reuse and interpolation of samples at homogeneous regions. Akimoto et al. [3] proposed a method that exploits the similarity between adjacent pixels to reduce the number of evaluations. Although they use a simpler sampling pattern, their procedure requires several verifications at run-time, which introduces a considerable overhead. Moreover, they use pixel intensities to determine similarity between samples, resulting in texture interpolation problems. Bala et al. [4] proposed a CPU-based system that uses per-surface interpolants to approximate and accelerate radiance computations. They decouple the acceleration of visibility and shading operations by exploiting temporal coherence and interpolating radiance samples. A hierarchical data structure called *linetrees* is used at run-time, being its maintenance one of the main drawbacks of this technique, as it is complex and costly. Adamson et al. [1] presented an acceleration method for intersectable models exploiting spatial coherence by adjusting the sampling resolution. Their main drawback lies in the lack of support for popular techniques such as shadows and ambient occlusion, which involve the computation of secondary rays. Moreover, as they shift the ray-scene

intersections computations to the CPU, GPU-based ray-tracers performance may drop due to data transfer operations.

2.4 MESH CLUSTERING

Mesh clustering techniques, also known as mesh segmentation, have become an important tool in many areas of Computer Graphics. They are characterized by the fact that they group triangles by their similarity (expressed in a given way). Mesh clustering techniques can be defined by both, their objective (surface-type or part-type segmentation) and approach (region growing, hierarchical clustering, iterative clustering, etc) used to segment the mesh. One important aspect is that the quality of a segmentation can be considered application dependent, as there is no unique clusterization.

Mesh segmentation techniques can be distinguished between two types:

- **Par-type segmentation**. The main goal of this type of segmentation is to cluster or segment the input mesh into meaningful parts. Thus, part-type segmentations decompose a 3D object into smaller parts with meaning. This segmentation has been used for modeling using each "semantic patch" to create new designs, that way the user can easily create new models from existing ones [29]. Also, part-type segmentation has been successfully used in shape matching and retrieval as well as in shape reconstructions approaches [102].
- **Surface-type segmentation**. Contrary to part-type segmentations, surface-type segmentations the objective is to divide the input mesh into charts or patches given a certain partitioning criteria. This type of segmentation is commonly used for texture mapping purposes [73, 85, 97]. Other important applications where this type of segmentation has been of great benefit are mesh remeshing, mesh simplification, morphing and mesh compression [7, 17, 86, 42].

As we have previously said, there exist several approaches to segment or cluster a mesh into charts:

- **Region growing**. This is the simplest approach for mesh segmentation. This algorithm starts with a seed element from the input mesh and grows until no more elements can be added. The main difference between region growing algorithms relies in the similarity criterion for adding a new element into the current cluster.
- **Multiple source region grow**. This approach is a variation of the region growing algorithm. The main difference relies on the fact that instead of using one single seed each time, it starts with multiple seeds advancing in parallel.
- **Hierarchical clustering**. Unlike the two previous algorithms, hierarchical clustering can be seen as a global approach. This method begins with each face being an individual cluster. Then, for each pair a cost function is computed and the lowest cost pair is merged. This procedure is repeated until no more merge can be performed.
- **Iterative clustering**. In this type of clusterization the number of clusters is given a priori. Then, the segmentation of the input mesh is formulated as a problem of iteratively finding the best segmentation for the given number of clusters. The key factor

for this algorithm is the convergence towards the final solution, which depends on the choice of the initial representative elements and the successive computation of the new ones.

- **Spectral analysis**. This algorithm uses Spectral mesh processing, which involves the usage of eigenvalues, eigenvectors, or eigenspace projections derived from appropriately defined mesh operators to carry out the segmentation of a given mesh.

Let us note, that one important aspect in all mesh segmentations techniques is the similarity criteria used to group triangles, which is usually based on a set of attributes obtained from the mesh (curvature, geodesic distances, parameterization distortion, etc) [73, 72]. For more information on mesh segmentation techniques we refer the interested reader to the excellent survey of Shamir [79].

2.5 INFORMATION THEORY

In this section we are going to review some basic concepts related to Information Theory (see [19]) and then we will discuss how they have been applied to Computer Graphics.

2.5.1 *Basic concepts*

Let $\mathcal{X}$ be a finite set, let X be a random variable taking values x in $\mathcal{X}$ with distribution $p(x) = Pr[X = x]$. Likewise, let Y be a random variable taking values y in $\mathcal{Y}$. An information channel between two random variables (input X and output Y) is characterized by a *probability transition matrix* (composed of conditional probabilities) which determines the output distribution given the input.

The *Shannon entropy* $H(X)$ of a random variable X is defined by

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log p(x). \tag{2}$$

It is also denoted by $H(p)$ and measures the average uncertainty of a random variable X. All logarithms are base 2 and entropy is expressed in bits. The convention that $0\log 0 = 0$ is used. The *conditional entropy* is defined by

$$H(Y|X) = -\sum_{x\in\mathcal{X}} p(x)\sum_{y\in\mathcal{Y}} p(y|x)\log p(y|x), \tag{3}$$

where $p(y|x) = Pr[Y = y|X = x]$ is the conditional probability. The conditional entropy $H(Y|X)$ measures the average uncertainty associated with Y if we know the outcome of X. In general, $H(Y|X) \neq H(X|Y)$, and $H(X) \geqslant H(X|Y) \geqslant 0$.

The *mutual information* (MI) between X and Y is defined by

$$I(X,Y) = H(X) - H(X|Y) = \sum_{x\in\mathcal{X}} p(x)\sum_{y\in\mathcal{Y}} p(y|x)\log\frac{p(y|x)}{p(y)}. \tag{4}$$

It is a measure of the shared information between X and Y. It can be seen that $I(X,Y) = I(Y,X) \geqslant 0$. A fundamental property of MI is given by the *data processing inequality* which

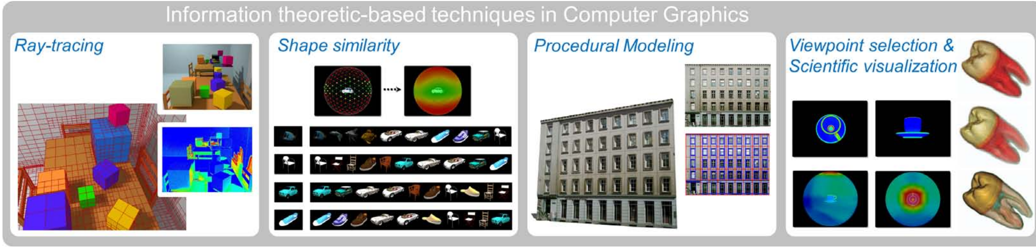


Figure 21: Techniques that use Information Theory and apply it to several Computer Graphics fields. Images from [69], [31], [58], [23] and [71].

can be expressed in the following way: if $X \to Y \to Z$ is a Markov chain, i.e., $p(x, y, z) = p(x)p(y|x)p(z|y)$, then

$$I(X, Y) \geqslant I(X, Z). \tag{5}$$

This result demonstrates that no processing of Y, deterministic or random, can increase the information that Y contains about X.

A convex function f on the interval $[a, b]$ fulfils the Jensen inequality: $\sum_{i=1}^{n} \lambda_i f(x_i) - f\left(\sum_{i=1}^{n} \lambda_i x_i\right) \geqslant 0$, where $0 \leqslant \lambda \leqslant 1$, $\sum_{i=1}^{n} \lambda_i = 1$, and $x_i \in [a, b]$. For a concave function, the inequality is reversed. If f is substituted by the Shannon entropy, which is a concave function, we obtain the *Jensen-Shannon inequality* [10]:

$$JS(\pi_1, \pi_2, \dots, \pi_N; p_1, p_2, \dots, p_N) \equiv H\left(\sum_{i=1}^{N} \pi_i p_i\right) - \sum_{i=1}^{N} \pi_i H(p_i) \geqslant 0, \tag{6}$$

where $JS(\pi_1, \pi_2, \dots, \pi_N; p_1, p_2, \dots, p_N)$ is the *Jensen-Shannon divergence* of probability distributions $p_1, p_2, \dots, p_N$ with prior probabilities or weights $\pi_1, \pi_2, \dots, \pi_N$, fulfilling $\sum_{i=1}^{N} \pi_i = 1$. The JS-divergence measures how 'far' are the probabilities p_i from their likely joint source $\sum_{i=1}^{N} \pi_i p_i$ and equals zero if and only if all the p_i are equal. It is important to note that the JS-divergence is identical to $I(X, Y)$ when $\pi_i = p(x_i)$ and $p_i = p(Y|x_i)$ for each $x_i \in \mathcal{X}$, where $p(X) = \{p(x_i)\}$ is the input distribution, $p(Y|x_i) = \{p(y_1|x_i), p(y_2|x_i), \dots, p(y_M|x_i)\}$, $N = |\mathcal{X}|$, and $M = |\mathcal{Y}|$ [10, 84].

2.5.2 *Information Theory in Computer Graphics*

Many concepts of Information Theory have been previously applied in many areas of Computer Graphics, introducing measures and relationships with important properties for different scenarios [74] (see Figure 21): Rigau et al. [69] (see the first image in Figure 21) introduced several refinement criteria for hierarchical radiosity based on the information content of a ray between two patches and the loss of information transfer between them due to the discretization. Also, Rigau et al. [70] showed how to perform and adaptive sampling method based on entropy for Ray Tracing. González et.al [31] (see the second image in Figure 21) presented a technique using several information-theoretic measures as a shape descriptions of 3D objects as a way to perform mesh similarity computations. Muller et al. [58] (see the third image in Figure 21) proposed a technique that automatically derived 3D models from single facade images and used Mutual Information as a measure to detect similarity between image regions. In the field of geometry simplification [14]

several measures has been used to perform simplifications of meshes while preserving their main properties. Finally, both in viewpoint selection [23] and in scientific visualization [71] (see the fourth image in Figure 21) Information Theory has been shown to be an important a valuable tool to obtain methods that allows the selection of the best point of views of an object, as well as a way to obtain transfer functions for a better visualization of volumetric data.

2.6 GRAPHICS HARDWARE PIPELINE

During the main chapters of the thesis we will show a number of techniques that aim to solve the continuity and interpolation problems that most of the current state of the art techniques suffeer in 2D, 3D and screen space. We have targeted these methods for working in interactive/real-time applications and, for that purpose, we have taken advantage of the features that graphics processors (GPU's) make available to speed up our algorithms, taking advantage of the parallelism in the programable units. Let's review the current graphics hardware pipeline and its programmable stages.

The graphics hardware pipeline, can be defined as a sequence of stages operating in parallel in a fixed order. Each stage of the pipeline receives an input from the prior stage and then it sends an output to the subsequent stage. In Figure 22 we show the graphics hardware pipeline with all the stages and programmable units. The stages that are rounded and colored in light blue are the programmable stages, making that way a flexible and adaptable pipeline to many scenarios. Following we explain each stage in more detail:

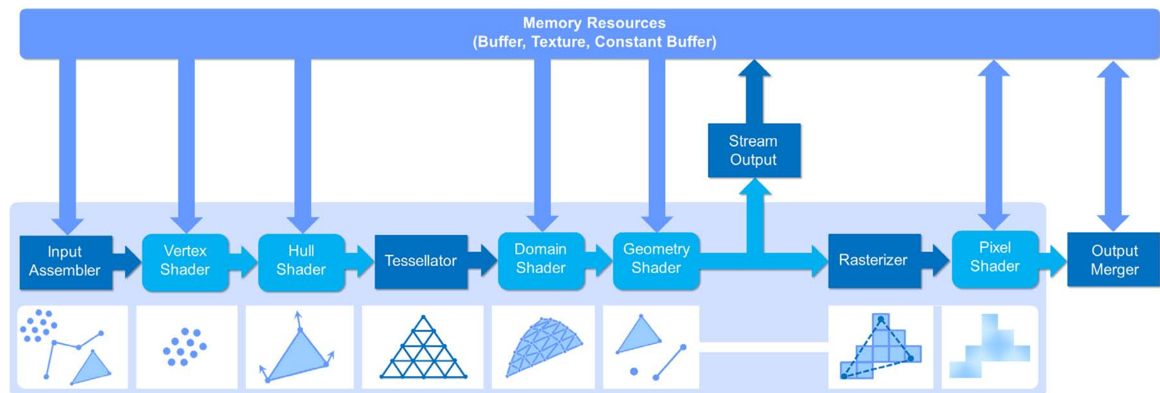


Figure 22: Graphics pipeline with all the stages and programable parts.

1. **Input-Assembler Stage**. The input-assembler stage is the responsible for supplying data (triangles, lines and points) to the whole graphics pipeline from the user-defined buffers. This stage can assemble vertices into several different primitive types, such as line lists, triangle strisp, etc. New primitive types like a triangle list with adjacency have been added to support the geometry shader stage. Another purpose of the input-assembler stage is to attach values generated by the system to help make shaders more efficient. These, system-generated values are text strings that are also called semantics, that allow to each shader stage to process only the data not yet processed. As can bee seen in the pipeline diagram (see Figure 22), once the input-assembler stage reads data from memory, assembles the data into primitives

and attaches the system-generated values, the data is output to the vertex shader stage.

2. **Vertex-Shader Stage**. The vertex-shader stage is the responsible to process the vertices coming from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. One important feature from vertex shaders is that they always operate on a single input vertex, so for each input vertex they produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. That means that if our vertex shader has no special operation to perform, we have to supply at least a pass-through vertex shader. The vertex-shader stage can consume two system generated values from the input assembler: *VertexID* and *InstanceID*. Vertex shaders are ran on all vertices, including the ones supplied for adjacency purposes and can access texture information where screen-space derivates are not required.

3. **Hull-shader, tessellator, and domain-shader stages.** The hull-shader stage, the tessellator stage and the domain-shader stage comprise what is known as the *tessellation stages* (see Figure 22). Tessellation stages convert low-detail subdivision surfaces into higher-detail primitives on the GPU, allowing to save a lot of memory and bandwidth (data flow from CPU to GPU) when rendering a high detailed surface. It can also be used to support level-of-details (LOD) techniques in real-time. First, the hull-shader stage (programmable shader) produce a geometry patch corresponding to each input patch (quad, triangle or line). Then, the tesselator stage (fixed function pipeline) generates a sampling pattern of the domain that represents the geometry patch and generate a set of smaller objects (points, lines or triangles) that connect these samples. Finally, the domain shader (programmable shader) calculates the vertex position that corresponds to each domain sample.

4. **Geometry-Shader Stage**. The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). This is one of the main differences from the vertex-shader stage, which operates on a single vertex. In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at most additional three vertices for a triangle or additional two vertices for a line. The geometry shader also supports limited geometry amplification and de-amplification, which means that given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives. The output from the geometry shader may feed the rasterizer and/or to a vertex buffer in memory through the stream output stage. This output data is done in one vertex at a time by appending vertices to an output stream object. The topology of the streams is determined by a fixed declaration, choosing one of *PointStream*, *LineStream*, or *TriangleStream* as the output for the geomtry stage. As in the previous vertex-shader stage, the geometry shader can perform load and texture sampling operations where screen-space derivatives are not required.

5. **Stream-Output Stage**. The stream-output stage streams primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU. There are two ways to

feed stream-output data into the pipeline: First, stream-output data can be fed back into the input-assembler stage and second, stream-output data can be read by programmable shaders using loading functions.

6. **Rasterizer Stage**. The rasterization stage is the one that converts vector information, which is composed of shapes or primitives, into a raster image composed of a set of pixels. During this process each of the primitives (points, lines and triangles) are converted into pixels and all the attributes associated to each vertex are interpolated across each primitive. In this step several other actions are taken into account: the clipping of vertices to the view frustum, the perspective correction, the mapping of the primitives to the 2D viewport and the determination of when a pixel shader is invoked. Culling of degenerate geometry, as the one giving adjacency information for the geometry shader stage, will happen in this stage.

7. **Pixel-Shader Stage**. The pixel-shader stage is the responsible to enable the generation of shading techniques in a per-pixel basis (i.e., per-pixel lighting and image post processing). Usually, a pixel shader program uses constant variables, texture data and the interpolated per-vertex values coming from previous stages to provide a per-pixel output data such as the color.

8. **Output-Merger Stage**. The output-merger stage combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

3 CONTINUITY AND INTERPOLATION IN TEXTURE SPACE

Inspira y suelta, eso mismo harás cuando sientas miedo para volver a recordar que estoy a tu lado.

Usually, 3D models are created as a soup of triangles that corresponds to a discrete representation of a hypothetical continuous and smooth surface. Artists parameterize these models as a way to apply material properties to them. One of the most used types of parameterizations is the multi-chart parameterization technique. It is well known that this type of parameterization introduce seams over meshes, breaking down a continuous mesh in 3D space into a disconnected set of triangles in texture space (2D), called *charts*. Charts have no connectivity between them, making discontinuities appear: First, spatial discontinuities appear as charts have no neighboring information. Second, sampling mismatches at chart boundaries arise because neighboring triangles in 3D that belong to different charts in texture space are parameterized differently, and as a consequence they have different sampling. So, it is clear that seams cause serious problems and these problems are even more noticeable for certain applications like texture filtering, relief mapping and simulations in the texture domain (e.g. fluid or chemical simulations). In this chapter we present two techniques, collectively known as *Continuity Mapping*, that together make any multi-chart parameterization seamless: *Traveler's Map* is used for solving the spatial discontinuities of multi-chart parameterizations in texture space thanks to a bidirectional mapping between areas outside the charts and the corresponding areas inside; and *Sewing the Seams* which addresses the sampling mismatch at chart boundaries using a set of stitching triangles that are not true geometry, but merely evaluated on a per-fragment basis to perform consistent linear interpolation between non-adjacent texel values. These triangles enable consistent interpolation from both sides of the chart boundary, without resampling (the vertex colors are texel center colors from the texture map). Thus, *'Sewing the Seams'* allows decoupling the interpolation along the seams from the regular texture sampling. *Continuity Mapping* does *not* require any modification of the artist-provided textures or models, it is fully automatic, and achieves continuity with small memory and computational costs.

3.1 INTRODUCTION

Multi-chart parameterization methods are widely used both to achieve low distortion texturing and to parameterize topologically complex models. When texturing, the standard approach is to perform a bilinear interpolation over a regular grid of texels defined on each chart, which can be at completely different regions in texture space. The problem is that these regular texel grids do not match up at chart boundaries (Figure 23) and bilinear signals defined over two adjacent charts will not be continuous across the boundary, which causes discontinuities in the texturing function and visible artifacts that artists usually hide by a tuning process.

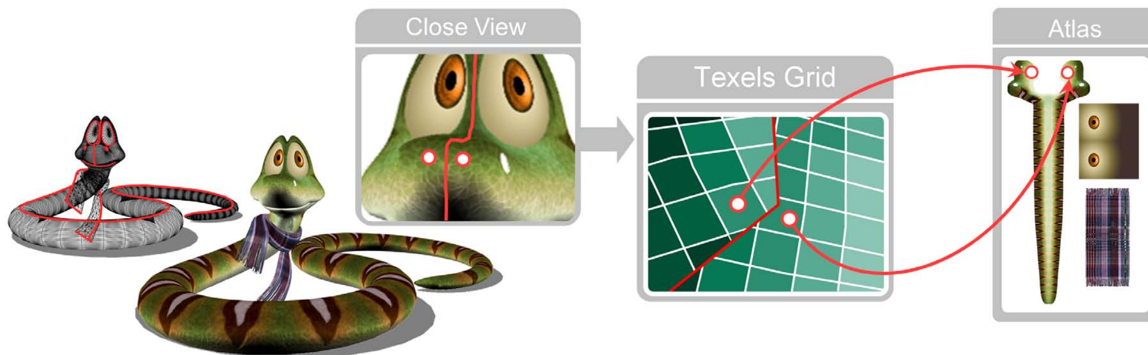


Figure 23: Motivation. Fragments from different sides of a seam are parameterized onto disjoint areas in texture space (different orientation and stretching), leading to discontinuities that are more visible in close-up views.

This problem is heightened when automatic parameterizations are used, as they usually result in an increased number of seams that are not manually fine tuned by artists. This situation causes problems for several application domains such as texture filtering or displacement/height maps, where it is often referred to as *watertight texture sampling*. All the continuity problems that multi-chart parameterizations suffer can be slightly abated by:

- **Increase texture resolution**. Increasing the resolution of the atlas does not eliminate seams, and as a consequence it will not remove the discontinuities. In fact, artifacts can be seen at *any* resolution, although the larger the resolution is, the less noticeable the seams are. In contrast, our method works correctly even with low-resolution texture atlases.

- **Padding**. Another possible solution is extending the chart and filling the neighboring area with texture values from corresponding neighboring charts. This technique is one of the most used approaches to alleviate multi-chart parameterizations problems in current texturing pipelines (games, virtual reality, ...). However, again, this only serves to reduce the visibility of the seams, which are still visible at shorter distances. In fact, any pre-filtering technique is most likely to fail at chart boundaries, mainly because of the mismatch in texel frequencies at both sides of a seam in texture space. Actually, discontinuities are intrinsic to the sampling and bilinear reconstruction process, so any automatic amendment technique can at most reduce the effects of the different sampling frequencies at the chart boundaries, but never

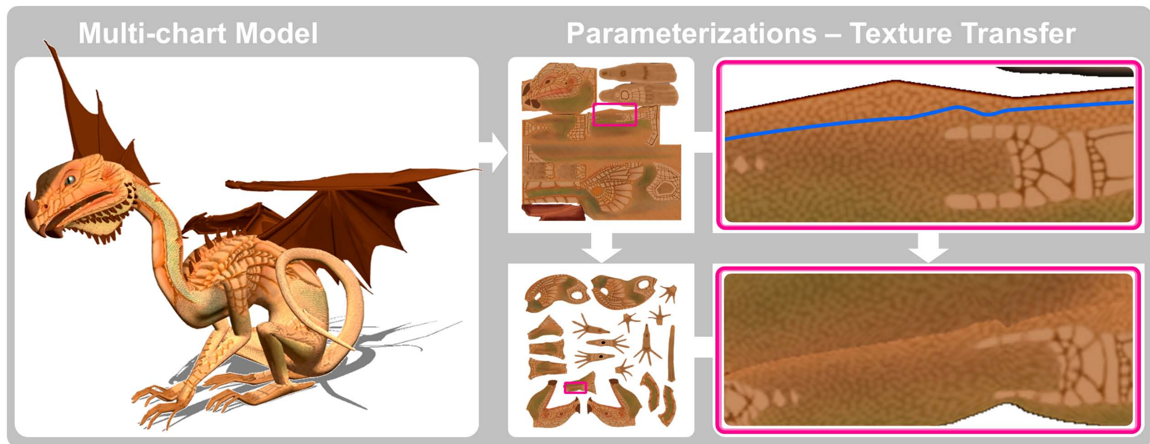


Figure 24: Texture transfer artifacts between different parameterizations.

make them disappear completely.

- **Texture transfer**. One may think to transfer the multi-chart texture into another parameterization with less seams and so less discontinuity problems, or even to transfer it into a seamless parameterization. But this approach fails, as seams exist in the original parameterization and so they are transferred as texture artifacts (discontinuities in the content of the texture itself) to the new parameterization. This effect can be seen in Figure 24 where the seam from the top parameterization (source) is transferred into a continuous region in texture space of the bottom parameterization (target). Observe how the sampling mismatch in the source parameterization cannot be erased in the target parameterization.

Our solution, *Continuity Mapping*, is a combination of two independent, but related techniques: *Traveler's Map*, which solves the spatial discontinuity problem, and *Sewing the Seams*, which solves the sampling mismatch at chart boundaries with a set of virtual texture-space triangles that connect texels in *different* charts. Thus, *Continuity Mapping* fixes the continuity problem of multi-chart parameterizations by defining a continuous reconstruction across the adjacent charts, with *small* computational and memory costs.

Chapter contributions: They can be summarized as follows

1. With *Continuity Mapping*, the texturing function becomes continuous, which is a fundamental condition for correct filtering. As a consequence, we avoid artifacts and make the parameterization seamless.
2. The method works completely in texture space and it uses the original artist's designed model with a multi-chart texture, and without modifying the artists' contents at all.
3. *Continuity Mapping* does *not* require any re-parameterization of the artist-designed model, nor the use of non-accurate texturing operations like texture transfers. This avoids the usual blurring and other similar problems that result from the mismatch of the original and the target texture resolutions.

4. It is a GPU-friendly technique that can be evaluated completely in runtime using only single-pass fragment shaders, with very low computational and memory costs.

3.2 OVERVIEW

Given a 3D textured model that has already been parameterized with any multi-chart technique, in a pre-processing stage we build both *Traveler's Map* and *Sewing the Seams* data structures to eliminate texture discontinuities later at runtime (see Figure 25).

- **Traveler's Map** defines a correspondence in texture space, which allows any point (and direction) outside a chart to be related to the corresponding point inside a neighboring chart. This information is encoded in texels *surrounding* the artist-provided charts, without modifying the artist's content at all.

- **Sewing the Seams** uses the information created by *Traveler's Map* to generate a thin border of interpolating triangles between charts to consistently filter texture values across the seams. Note that these triangles are only created in texture space, without altering the original 3D mesh. This is a great enhancement over the texture zippering techniques described, for instance, by Castaño [13] or Sander et al. [72].

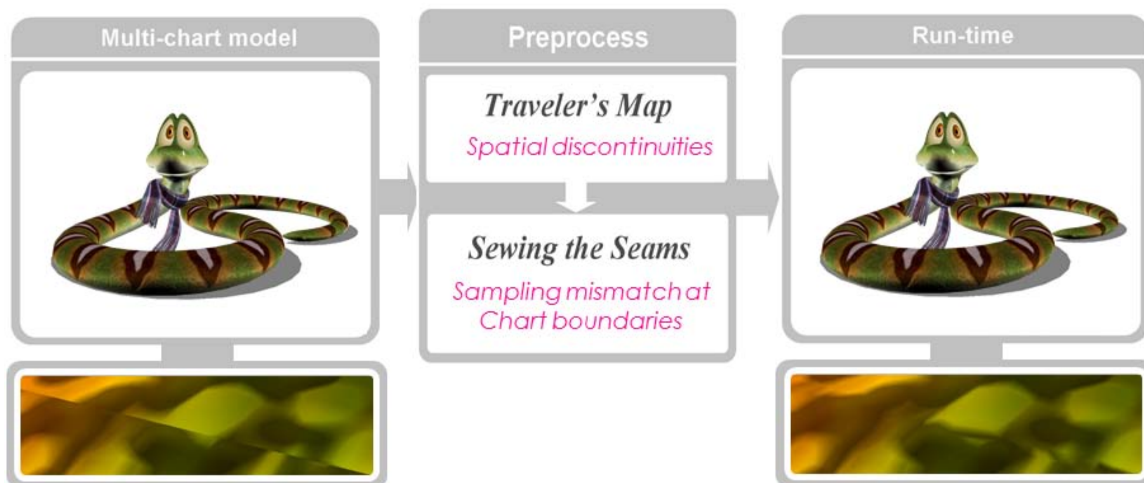


Figure 25: Continuity Mapping overview.

3.3 TRAVELER'S MAP

As mentioned before, the main objective of *Traveler's Map* is to solve spatial discontinuities in texture space introduced by multi-chart parameterizations. In the following subsections we explain how to build and use them.

3.3.1 *Construction*

Given a multi-chart parameterized 3D model, *Traveler's Map* first checks for the 3D seam edges, which are the set of edges belonging to the chart boundaries of the mesh. This procedure is done simply by looking for the edges in 3D that are parameterized to different positions in texture space. In fact, each seam edge has two unique instances, s and s', in texture space (see Figure 26) and we pair them through a transformation matrix $T_{s \to s'}$ (see Figure 27, left) defined as:

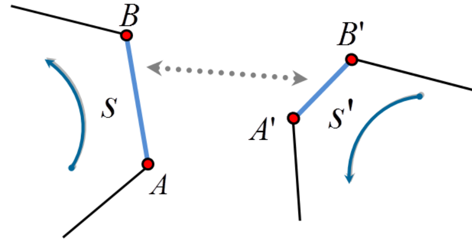


Figure 26: Transformation between edges s and s',

Transformation matrix $T_{s \to s'}$:

$$T_{s \to s'} = T \times T_A \times R_{s \to s'} \times S_{s \to s'} \times (-T_A) \quad (7)$$

where A is the projection, in texture space, of one of the two vertices of a 3D seam edge s, T_A is the translation of vertex $A \in s$ from the origin of coordinates, $S_{s \to s'}$ and $R_{s \to s'}$ are the respective scaling and rotation relationship between s and s' and T is the translation from vertex $A \in s$ to $A' \in s'$.

So, matrix $T_{s \to s'}$ transforms the points from s to s' by translating, rotating and scaling them longitudinally. Once we have computed for each seam edge s in texture space its respective transformation matrix $T_{s \to s'}$, we store it in a pair of 1D textures (float RGB32), called *Transformation Textures* for being used at run time.

Then, as shown in Figure 27, middle, we create a security border a couple of pixels wide around each chart by drawing quads that extend 2D seam edges to the exterior of the chart. For every seam edge in a chart, we consider its exterior perpendicular 2D vector as the 2D normal. The averaged 2D normal at a vertex is computed as the average of the 2D normals at the neighboring edges. The quads are built using the original seam edge, the two averaged 2D normals at each seam vertex, and the segment that closes the quad.

We render to texture these quads, and for each rendered texel, we store a reference to the respective entry in the *Transformation Textures*. It is important to note that this extra information is stored in the empty spaces between charts without modifying the artist-provided texture. As illustrated in Figure 27, right, this implies that the separation between the charts should be a few texels wide as happens with padding techniques or Indirection Maps [49].

Continuity Mapping needs a texture that stores the *Traveler's Map*. As the security borders are kept in the empty space between the artist-provided charts, we merge both textures (artist and *Traveler's*) into a single one (8 bits RGBA), thus considerably reducing memory requirements. We also add a 1-bit mask in the alpha channel to determine if a point lies inside/outside a chart. *Transformation Textures* are kept separate.

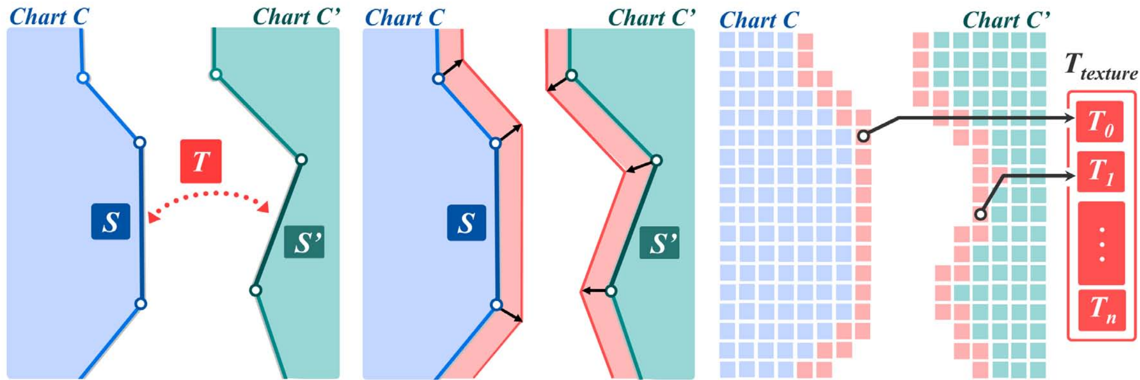


Figure 27: Traveler's Map definitions. Seams s and s′ are paired using transformation $T_{s \to s'}$ (left). We create a security border (middle) around each chart storing references to the respective $T_{s \to s'}$ (right).

3.3.2 *Usage*

Using a *Traveler's Map* is quite easy. Whenever a point with texture coordinates (u, v) has to be evaluated, we query the combined artist-*Traveler's* texture to know if it lies outside a chart, but on the security border. If it does, then the corresponding transformation $T_{s \to s'}$ is retrieved from the *Transformation Textures* and the point is transformed with $(u', v') = T_{s \to s'} \cdot (u, v)$ to a point inside the chart. The values at the coordinates (u', v') are then used to fetch the correct values for the given point. Hence, if evaluation is required outside a chart, only two extra texture fetches are needed.

3.4 SEWING THE SEAMS

As the name suggests, the function of this technique is to sew (zipper) the seams together by generating for each chart a thin border of **filtering triangles** in texture space. These triangles are then used to correctly **interpolate** and filter texture values at chart boundaries where there is a mismatch of resolutions and orientations.

3.4.1 *Construction*

The construction process for *Sewing the Seams* consists of three steps: identification of *trustworthy* texels, construction of the *Shared Triangulation*, and the construction of the *Non-Shared Triangulation*. Figure 29 illustrates this process.

Trustworthy Texel Identification: Traditional bilinear filtering is performed in the GPU by interpolating the four nearest neighboring texel centers, but here special care must be

taken if one of these centers is "outside" the chart, as its value is considered to be untrustworthy. Thus, a trustworthy texel center is defined as one being "inside" the projected chart line in texture space, with one of its eight neighbors being "outside". We can think of these texel centers as the representation of the true boundary of the artist-defined content inside the charts, that, in the authors' point of view, should be strictly preserved (see Figure 29(b)).

Once trustworthy texels have been identified, we create segments that join them. This can be done easily since almost all trustworthy texels can be 4-connected with neighboring trustworthy texels (see Figure 29(b)). If a texel center cannot be connected (e.g. because of a very acute angle between two seam edges), it is provided anyway as an independent point for the triangulation.

Next, we create an association between trustworthy texels and seam edges (see Figure 28). A trustworthy texel will be associated with the seam edges lying on the square formed by its eight neighboring texel centers, as they would affect the bilinear interpolation (see Chapter 2). Trustworthy texels (and segments) associated with only one seam edge (see Figure 28, left) will be used as input for the *Shared Triangulation*, while trustworthy texels associated with two or more seam edges (see Figure 28, right) will be taken into account during the *Non-Shared Triangulation* step. The reason to distinguish between two triangulations according to the texels involved is simple: trustworthy texels shared by more than one seam edge have more than one matrix $T_{s \to s'}$ to choose from (one for each seam edge), so we cannot use them independently, or even an average matrix, as the triangulations would not match when back-projected in 3D. That is, the triangulation cannot be shared.

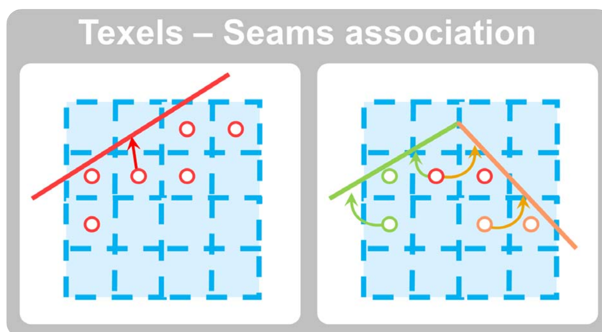


Figure 28: Texels-Seam edges association. Left: Association for trustworthy texels with a single seam edge. Right: Association for trustworthy texels with multiple seam edges.

Shared Triangulation Construction: As said before, in this step we build the triangulation using only the trustworthy texel centers shared by one seam edge. So, for every seam edge s in texture space, we use *Traveler's Map* to transform the associated texel centers (and segments) from the twin seam edge s' to the outside of s (see Figure 29(c)). Then, to guarantee valid and nice (not skinny) triangulations and, at the same time, avoid the generation of undesirable interior-chart triangles, we use a constrained Delaunay triangulation of a planar straight line graph (PSLG) [81]. To build the PSLG, we use the previously mentioned vertices and segments and close it with the first texel center, starting from the extremes, that generates a closing segment that does not intersect with an already existing segment (see Figure 29(d), bottom row). The texel centers and segments discarded by this

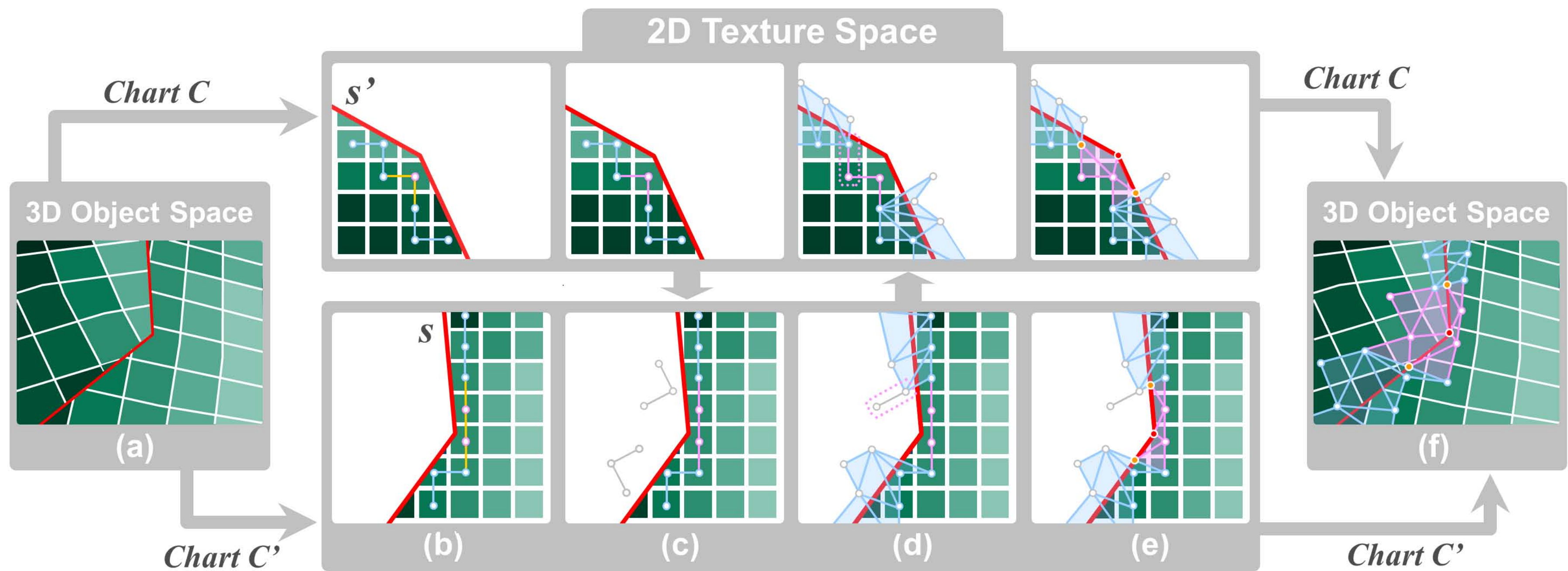


Figure 29: Construction of *Sewing the Seams*. (a) Charts and seams in 3D. (b) Identify and join the trustworthy texel centers in every chart. (c) With *Traveler’s Map*, transform the centers to the outside of the corresponding twin seam on the other chart. (d-e) Triangulate both the interior and the exterior centers, taking special care with corners where two seam edges meet. (f) The triangulation mapped back in 3D.

iterative process (like the dotted segment from Figure 29(d)) are treated later in the section on *Non-Shared Triangulation*.

Due to the shared nature of this triangulation, the twin seam edge s' will use the same set of triangles created for s but will employ the corresponding transformation $T_{s' \to s} = T^{-1}_{s \to s'}$, from *Traveler's Map* (see Figure 29(d), top row).

Non-Shared Triangulation Construction: We create a *Non-Shared Triangulation* to account for texel centers shared by more than one seam edge, along with all the texel centers and edges previously left aside. Before creating the *Non-Shared Triangulation*, we compute the two intersections between the *Shared Triangulations* near a seam vertex (red dot in Figure 29(e)) and the seam edges. We are interested in the intersections closest to the seam vertices, which create two new vertices called *Intersection Vertices* (orange dots in Figure 29(e)). Then, we build another constrained Delaunay triangulation of a PSLG with the vertices and segments discarded from the *Shared Triangulation*, the intersection vertices, the edges from the intersection vertices to the respective seam vertices, and the vertices and segments involving texels shared by more than one seam edge. Although *Non-Shared Triangulations* share the seam edges to guarantee continuity, they are built independently for each chart (see Figure 29(e)).

It may be noted that the *Sewing* triangulation works for any number of charts meeting in a single seam vertex, without requiring any special consideration, as shown in Figure 30. Also, cases where there is no possibility of building a *Shared Triangulation* (e.g. when all trustworthy texels in a chart are shared by more than one seam edge), using the algorithm described above, a *Non-Shared Triangulation* can be built without any problem. The algorithm works even with one-texel sized charts, as long as they have at least one trustworthy texel center - A chart that does not cover a texel center is a problematic case, even for the graphics hardware itself.

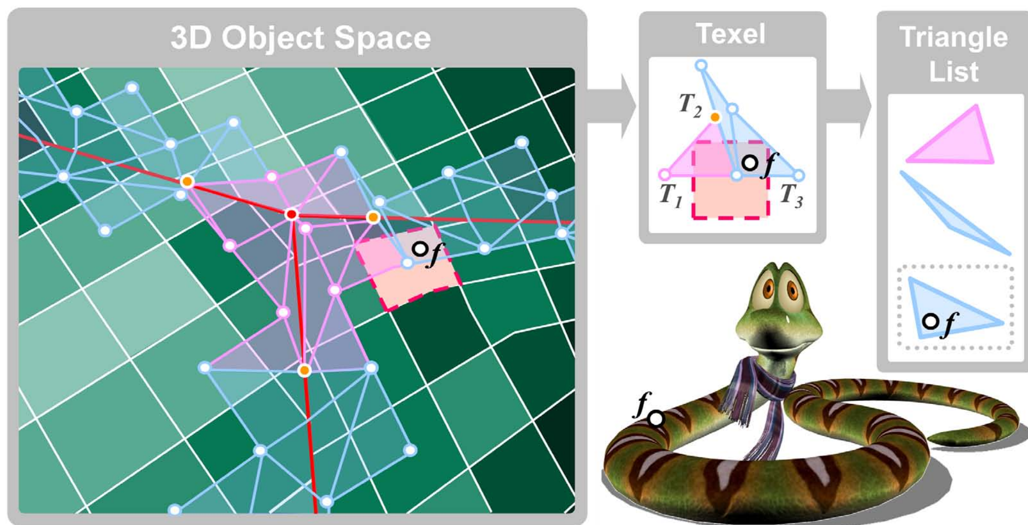


Figure 30: *Sewing the Seams* usage and data structures. Fragment f queries the list of triangles associated with the texel (T_1, T_2 and T_3), and point f is found in triangle T_3.

3.4.2 *Storage details*

For the *Sewing the Seams* technique, in the artist's texture covered by the triangulations we need to store in every texel a list of all sewing triangles that overlap with it (see Figure 30). In these texels we store a reference to a texture called *Sewing Indexes Texture* (RGB8), which stores lists of triangle identifiers. These identifiers point to the triangles stored in a third texture, called the *Sewing Triangles Texture* (float RGB32). As trustworthy texel centers have an artist-defined color, we move the color information to the first entry in the lists in the *Sewing Indexes Texture*, followed by the actual list of triangle identifiers.

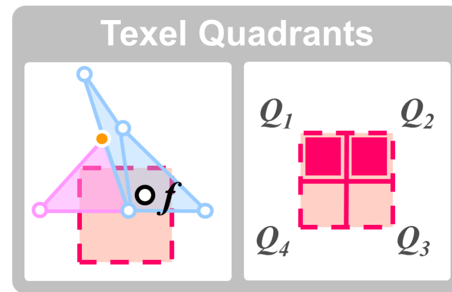


Figure 31: Sewing the Seams performance improvement by dividing each texel in four quadrants and storing 1 bit for

As all our one or two texel-wide triangulations mainly use texel centers as triangle vertices, texels are usually only half covered by the triangles. As we show in Figure 31, to avoid unnecessary evaluations, we subdivide our texels into four sub-texel quadrants, and store four bits in the empty channels of the combined artist's Texture telling the shader if there are triangles in that quadrant. In this manner, we avoid about 50% of the evaluations, achieving a significant increase in performance.

Attribute information (e.g. color) for the triangle vertices is not stored directly in our textures. Rather, we store the texture coordinates instead and use them as vertex coordinates and to retrieve attribute information. This facilitates the use of dynamic content in textures, like simulations or animations in texture space. For trustworthy texel centers, we simply store their texture coordinates and, in the case of centers belonging to the shared triangulation, the seam ID to retrieve the respective matrix $T_{s \to s'}$ for transformations between charts. For vertices that are not texel centers (intersection and seam vertices), as they do not have artist-defined attributes, we take their values from two nearby trustworthy texels. So, we just only store the coordinates and weights of these texel centers. For intersection vertices these weights come from the linear interpolation to the edge on the *Shared Triangulation*, while for seam vertices the two closest texel centers are chosen.

It is noteworthy to observe that the *Sewing the Seams* technique uses only the transformation matrices created for *Traveler's Map*, and not the security border required for the same. Hence, the spacing between the charts can be smaller than the spacing when *Traveler's Map* is used alone.

3.4.3 *Filtering with Sewing the Seams*

If a fragment f to evaluate falls in a texel which contains only color information (an alpha value different than 0), or if its corresponding quadrant has no sewing information, we evaluate it with a typical bilinear interpolation. Otherwise, we access both the *Sewing Indexes Texture* and the *Sewing Triangles Texture*, and search for the triangle that contains

the fragment coordinates. The final color is obtained by a simple barycentric coordinate interpolation of the attributes associated with the respective texel centers (e.g. the artist-provided texture) (see Algorithm 1 and the corresponding Cg code in Appendix A). Note that if needed, other sampling-based interpolation schemes, like anisotropic filtering or wider kernels, can be implemented quite easily from these data structures.

Algorithm 1 filteringWithSewingTheSeams

```
1: value = artistTex[texCoord]
2: if isAColor(value) then
3:   return getInterpolatedValue(value, artistTex, texCoord)
4: else
5:   intersectionFound = False
6:   qFlags = value.z //read the 4 quadrant flags
7:   if currentQuadrantHasSewingInfo(qFlags,texCoord) then
8:     triangleListId = value.xy
9:     texelCenterColor = sewingIdsTex[triangleListId]
10:    numTriangles = sewingIdsTex[++triangleListId]
11:    while numTriangles> 0 & !intersectionFound do
12:      triangleListId++
13:      intersectionFound = pointInTriangle(sewingTrisTex,
                                triangleListId, texCoord)
14:      numTriangles– –
15:    end while
16:  end if
17:  if intersectionFound then
18:    return computeTriangleColor(sewingTrisTex,
                 triangleListId, texCoord)
19:  else
20:    return getInterpolatedValue(value, artistTex, texCoord)
21:  end if
22: end if
```

3.5 MIP MAPPING AND SHADER LOD

Continuity Mapping allows seamless mip-mapping over multi-chart textures. First, we construct the *Continuity Mapping* pyramid repeating the procedure described in Section 3.2 for different resolutions. As the results in Section 3.7 demonstrate, the construction time required is short, making this procedure quite practical. Then, in run-time, given the sewing triangles at each mip-map level, interpolation is computed at two texture resolutions and then interpolated between them. Each sample for each level may come from either a barycentric (linear) interpolation on a filtering triangle (evaluated with attributes from the textures themselves), or a standard bilinear interpolation on the texel grid. Obviously, the cost is the sum of these evaluations plus the interpolation of the final values.

Although *Continuity Mapping* works for every distance, we shift to regular textures at medium distances, when the use of *Continuity Mapping* is no longer noticeable. It is important to note that, even for large distances where we use a regular texture, if there is an animation or we are computing a continuous simulation in texture space, a *Traveler's Map* should at least be built to guarantee the continuity of the simulation in all mip-map levels.

3.6 APPLICATIONS

In this section we are going to introduce some applications that demonstrate the capabilities of *Continuity Mapping*.

Seamless Texture Filtering. This is one of the most straightforward applications of *Continuity Mapping*. As can be seen in Figures 32, 33 and 34 we show three models: the bunny (5058 triangles), the Neptune (80000 triangles) and the snake (25448 triangles), respectively. The first two models have been parameterized with Iso-charts [99], while the later has been carefully parameterized by an artist to explicitly hide the seams. Also, observe how the bunny and Neptune models are parameterized into 100 charts (7218 seam edges) and 16 charts (379 seam edges), and the snake model is made by only four charts (1422 seam edges). The first and third columns in the figures show the corresponding textures applied to the models using the padding technique, while the second and fourth columns show the same texture but using the *Sewing the Seams* solution. The bottom row show an inset of the mentioned images. As can be seen, seams are still visible at close ranges, even in the snake model, despite of the fine-tuning the artist did over the parameterization.

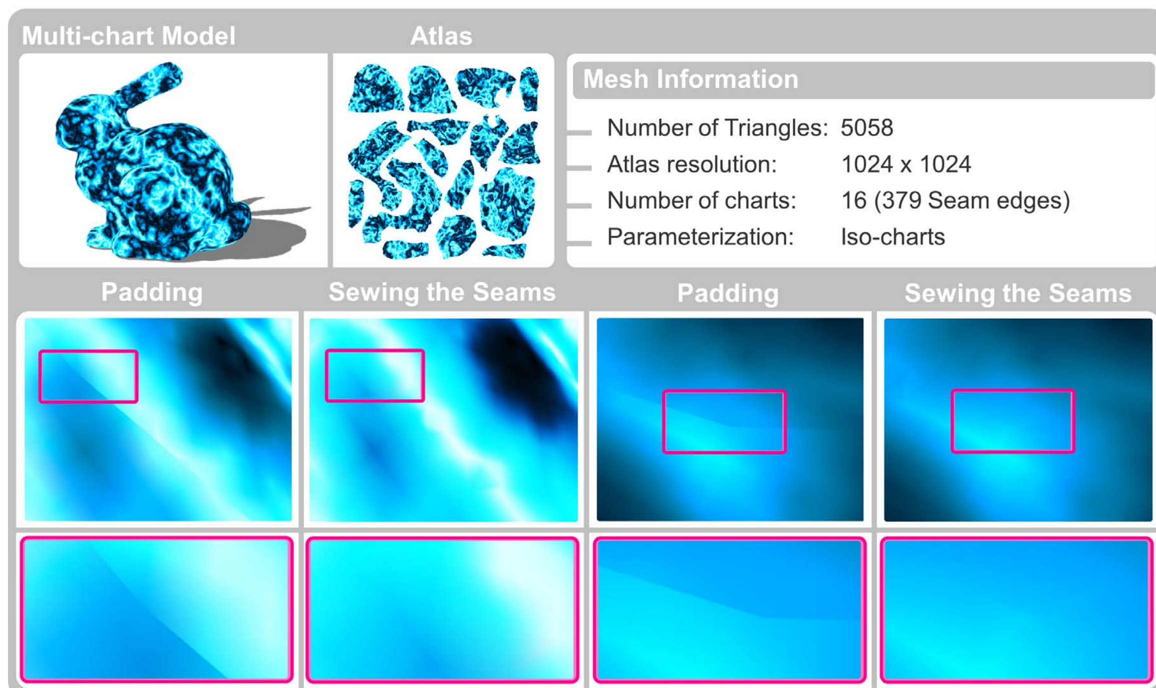


Figure 32: Seamless Texture Filtering on the bunny model (Atlas 1024^2). First and third column: padding. Second and fourth column: *Sewing the Seams*.

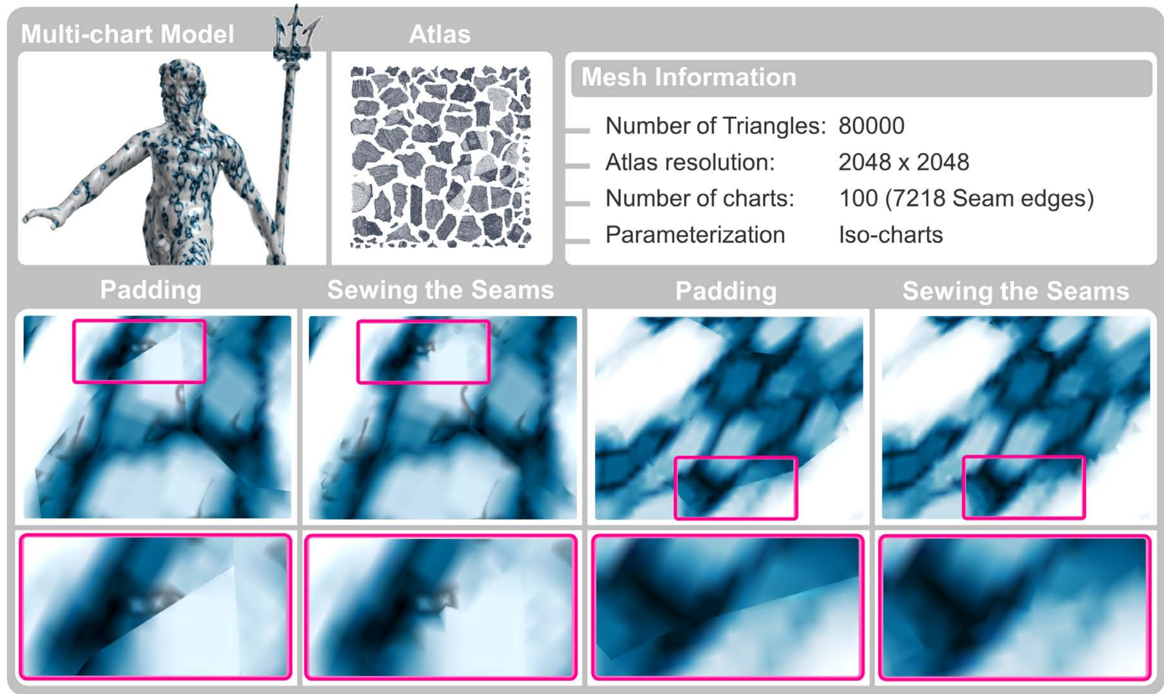


Figure 33: Seamless Texture Filtering on the Neptune model (Atlas 2048^2). First and third column: padding. Second and fourth column: *Sewing the Seams*.

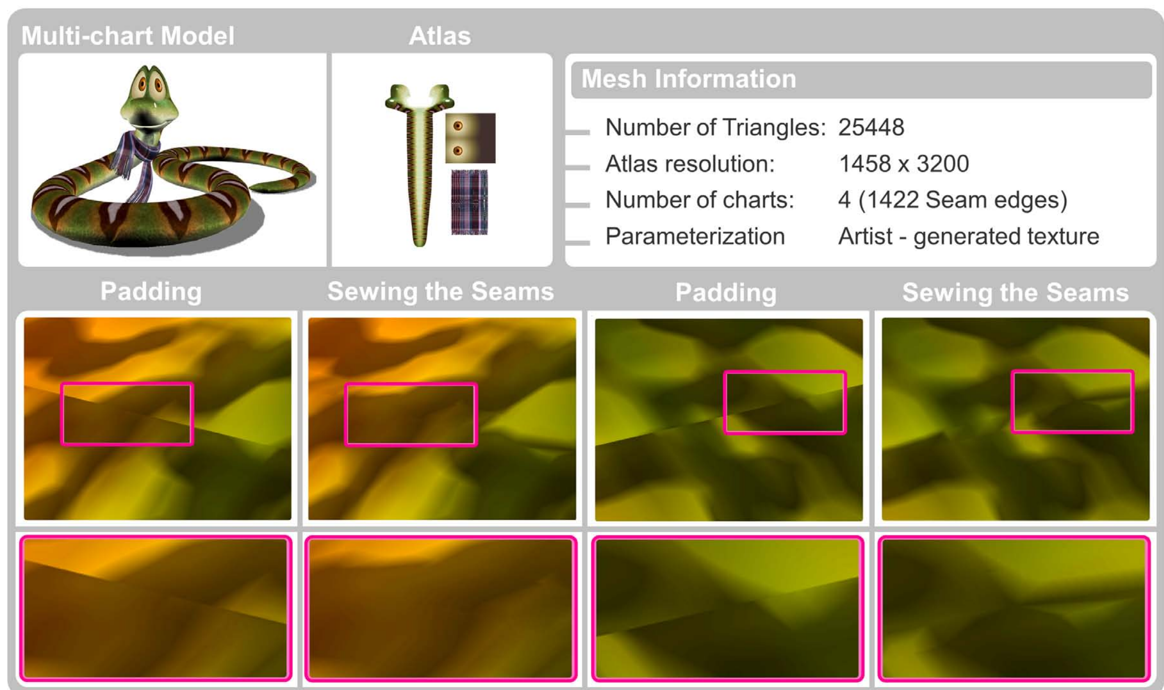


Figure 34: Seamless Texture Filtering on the snake model (Atlas 1458x3200). First and third column: padding. Second and fourth column: *Sewing the Seams*.

Also, *Sewing the Seams* shows its stability working with atlases like the one used for the Neptune mesh, which has a large number of charts, including several small ones, even

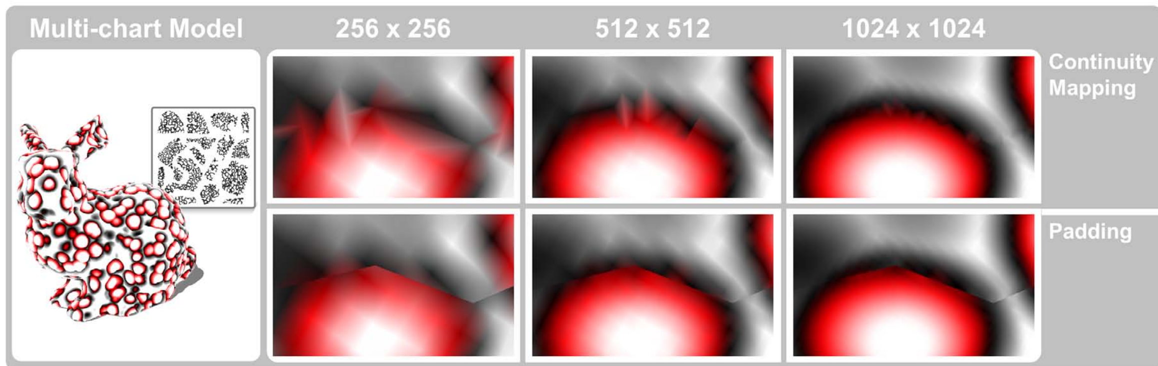


Figure 35: *Sewing the seams* at different resolutions on the bunny model. Top row: *Sewing the Seams*. Bottom row: Padding.

containing seams at the subtexel level.

In Figure 35 we show the quality of the interpolation achieved by *Sewing the Seams* depending on the texture resolution of the atlas. As can bee seen, even when differences in the content from both sides of the charts are high, as can bee seen in the sharp feature shown, our approach (top row) provides a better solution than traditional padding (bottom row). At low resolutions with really sharp features in the textures the *sewing* triangulation may result apparent (Figure 35, left) and as a consequence the interpolation can be less smooth. Note how the bigger the resolution is, the smoother the interpolation performed through the seams.

Continuous simulations. Another interesting application of *Continuity Mapping* is in the area of continuous simulations like 2D fluid simulations [87], droplets [41] and Reaction Diffusion [93]. In general these simulations are usually done onto regular mappings, as did Carr et al. Carr et al. [12], to simplify texel neighborhood computations. Not only is this restrictive in nature, it also introduces a strong texture distortion that is not good for numerical simulations involving derivative computations [93]. In Figures 36 and 37 we show a Reaction Diffusion simulation while in Figure 38 we show a droplet simulation. All these simulations are carried out completely in texture space, so we can use the *Traveler's*

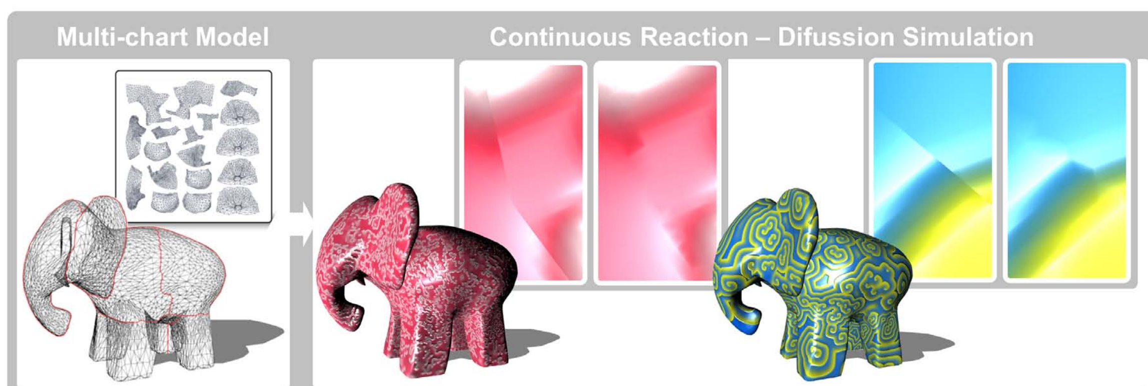


Figure 36: A parameterized elephant model (13402 triangles) and two different results for continuous reaction-diffusion simulations. Left: padding. Right: *Continuity Mapping*.

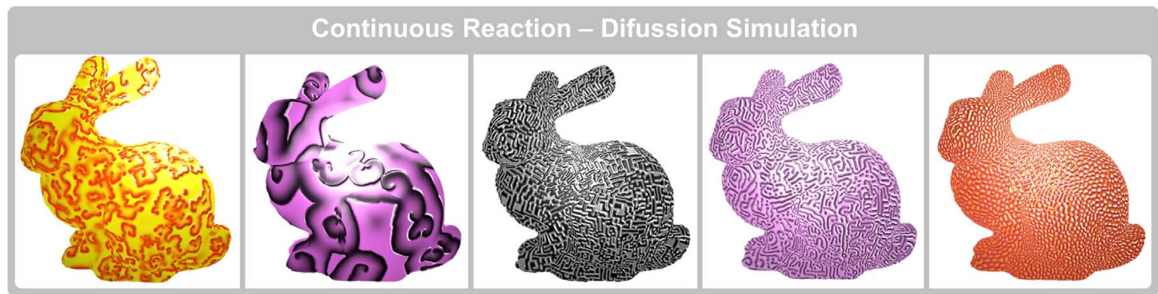


Figure 37: The bunny model from Figure 32 with different results for continuous reaction-diffusion simulations.

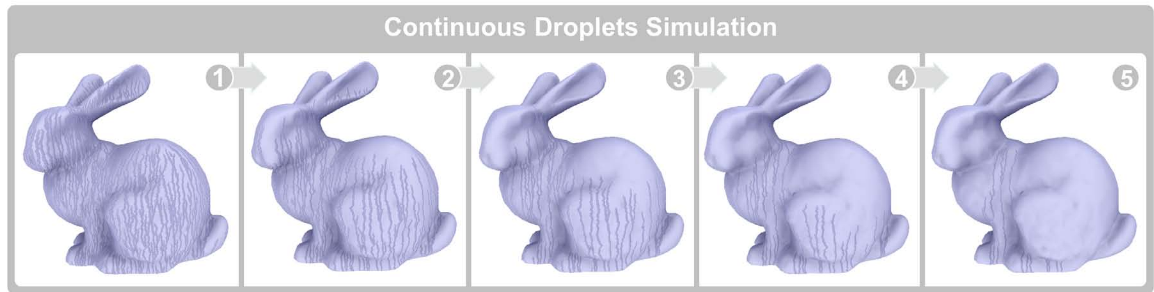


Figure 38: Droplet simulations performed on the bunny model from Figure 32.

Map to evaluate simulation properties when the methods require sampling outside a chart. In that case, we use the associated matrix to compute a texel inside the chart sharing the common seam edge. For rendering, we use the full *Continuity Mapping* technique, including *Sewing the Seams* for correct texture filtering in addition to bump mapping. Note that a correct bump mapping requires the computation of the normal field and its interpolation through the seams, which can also be done with our technique.

Multi-Chart Relief Mapping. Another application of *Continuity Mapping* is Relief Mapping [65] over multi-chart textures. Up to now, general objects have been taken into account only over simple continuous parameterizations [15] or by solving only for the silhouettes [62], but the problem of using Relief Mapping in combination with a multi-chart parameterization remains unsolved. The work done in [20] is the closest to this applica-

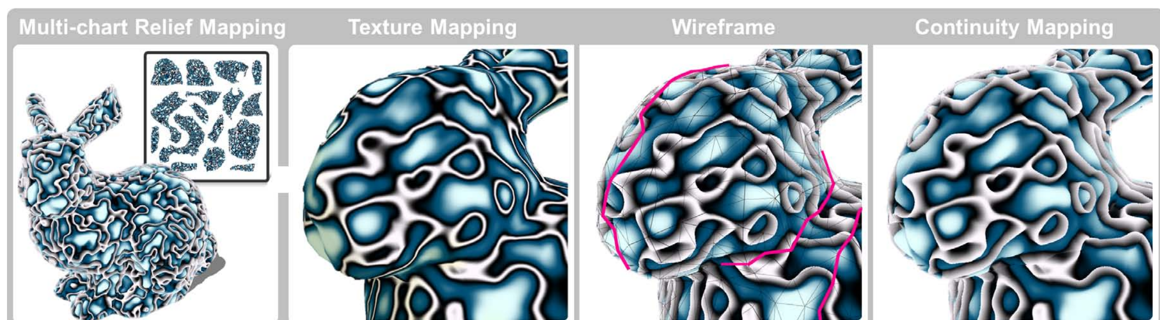


Figure 39: Multi-chart Relief Mapping. Comparison between simple padding, Indirection Maps and Continuity Mapping.

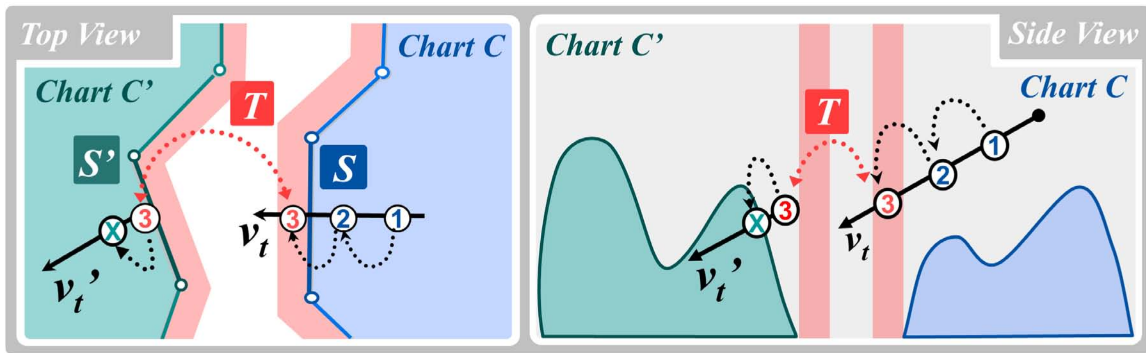


Figure 40: Tracing a ray with Multi-Chart Relief Mapping.

tion of *Continuity Mapping*, where continuity between patches is solved by extending and overlapping them, solving visibility by depth-fight with the *same* color and depth. But, as mentioned there, this can cause serious problems for grazing viewing angles, shadows and filtering operations on the object surface. With *Continuity Mapping*, seamless multi-chart relief mapping is possible (see Figure 39). Figure 40 shows that it is possible to sample outside a chart while searching for the intersection point. Since there are security borders surrounding the charts, if we sample on a security border, we use the *Traveler's Map* to retrieve the 2D matrix that will transform the point on the *outside* of one chart to the *inside* of the corresponding chart where the search should continue. It is important to note that we not only transform the point, but also the traveling direction, so the search for the intersection point can continue without any issues on the other chart. In fact, the *Traveler's Map* gets its name from this functionality – if you are lost while travelling the multi-chart parameterization, simply use the *Traveler's Map* to find the direction. When the intersection search process requires sampling in areas very close to the seams, we use *Sewing the Seams* to correctly interpolate height and color values in this area. The only extra precaution that needs to be taken is adjusting the maximum step length to be smaller than the security border width. However, if the step is made larger, the algorithm can backtrack until it finds the security border when empty space is found instead.

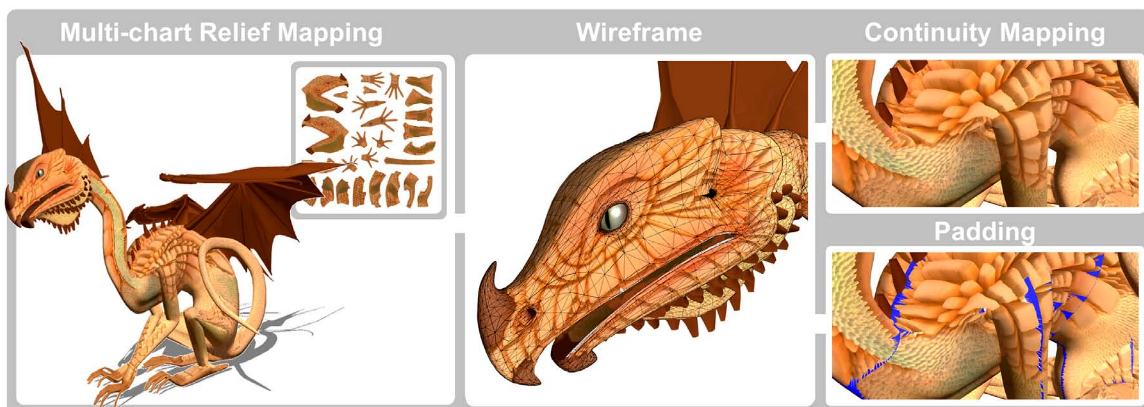


Figure 41: Multi-chart Relief Mapping on the Loiosh model. Comparison between simple padding, and Continuity Mapping.

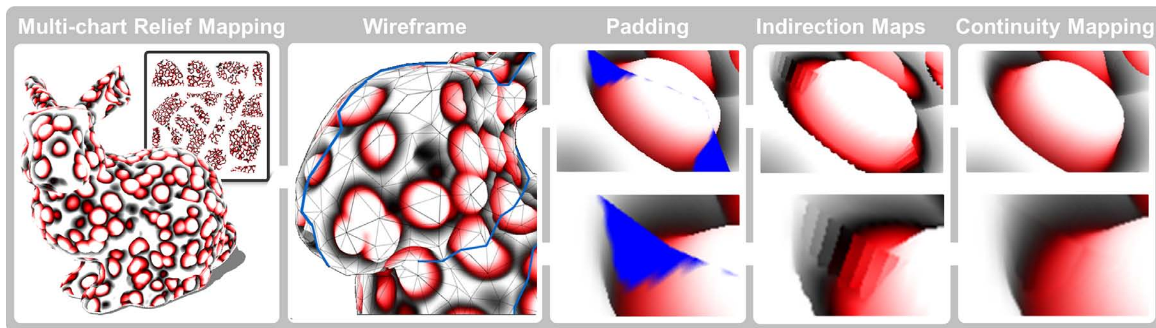


Figure 42: Multi-chart Relief Mapping on the bunny model. Comparison between simple padding, Indirection Maps and Continuity Mapping.

Also, as demonstrated by Figures 41 and 42, *Continuity Mapping* proves to be more accurate than padding (blue artifacts are fragments evaluated outside the charts) and Indirection Maps [49], which is not surprising considering that our approach provides sub-texel accuracy transformations and also preserves directions.

Other applications. Displacement Mapping and Caustics computations in the GPU are applications that can also benefit from the use of *Continuity Mapping*. On one hand, displacement mapping using multi-chart textures introduces undesirable holes in the mesh along the seams, producing much more visible artifacts that are harder to hide than texturing artifacts [13]. Here, *Continuity Mapping* can provide a continuous sampling over the seam regions avoiding the aforementioned geometry cracks. On the other, real-time caustics (for example, Photon Mapping in texture space) require photon hit filtering. This is usually achieved by blurring hits using splatting, but photon splats may fall on a chart border, resulting in energy being lost due to part of the photon being splatted outside a chart [89]. Yet again, *Continuity Mapping* can prove to be a valuable tool as it accumulates energy at the correct texture charts.

3.7 RESULTS AND DISCUSSION

Preprocessing times for *Continuity Mapping* are small, ranging from 22 seconds for the bunny model (5058 triangles) up to about two minutes for the Neptune model (80000 triangles). Also, in the latter example we can see that *Continuity Mapping* behaves correctly with large meshes with subtexel seam edges, as they are naturally included in the *Non-Shared Triangulation*. The only requirement is that the charts must have at least one trustworthy texel center. Also, just like the padding technique, there must be a few pixels of separation between charts, between non-topologically-adjacent edges of the same chart or between consecutive edges with very acute angles, for the *Traveler's Map* to work. However, in all our experiments we have not come across any noticeable problems in these cases. Some applications might require extremely high precision for the case of subtexel edges in the *Traveler's Map*, which cannot guarantee which edge will provide the matrix for that texel. This can be solved by keeping, for every texel in the border, a list of all quads that project onto it, as done in the *Sewing the Seams* technique, but we considered this to be unnecessary and inefficient, not having observed any noticeable artifact in any of our experiments.

The increase in memory consumption of an artist' texture goes from 4MB for a 1024^2 resolution to 36MB for a 3072^2 resolution, while the memory consumption for *Continuity Mapping* goes from 1.3MB for a 1024^2 texture resolution to a mere 3.46MB for a 3072^2 texture resolution. A quick look at Figure 43, reveals that *Continuity Mapping* data structures grow in the order of $O(n)$ as the texture resolution grows in the order of $O(n^2)$. The reason for this is simple: these data structures follow the chart boundaries, which are basically 1D, embedded in a 2D space. The model used in the graph is the bunny from Figure 42 parameterized with LSCM [50]. This storage efficiency can be compared to Perfect Spatial Hashing (which requires unconstrained access) [48] because the *Traveler's Map* is not sparse. Furthermore, the textures for *Sewing the Seams* are equivalent to a Binary Image (embedded in the artist's texture) and a Hash Table texture, but without the need for an extra Offset texture.

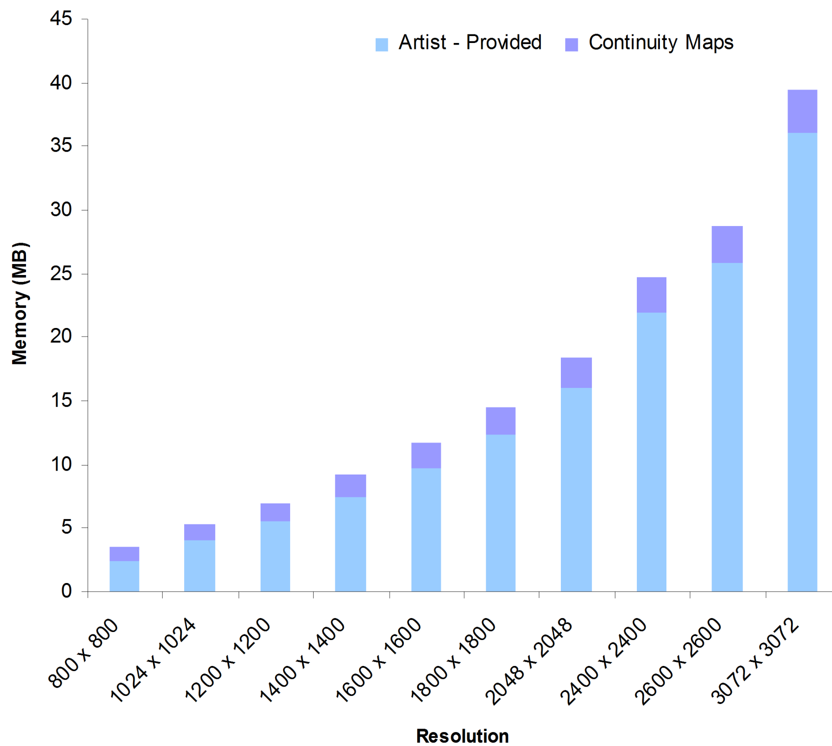


Figure 43: Left: Memory consumption of *Continuity Mapping* depending on the atlas resolution.

The evaluation cost for *Continuity Mapping* varies from case to case, but the results can be generalized into two main categories: the cases where only the *Traveler's Map* is used, and the cases where both *Traveler's Map* and *Sewing the Seams* are used. While in the former scenario the cost is independent of atlas complexity (as explained in Section 3.3.2), in the latter, using *Sewing the Seams* varies the cost in different situations. But in general, we obtained high frame rates in all our experiments, with more than 1150 fps in an Nvidia GeForce GTX 280, while the models with regular texture filtering were displayed at 2400 fps. The graph in the Figure 44 shows that for a given resolution, *Continuity Mapping* requires less computational power as the model on screen gets smaller. This is due to the fact there are fewer fragments to evaluate. The aberration on the left part of the graph is

caused by the continuously varying fragment count (the ones requiring *Sewing the Seams*), since the model is only partially visible. Also, the *sewing* lists were observed to have, in general and on average, 3.5 entries.

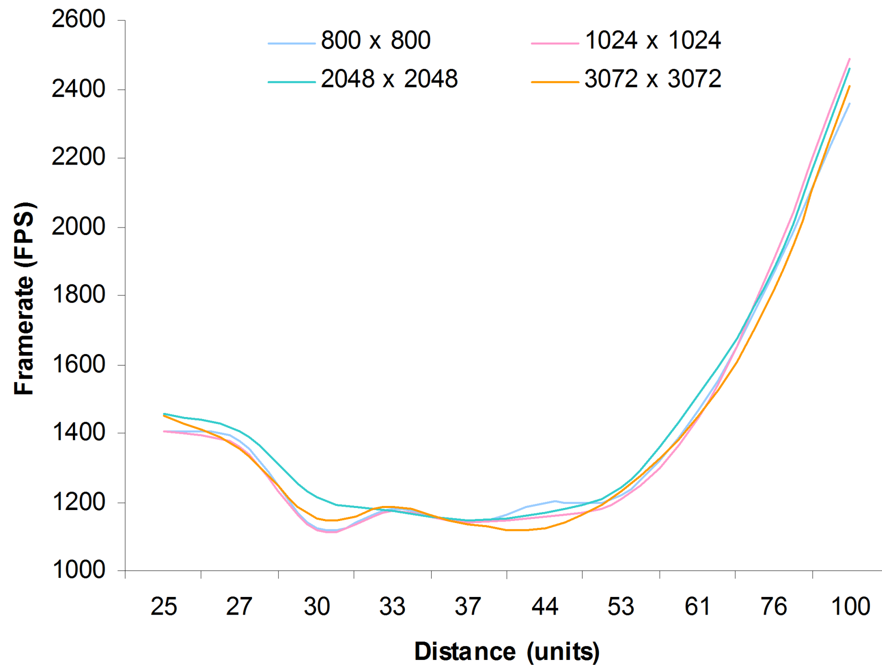


Figure 44: Dependence of the frame-rate on the distance to the observer (bunny model, viewport: 1024×768).

It is important to note that matrix $T_{s \to s'}$ (see Section 3.3) stores a simplistic edge-longitudinal stretching measure. In cases of extreme stretching in the direction orthogonal to the edge, information from one texel can be taken farther away from where it should be. This case may require storing a per texel Jacobian, which is more accurate, but would largely increase memory requirements, and in our experience this has not been necessary.

Another important consideration regarding *Continuity Mapping* is that it works entirely in texture space, so it strongly depends on the texture resolution and stretching. For instance, if the texture resolution is doubled, the resulting triangulation for *Sewing the Seams* would be much thinner in 3D space around the seams, and would in turn cause a smaller and smoother interpolation. On the other hand, if the texture is stretched more in one direction than the other, the resulting triangulation could result in skinnier triangles, and thus a lower quality triangulation. Also, if the scaling between two seam edges is too large, Continuity Maps will produce a smooth interpolation, but at a lower quality than when the edges have similar transformations. In our experiments, we did not observe any problems for scalings up to slightly more than 50%.

Continuity Mapping is limited when trying to smooth the seams of a texture with sharply varying features because it only works on a thin layer, up to two texels wide, across the

seams. If the features on both sides of the seam do not match, *Continuity Mapping* will provide smooth continuity at short distances, but the seams will still be visible at medium and large distances because the features mismatch at larger scales, even when mip-maping is applied. This would probably require a smoothing pass with a large matrix, which could easily be computed with a *Traveler's Map*. In any case, solving a global texturing mismatch is not the objective of this article, but it does to add continuity at the seams between charts without changing the original parameterization.

Sometimes artists use mirroring operations to build their meshes, resulting in the two halves of the object being mapped in the same area in texture space. The techniques presented so far cannot deal with this non-bijective parameterization, but this can be easily solved by mirroring the charts in texture space and restoring the bijectivity, which can be done automatically in a pre-process stage.

The explanations above refer to achieving correct interpolation of values/attributes across chart boundaries. Achieving continuity of higher order functions like successive derivatives (e.g. normal mapping) is simply a matter of using the functionality described above with that information. For instance, continuous normal mapping for a height field could be built in two simple passes: the first one computes a normal for each texel by using the height of its four immediate neighbors (using the full *Traveler's Map* at the chart boundaries), and the second pass would use *Sewing the Seams* to generate the continuous from the normal map.

Continuity Mapping is compatible with mesh deformations/animations, as it works completely in texture space without altering the original model geometry. Moreover, as seen in the Continuous Simulation, the information stored in the textures need not be static, and can be used with dynamic simulations or animations in texture space.

3.8 conclusions

We have presented *Continuity Mapping* a technique that solves the continuity problems that arise in multi-chart parameterizations, which currently are one of the most used texturing techniques in the industry (videogames, virtual reality, films...). The technique is composed of two independent but related sub-techniques: *Traveler's Map* and *Sewing the Seams* allowing to solve both, spatial discontinuities and sampling mistmach at chart boundaries where discontinuities appear. We also have demonstrated the power of this technique by suggesting some applications that can be made seamless and, as a consequence, continuous, in contrasts with previous approaches. Now we are able to provide seamless texturing on a mesh even if its source parameterization is discontinuous. We would like to note that, one year after our work got published, Ray et. al in [68] presented an approach to make seams invisible, which is close to our technique. They align texel grids across chart boundaries by assigning them a new set of texture coordinates computed through a global parameterization. In addition, they modify the colors stored in the texels related to chart boundaries, as well. Comparing their approach with our work, let us note that we do not need to recompute texture coordinates for any mesh vertex and we do not even need to modify the color of the corresponding chart boundary texels, as we wantd to preserve the original artist-provided texture and parameterization. On

the contrary, they do not need to use any run-time shader to hide discontinuities, thus accelerating computations.

4 CONTINUITY AND INTERPOLATION IN OBJECT SPACE

Lo importante no es el resultado, sino el camino y esfuerzo realizado. Recupera tu fuerza, vuelve a confiar, márcate tu objetivo y nosotros estaremos allí. Confía.

Let's move into the second stage of our pipeline: the deformation stage. Once we have the model parameterized and textured without discontinuities, we may want to deform it and create new alternative poses other than the initial one. As mentioned in previous chapters, for that purpose there exist several approaches, but cage-based methods have been one of the main trends for mesh deformation in recent years, with a lot of interesting and active research. They are characterized by the fact that they apply deformations by using a cage which encloses the model. This cage should be similar in shape to the model to be deformed, but much simpler. The main advantages of these techniques are their simplicity, relative flexibility and speed. However, to date there has been no widely accepted solution that provides both user control at different levels of detail and high quality deformations. Thus, in this chapter we present *Cages (star-cages), a significant step forward with respect to traditional single-cage coordinate systems, which allows the usage of multiple cages enclosing the model for easier and faster mesh deformation. As we will show, the fact that we use several cages instead of one has many benefits, but at the same time forces us to face new challenges regarding discontinuities in 3D that arise between cages. So, we will need to solve these challenges if we want to obtain a smooth deformation as a result. The proposed deformation scheme is extremely flexible and versatile, and it will allow the usage of heterogeneous sets of coordinates as well as perform deformations at different levels of details, ranging from a whole-model deformation to a very localized one. Locality will give as a result faster evaluation and a reduced memory footprint, and as a consequence we will outperform single-cage approaches in flexibility, speed and memory requirements for complex editing operations.

4.1 INTRODUCTION

Shape deformation in both in two and three dimensions plays a central role in computer graphics. Space deformation techniques especially cage-based methods as a practical means to manipulate 3D models [26] [39] [38] [54] [92], have received a lot of attention. A cage is a low polygon-count polyhedron, which typically has a similar shape to the enclosed object. The object points inside the cage are represented by affine sums of the cage elements (vertices or faces) multiplied by special precalculated weight functions called coordinates. The main advantages of these space deformation techniques are their simplicity, relative flexibility and speed on applying the deformation. Also, as each point is transformed independently, these techniques are indifferent to the surface representation and in general free of discretization errors.

However to date there has been no widely accepted solution that provides both user control and high-quality deformations. It is commonly accepted that an ideal deformation system should allow user intervention when required, but automatically infer all the missing data. For instance, given a user-chosen set of constraints, the system should find the best deformed shape that satisfies those constraints. Several possible alternatives do exist, such as Mean Value Coordinates (MVC) [26], Harmonic Coordinates (HC) [38] or Green Coordinates (GC) [54]. All of them are characterized by the use of a single cage to compute the final deformation. This particularity presents some problems:

- **Locality**. Current cage-based deformation approaches can be classified as global deformation methods because they are defined in terms of a single cage which affects *all* mesh vertices and, meaning they cannot produce local deformations.
- **Time and memory consumption**. The global behavior of single cage-based techniques results in each point storing weights for all cage vertices, increasing both the memory consumption and the number of evaluations.
- **Smoothness**. As we will later explain, all single cage-based methods have continuity problems on the cage boundaries, ranging from the lack of smoothness to the presence of discontinuities. This is one of the main reasons why their use is currently limited to monolithic single cages.
- **Coordinate combination**. Each single cage-based method uses different types of coordinates and, as a result, the deformations also differ (e.g. MVC and GC). The user has to decide to use one coordinate type or another for the whole model, depending on the desired results, and without the option of combining their strengths.
- **Usability**. The design and manipulation of a unique cage to deform a whole model may result in a daunting task and may convert single cage approaches into a much less user-friendly tool for current animation pipelines.

In order to avoid such problems we should be able to use many cages instead of just one, and as a consequence, they would be easier to create and manipulate. Each of these cages should use different coordinate types, which would increase the potential of different results over the final deformations. All these cages should be used at different levels of granularity to allow a more localized control over the final deformation, and so consume

only the resources needed for each cage in isolation.

In this paper we present *Cages (pronounced *star-cages*), a cage-based deformation method that involves a hierarchical set of cages where the leaf cages bound the object in a piecewise manner. Cage-coordinates can be individually defined for each leaf cage, and blended among neighboring cages to produce a smooth (class C^1) deformation, thus offering localized deformation control with less computation. The hierarchy further allows deformation control to take place at multiple levels. In this sense, we can say that *Cages *complements* the existing techniques rather than *competing* with them. Hence, the rationale behind of its name: *Cages can accommodate *any* coordinate system inside a cage, and smoothly combine *any* number of cages to obtain a flexible and general deformation system at *any* level of detail.

The main contributions of the proposed technique are:

1. An unlimited number of cages to smoothly deform a base mesh.
2. This set of cages produces more localized deformations and as a consequence consumes less time and memory.
3. This is the first system that allows the usage of heterogeneous sets of coordinates, and is able to define different coordinates for different cages and use them together in combination.
4. The ability to produce multi-level deformations, where different cages are used to control different levels of detail in the deformation of a model.

All this together gives rise to an extremely versatile deformation approach, which is much more intuitive and user-friendly.

4.2 *CAGES

As has been previously explained, current cage-based deformation approaches use a single cage to deform a mesh. Some of these methods, such as MVC, PMVC and HC, express a point p inside a cage c as an affine combination of the cage vertices v by:

$$p = \sum_{v \in c} w_c(v, p) v \tag{8}$$

where $w_c(v, p)$ are the coordinate basis functions.

Let us note that, while the use of the cage subindex c may seem redundant, its usefulness will become apparent in the following explanations. The natural way to define a deformation inside cage c of a point p is given by:

$$T_c(p) = \sum_{v \in c} w_c(v, p) v' \tag{9}$$

where v' are the deformed control cage vertices.

Instead, *Cages encloses a mesh in a connected set of controlling cages $\{c_0, c_1, \ldots, c_n\}$. These cages do not intersect and they use their own independent set of coordinates (See

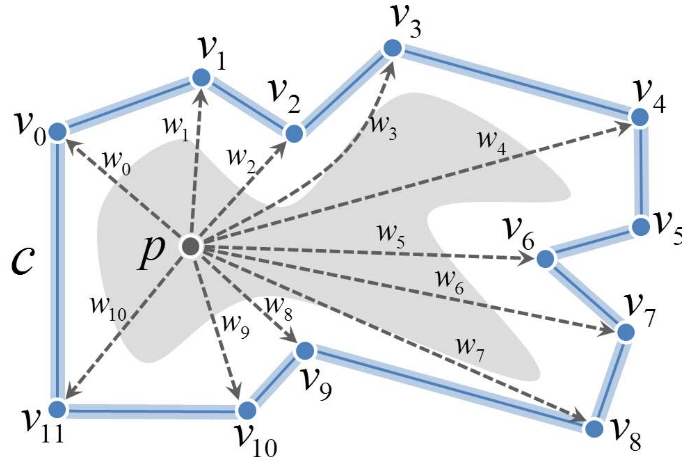


Figure 45: Definition of a point inside a cage by a set of weights respect to the cage vertices.

the left image in Figure 46 for an example). Let C be their union. For each of these cages, we can define a transformation $T_{c_i}(p)$. In general, the piecewise transformation defined on C by transformations $T_{c_i}(p)$ is at most C^0, e.g., it can generate first-order discontinuities across boundaries between adjacent cages. See Figures 48(b) and 48(d) where classic MVC/PMVC/HC and GC were used, respectively, to deform the cube shown in Figure 47. In the case of GC, $T_{c_i}(p)$ would use both vertices and face normals of the cages in its definition, but the corresponding piecewise transformation would be discontinuous at the cage boundaries. The insets in the images of Figure 48 show a detail of these discontinuities.

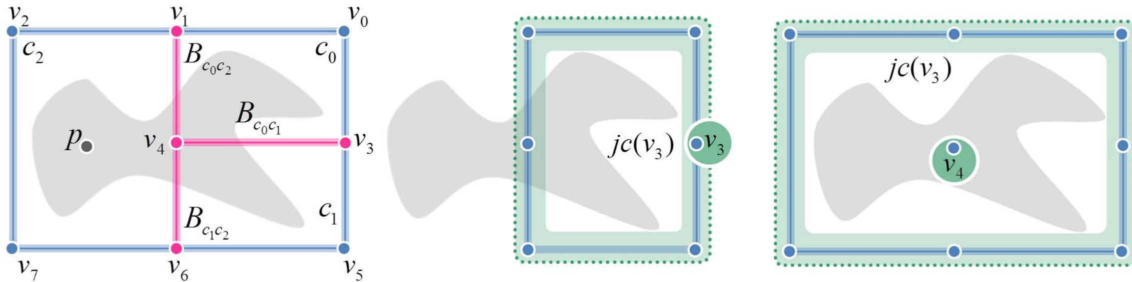


Figure 46: Left: $c_0 = v_0v_1v_4v_3$, $c_1 = v_3v_4v_6v_5$, $c_2 = v_1v_2v_7v_6v_4$, $B_{c_0} = B_{c_0c_1} \cup B_{c_0c_2}$, $B_{c_1} = B_{c_0c_1} \cup B_{c_1c_2}$, $B_{c_2} = B_{c_0c_2} \cup B_{c_1c_2}$. Middle: Join cage generated by v_3. Right: Join cage generated by v_4.

Therefore, if we want to use a set of cages as the base of our deformation tool instead of a single cage, it seems clear that we need to address this continuity problem. For that purpose we are going to define a *smooth transformation* S that will replace the classic $T_c(p)$ to deform the points. S will be piecewise defined on each cage c_i by transformations $S_{c_i}(p)$, so it is of class C^1 in the interior of C. Our proposal consists of defining the transformation $S_{c_i}(p)$ by blending the traditional transformation $T_{c_i}(p)$ defined in each cage c_i with a new transformation $J_{c_i}(p)$, called *join transformation*. $J_{c_i}(p)$ will be responsible for guaranteeing smooth transitions between neighboring cages and will behave similar to standard cage-based transformations.

More formally, we define the transformation $S_{c_i}(p)$ by:

$$S_{c_i}(p) = b_{c_i}(p)T_{c_i}(p) + (1 - b_{c_i}(p))J_{c_i}(p) \quad (10)$$

where $b_{c_i}(p)$ is the *boundary weight function* (See Section 4.2.2), which is a class C^1 function in c_i that is equal to zero at any point p lying on the border between c_i and any adjacent cage and different to zero at the interior of c_i. In this manner, $J_{c_i}(p)$ or $T_{c_i}(p)$ will be fully applied on a mesh point p depending on its position in respect to any border of c_i. The fact that transformation $T_{c_i}(p)$ is the one used by previous single-cage approaches will allow the user to choose a different coordinate system for each individual cage in a way that suits his/her needs. Thus, in the following subsections we will explain the two elements needed to define the smooth transformation $S_{c_i}(p)$: the *join transformation* $J_{c_i}(p)$ and the *boundary weight function* $b_{c_i}(p)$.

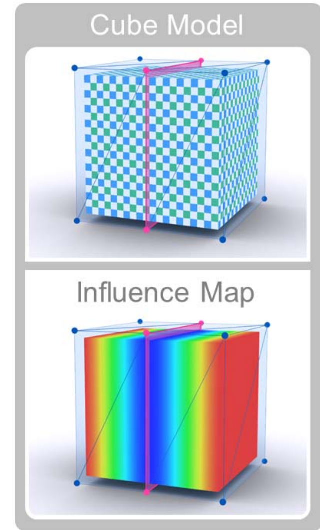


Figure 47: Cube model at binding time and its correspondence influence map.

4.2.1 *Join transformation* $J_{c_i}(p)$

First, let us introduce some definitions illustrated by the scheme shown in Figure 46. As mentioned before, *Cages uses a set of controlling cages. Given two cages c_i and c_j, we consider them adjacent if they share a set of faces. Then, given cage c_i let us denote its adjacent cages by $Adj(c_i)$ (e.g. $Adj(c_0) = \{c_1, c_2\}$ in Figure 46). For a cage $c_j \in Adj(c_i)$ let $B_{c_ic_j} = c_i \cap c_j$ be the border between c_i and c_j (e.g. $B_{c_0c_1} = c_0 \cap c_1$) and let the *boundary* of c_i, noted B_{c_i}, be the union of all borders $B_{c_ic_j}$ (e.g. $B_{c_0} = B_{c_0c_1} \cup B_{c_0c_2}$).

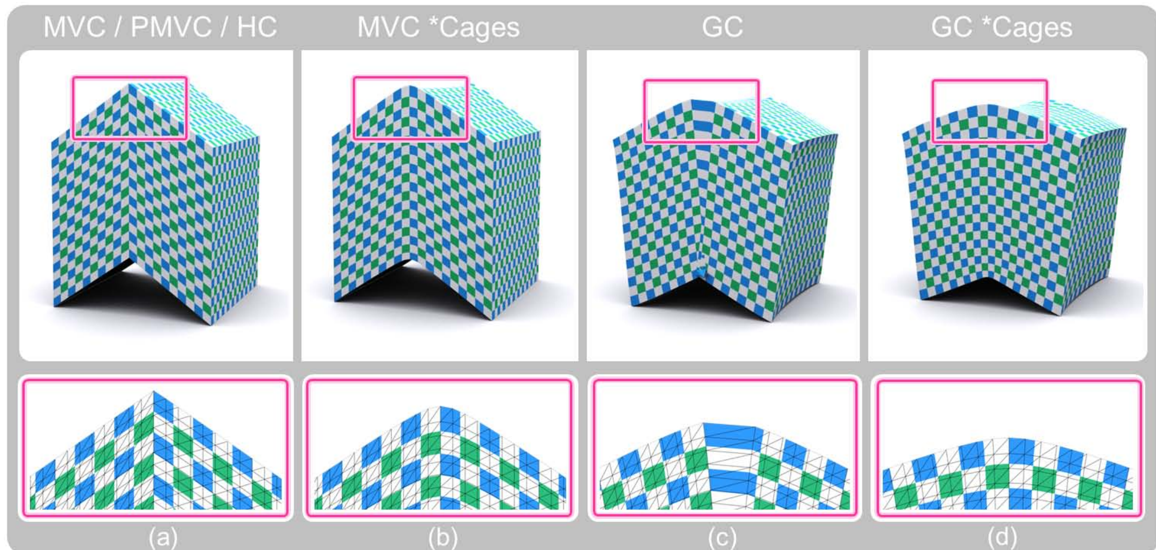


Figure 48: A comparison between piecewise deformations. (a) MVC/PMVC/HC deformation. (b) *Cages with MVC deformation, both for $J_{c_i}(p)$ and $T_{c_i}(p)$. (c) GC deformation. (d) *Cages with GC deformation, both for $J_{c_i}(p)$ and $T_{c_i}(p)$. The second row shows close views of the deformed model. Notice that only (b) and (d) are C^1.

Our goal is to define a transformation $J_{c_i}(p)$ that can smoothly cross boundaries between adjacent cages while verifying two important constraints: First, as $J_{c_i}(p)$ will be responsible for glue cage-based deformations it must behave similar to standard cage-based methods to produce fair deformations. Second, to provide a high degree of locality, $J_{c_i}(p)$ has to take into account only the local information concerning the boundaries between cages. As we will explain during this section, transformation $J_{c_i}(p)$ verifies both restrictions: First, $J_{c_i}(p)$ is built by using standard coordinates (MVC/GC), which allows not only a way to control the behavior of the blending region, but also a way to ensure the fairness of the deformations. Second, $J_{c_i}(p)$ keeps transformations as local as possible to the boundaries between cages, by relying only on their local information, which is their vertices.

So, given a boundary $B_{c_i c_j}$ between cages c_i and c_j, we define as *boundary vertices* the set of vertices that belong to that boundary $B_{c_i c_j}$. As can bee seen in Figure 46, any number of cages can meet at a *boundary vertex*. Thus, let us define the *join cage* of a *boundary vertex* v, denoted by $jc(v)$, as the union of all the cages incident at v. Figure 46 shows a set of control cages (left), the join cage of the vertex v_3 (middle), and the join cage of the vertex v_4 (right). As can bee seen, a *boundary vertex* may or may not belong to its associated join cage. For the former we will call them *non-interior vertices*, while for the later we will call them *interior vertices*. Relying on the concept of *join cage*, we will define, vertex-wise for each *boundary vertex* v, a smooth and local transformation $L_v(p)$. Then, for a cage c_i, the final *join transformation* will be defined as a blending of all the smooth transformations $L_v(p)$ related to the vertices of its boundary B_{c_i}.

Thus, in our scheme, the transformation $J_{c_i}(p)$ in c_i is defined by:

$$J_{c_i}(p) = \sum_{v \in B_{c_i}} W(v,p) L_v(p) \tag{11}$$

where weights $W(v,p)$ are normalized to 1. As an example, in Figure 46, note that

$$J_{c_2}(p) = W(v_1,p)L_{v_1}(p) + W(v_4,p)L_{v_4}(p) + W(v_6,p)L_{v_6}(p) \tag{12}$$

and

$$J_{c_0}(p) = W(v_1,p)L_{v_1}(p) + W(v_3,p)L_{v_3}(p) + W(v_4,p)L_{v_4}(p) \tag{13}$$

The weight function $W(v,p)$ will be responsible for measuring how much of the transformation $L_v(p)$ from each boundary vertex v is blended. If we are at v itself, then transformation $L_v(p)$ will be fully applied and its contribution will smoothly decrease as we move away. $W(v,p)$ will completely vanish at the other boundary vertices of B_{c_i}. Let us define the weight function $W(v,p)$ by:

$$W(v,p) = \Omega(v,p) I(v,p) \tag{14}$$

where $\Omega(v,p)$ and $I(v,p)$ are two smooth functions. The first, called *vertex influence function*, will provide us with a way to express the influence of vertex v in its join cage, while the second, called *interior vertices influence function*, will take into account the influence of *interior vertices*, if any, on the rest of boundary vertices (*interior* or *non-interior*). Both are explained below.

vertex influence function $\Omega(v, p)$. This function is a bell-shaped function defined over the join cage $jc(v)$ of vertex v, which allows us to specify a smooth region of incidence of vertex v on its join cage $jc(v)$. This function has to satisfy the following properties: be non-negative, be smooth, be equal to one at v and equal to zero on any face of $jc(v)$ not incident to v. As an example, in Figure 46, $\Omega(v_3, p)$ should be equal to one at v_3 and equal to zero on all the faces of $jc(v_3)$ except the two containing $v3$. One way to define $\Omega(v, p)$ could be based on Gaussian functions. However, the need to keep the transformations local has led us to define it by the use of a weight measure with respect to the faces of the join cage (a sort of distance). So, let t be a face of $F(jc(v), v)$, which is the set of faces of $jc(v)$ not incident to v. Thus, we define $\Omega(v, p)$ as a product of normalized and smoothed distances to that set of faces as follows:

$$\Omega(v, p) = \prod_{t \in F(jc(v), v)} f\left(\frac{d_{jc(v)}(t, p)}{d_{jc(v)}(t, v)}\right) \tag{15}$$

where f is a *smoothing* function that satisfies $f(0) = f'(0) = f'(1) = 0$, $f(x) = 1$ for $x \geqslant 1$ and $f'(x) \geqslant 0$. In our implementation we have used $f(x) = \frac{1}{2}\sin(\pi(x - \frac{1}{2})) + \frac{1}{2}$ for $x \in [0, 1]$.

The distance function $d_c(t, p)$ specified in cage c is defined by:

$$d_c(t, p) = 1 - \sum_{u \in V(t)} w_c(u, p) \tag{16}$$

where $V(t)$ are the vertices of face t and $w_c(u, p)$ are the coordinate basis functions (MVC/HC) used in cage c. In the case that c is not convex, weights $w_c(u, p)$ have to be computed by HC to prevent negative coordinate values. Otherwise MVC can be used. Observe that function $d_c(t, p)$ is of class C^1 inside c, it is equal to one on the faces of c non adjacent to t, and it is equal to zero on t.

Properties for $\Omega(v, p)$:

Notice that function $\Omega(v, p)$ possesses the required conditions and also satisfies:

- $\partial_p \Omega(v, v) = 0$, where ∂_p is the directional derivative respect to p
- $\partial_p \Omega(v, p) = 0$, for any point p lying on any face of $jc(v)$ not incident to v
- $\Omega(v, p) < 1$, for any point $p \neq v$

interior vertices influence function $i(v, p)$. Given a boundary vertex v (*interior* or *non-interior*), function $I(v, p)$ is responsible for introducing the influence of the rest of boundary vertices classified as *interior vertices* over v (e.g. $v4$ over v_3 in Figure 46). As we have explained, given the fact that function $\Omega(v, p)$ gives a way to determine the influence of a vertex v, we rely on it to define $I(v, p)$ as:

$$I(v, p) = \prod_{u \in Int(jc(v)) - \{v\}} f\left(\frac{1 - \Omega(u, p)}{1 - \Omega(u, v)}\right) \tag{17}$$

being $Int(jc(v))$ the set of interior vertices of $jc(v)$. Note that if the set of vertices $Int(jc(v)) - \{v\}$ is empty, this means that there are no interior vertices, so $I(v, p)$ is equal to one and thus $W(v, p)$ will be influenced only by the smooth function $\Omega(v, p)$. This means two things: first, the influence of vertex v is not affected by other interior vertices and second, the vertex v is only influenced by the rest of boundary vertices (*non-interior*) of $jc(v)$ which are taken into account in the first smoothing function $\Omega(v, p)$.

As an example, in Figure 49 we illustrate the weight function $W(v, p)$ on several different borders between two neighboring cages. In Figures 49(a) and 49(b), the border has only *non-interior vertices*, while on the rest of images a couple of *interior vertices* appear. In Figure 49(a) we show the weight $W(v_1, p)$ for vertex v_1 and in Figure 49(b) the weight $W(v_4, p)$ for the vertex v_4. Observe the smoothness of the weights and, in the later case, the lack of negative coordinates as we use HC. In Figure 49(c) we show the weight $W(v_6, p)$ of the *interior vertex* v_6 and in Figure 49(d) we show the weight of vertex v_1 but now with the influence of the *interior vertex* v_6. Observe the difference with the same weight of Figure 49(a). Finally, in Figures 49(e) and 49(f) we show the mutual influence of two *interior-vertices*.

Properties for $W(v, p)$:

The weight $W(v, p)$ is a non-negative function of class C^1 in $jc(v)$ satisfying the following properties:

- $W(v, v) = 1$ and $\partial_p W(v, v) = 0$
- $W(v, p) = 0$ and $\partial_p W(v, p) = 0$, for any point p lying on any face of $jc(v)$ not incident to v
- $W(v, u) = 0$ and $\partial_p W(v, u) = 0$, for any vertex $u \in Int(jc(v)) - \{v\}$

Now that we have set the weight $W(v, p)$, we still lack the definition of the transformation $L_v(p)$ to have the join transformation $J_{c_i}(p)$ completely specified. Thus, let us define transformation $L_v(p)$ differently depending on whether v is *boundary vertex* classified as an *interior* or a *non-interior* vertex of $jc(v)$:

- **v is a *non-interior* vertex of** $jc(v)$. Transformation $L_v(p)$ is defined by a smooth arbitrary transformation $T_{jc(v)}(p)$ (defined with MVC/HC/GC) in $jc(v)$ determined by the deformed vertices of $jc(v)$ by:

$$L_v(p) = T_{jc(v)}(p) \tag{18}$$

- **v is an *interior* vertex of** $jc(v)$. Transformation $L_v(p)$ is defined by a smooth arbitrary transformation $T_{jc(v)}(p)$ (MVC/HC/GC) in $jc(v)$, plus a smooth gradation of the transformation that moves $T_{jc(v)}(v)$ to v', which is the deformed cage vertex v. That is:

$$L_v(p) = T_{jc(v)}(p) + W(v, p)(v' - T_{jc(v)}(v)) \tag{19}$$

  Note that, as we get closer to the interior vertex v, the second term of the sum will increase, and thus the transformation $L_v(p)$ will be altered by adding the influence

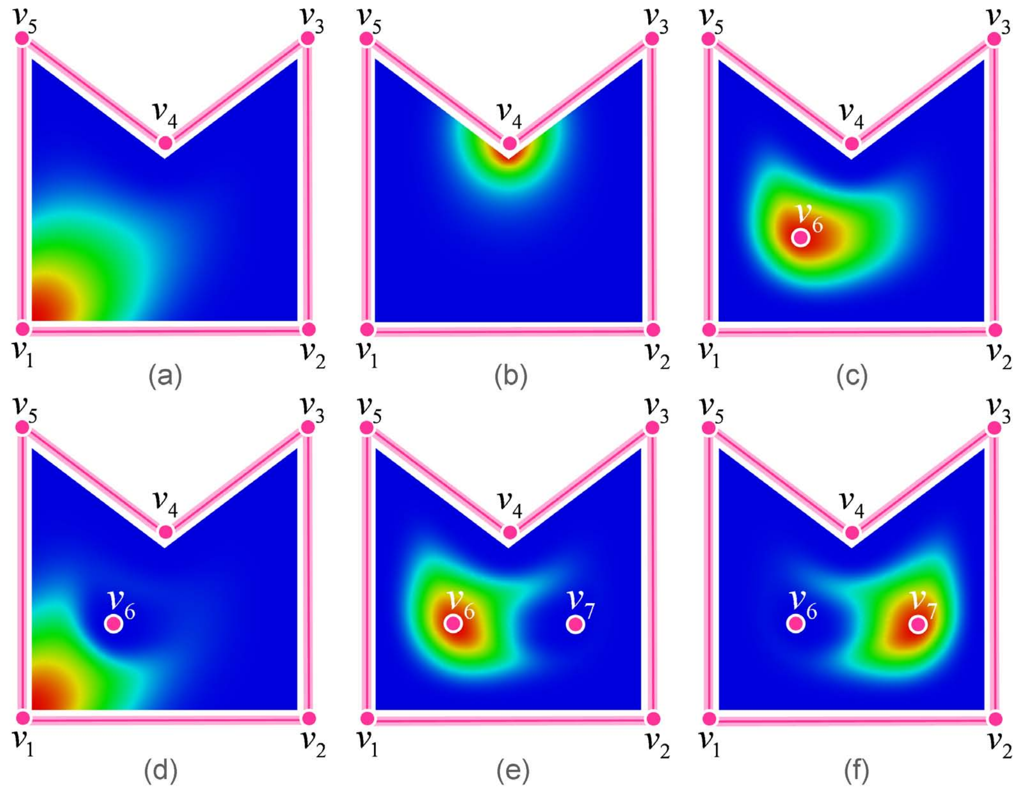


Figure 49: Variation of weight $W(v, p)$ on different borders between two cages.

of vertex v. Otherwise, as we get far away from the interior vertex v, the transformation of point p in $jc(v)$, that is $T_{jc(v)}(p)$, will be fully applied with no influence of v.

Properties for $L_v(p)$:

Observe that transformation $L_v(p)$ verifies the following properties:

- $L_v(v) = v'$ being v' the deformed cage vertex.
- $L_v(u) = u'$ for $u \in V(jc(v))$, where u' is the deformed cage vertex u and $V(jc(v))$ are the vertices of cage $jc(v)$.
- If $T_{jc(v)}(p)$ and v' are given by a linear function then $v' = T_{jc(v)}(v)$ and $L_v(p)$ also is given by a linear function.

Now that we have explained in detail the *join transformation* $J_{c_i}(p)$ through its two components, the weight $W(v, p)$ and the local transformations related to the boundary vertices $L_v(p)$, we can guarantee the following property with respect to its smoothness:

Properties for $J_{c_i}(p)$:

Observe that for a point p on a boundary $B_{c_i c_j}$ between cages c_i and c_j we have:

$$J_{c_i}(p) = \sum_{v \in B_{c_i c_j}} W(v,p) L_v(p)$$

which guarantees:

- $J_{c_i}(p) = J_{c_j}(p)$, $\partial_p J_{c_i}(p) = \partial_p J_{c_j}(p)$, and consequently smooth transitions between cages are obtained.

Note that $J_{c_i}(p)$ is not defined on the faces that do not have any boundary vertex (e.g face consisting of vertex v_2 and vertex v_7 of cage c_2 in Figure 46). As we will show in the following subsection, this is not problematic because transformation $J_{c_i}(p)$ will never be applied there, as the boundary weight function $b_{c_i}(p)$ will be equal to one and thus, $J_{c_i}(p)$ will not be applied.

4.2.2 *Boundary weight function* $b_{c_i}(p)$

In this subsection we specify the weight function $b_{c_i}(p)$ involved in the *smooth transformation* $S_{c_i}(p)$. Thus, let us define in a cage c_i the weight function $b_{c_i}(p)$ as a product of weights with respect to any border $B_{c_i c_j}$ as follows:

$$b_{c_i}(p) = f_{h_{c_i}}\left(\prod_{c_j \in Adj(c_i)} (1 - \sum_{v \in B_{c_i c_j}} w_{c_i}(v,p))\right) \tag{20}$$

where $f_{h_{c_i}}$ is the smoothing function defined in Eq. 15, parameterized with the parameter $h \in (0,1]$ for cage c_i. If $B_{c_i} = \emptyset$, i.e., when c_i does not have any neighboring cage, $b_{c_i}(p)$ is set to be 1.

Observe that $b_{c_i}(p)$ is equal to 0 when $p \in B_{c_i}$ and is equal to 1 on the faces of cage c_i that are not incident to any vertex of B_{c_i}. As an example, in Figure 46 the distance $b_{c_2}(p)$ of cage c_2 will be 0 when p belongs to the boundary of the cage ($B_{c_2} = B_{c_0 c_2} \cap B_{c_1 c_2}$) and to 1 on the face generated by vertices v_2 and v_7.

Properties for $b_{c_i}(p)$:

The weight function $b_{c_i}(p)$ verifies these properties:

- $\partial_p b_{c_i}(p) = 0$, for any point p satisfying $b_{c_i}(p) = 1$,
- $\partial_p b_{c_i}(p) = 0$, for any point p satisfying $b_{c_i}(p) = 0$, that is, lying on any face of B_{c_i}.

As can be seen in Formula (10), the weight $b_{c_i}(p)$ is a measure of the influence of the transformation $T_{c_i}(p)$ in $S_{c_i}(p)$, and can be adjusted by changing the parameter h_{c_i}.

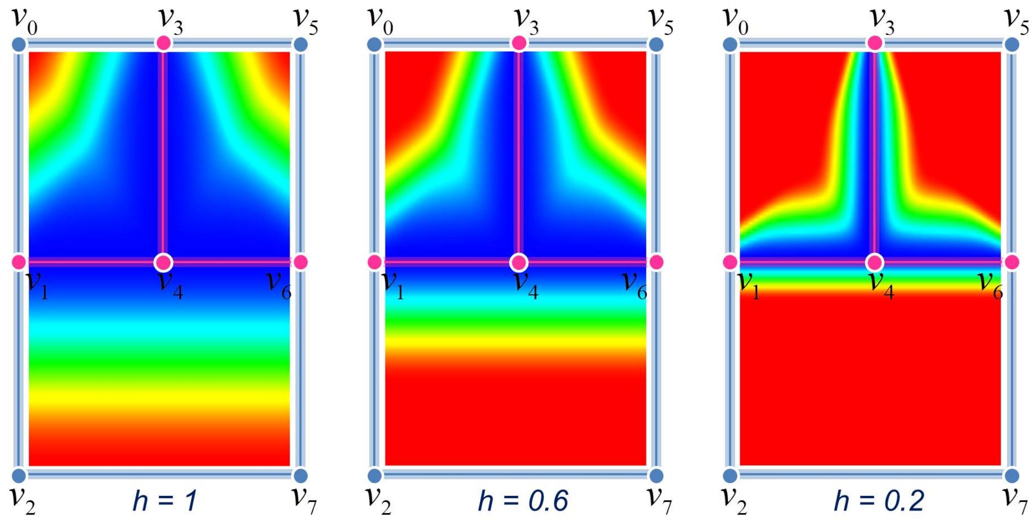


Figure 50: Influence map variation on the scheme shown in Figure 46.

To visualize the effect when altering h_{c_i} we use an influence map, where the model is painted in blue-red gradation according to the distance $b_{c_i}(p)$. As an example, in Figure 50 we visualize three different influence maps for the scheme shown in Figure 46. Also, in Figure 51 we show the chinchilla model enclosed in 9 cages. We have used GC for the ears and MVC for the rest of the cages, as well as the join transformations. At the right, results obtained from three different values of h_{c_i} corresponding to the left ear cage, are shown.

4.2.3 *Smooth Transformation* $S_{c_i}(p)$

Up to now, we have specified all the components that are part of the piecewise smooth transformation $S_{c_i}(p)$ defined in Formula (10). In Figures 48(c) and 48(e) we illustrate the effects of the proposed smooth transformation. In Figure 48(c) a deformation has been performed using MVC to compute both, the cage transformations $T_{c_i}(p)$ and the *join transformation* $J_{c_i}(p)$, while in Figure 48(e) GC are used.

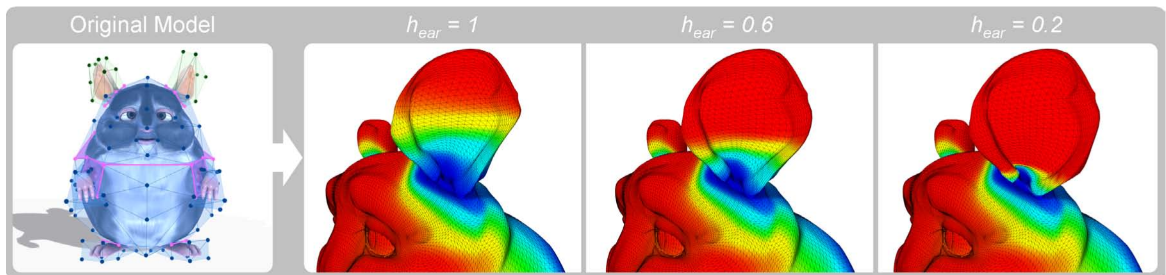


Figure 51: Influence map variation on the chinchilla model. Left: Original model and initial cages. Right: Results obtained by using different h_{c_i} values for the left ear cage. Red and blue regions mean transformations $T_{c_i}(p)$ and $J_{c_i}(p)$ respectively are fully applied.

Properties for $S_{c_i}(p)$:

The transformation $S_{c_i}(p)$ defined in a cage c_i satisfies the required continuity conditions for any point p on $B_{c_i c_j}$:

- $S_{c_i}(p) = J_{c_i}(p) = J_{c_j}(p) = S_{c_j}(p)$
- $\partial_p S_{c_i}(p) = \partial_p J_{c_i}(p) = \partial_p J_{c_j}(p) = \partial_p S_{c_j}(p)$

Now we want to show that the transformation S of the whole system of cages C preserves all the good properties of the standard cage-based techniques while satisfying the required continuity conditions. One important aspect is that transformation S inherits properties of transformations $T_{c_i}(p)$ and $L_\nu(p)$ (used to create $J_{c_i}(p)$) such as linear reproduction. The only requirement for the coordinate system used to compute these transformations is that it must be defined inside the respective cage as MVC/HC and GC. So, if A is an arbitrary linear function, we need to show that $S_{c_i}(p) = A(p)$ in case that $T_{c_i}(p) = A(p)$ and $L_\nu(p) = A(p)$ for all join cages. This can be easily demonstrated from the partition of unity property of the weight functions:

$$\begin{aligned}
S_{c_i}(p) &= b_{c_i}(p)T_{c_i}(p) + (1 - b_{c_i}(p)) \sum_{\nu \in B_{c_i}} W(\nu, p)L_\nu(p) = \\
&= b_{c_i}(p)A(p) + (1 - b_{c_i}(p)) \sum_{\nu \in B_{c_i}} W(\nu, p)A(p) = \\
&= b_{c_i}(p)A(p) + (1 - b_{c_i}(p))A(p) = A(p)\,.
\end{aligned}$$

Moreover, if transformations $T_{c_i}(p)$ perform boundary interpolation, transformation S performs boundary interpolation on faces not adjacent to any boundary between cages. For a point on this kind of faces all weights with respect to vertices of B_{c_i} are equal to 0. Consequently, the distance $b_{c_i}(p)$ with respect to B_{c_i} is equal to 1 and, then, $S_{c_i}(p) = T_{c_i}(p)$, which means that the *smooth transformation* in c_i is equal to the transformation (MVC/GC/HC) defined for cage c_i.

4.2.4 *Multi-level deformations*

We can use *Cages to build a multi-level system which gives flexibility, versatility, interactivity and control over the deformations to be applied to a part of the model. In our scheme, upper-level cages can own an arbitrary set of vertices of lower-level cages, the only restriction being that cages must have a hierarchical relationship (e.g. a *Directed Acyclic Graph* or a tree) and that a given cage vertex cannot be controlled by more than one parent cage. In general, two kinds of cages can be distinguished: *leaf-cages* that directly control the mesh and satisfy the conditions enumerated in Section 4.2, and the *internal-cages* that control cage vertices of lower-level cages and do not directly affect the mesh, so they can intersect and smoothness does not need to be enforced for them.

Our multi-level system relies on a simple yet effective observation: When a cage in the multi-level system changes, the effects of this change should only be propagated downwards but not upwards in the hierarchy. This means that, when a vertex ν of a cage is

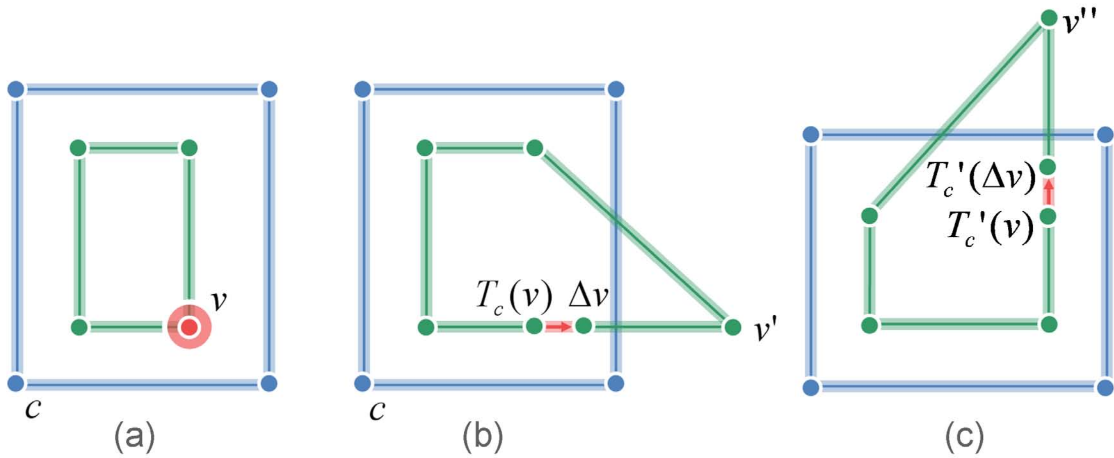


Figure 52: Multi-level deformation for coordinates not defined outside the cage. (a) Initial cages. (b) Direct cage vertex movement. (c) Parent transformation.

changed, the positions of all the vertices in the containing and neighboring cages should be updated as usual, but the parent cage c containing v would not be affected. However, if the parent cage c changes later on, v should be updated accordingly. Here, we cannot directly transform v, as it has a different position than the one used when computing its coordinates (binding time) with respect to c, so the coordinates for v must be recalculated, and then the process continues as usual.

If coordinates that are defined everywhere are used, such as MVC, then the above implementation works as described. However, in the case of coordinates *not* being defined outside (e.g. HC), special measures should be taken. We propose an easy but effective solution without the need of any cage recomputation. We can express any new position for v as $v' = T_c(v) + U(v)$ with $U(v)$ being the user-generated displacement, and $T_c(v)$ the transformation of v with respect to the parent cage c. We can express $U(v) = \lambda \cdot \Delta U(v)$, where λ is an adequate multiplicative factor and $\Delta U(v)$ is a displacement small enough to satisfy that point $\Delta v = T_c(v) + \Delta U(v)$ is within cage c at its current position. Now, if cage c undergoes another transformation T_c' that converts $T_c(v)$ into $T_c'(v)$ and Δv into $T_c'(\Delta v)$, we will update the current position of v by $v'' = T_c'(v) + \lambda(T_c'(\Delta v) - T_c'(v))$ (see Figure 52).

4.3 RESULTS AND DISCUSSION

Throughout the paper, we have used the following coloring code: Blue cages use MVC, red ones use HC and the green ones use GC. We have also drawn the boundaries between cages in pink. The implementation of *Cages has been carried out using the Ogre3D engine on a Quad Core Duo (2.83GHz) with 4GB of RAM. In the interests of fairness, we have implemented MVC/HC and GC in our unoptimized

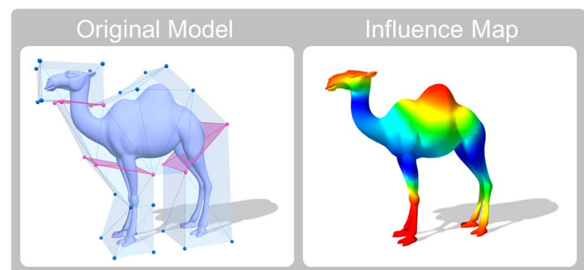


Figure 53: Camel model at binding time (left) and its corresponding influence map (right).

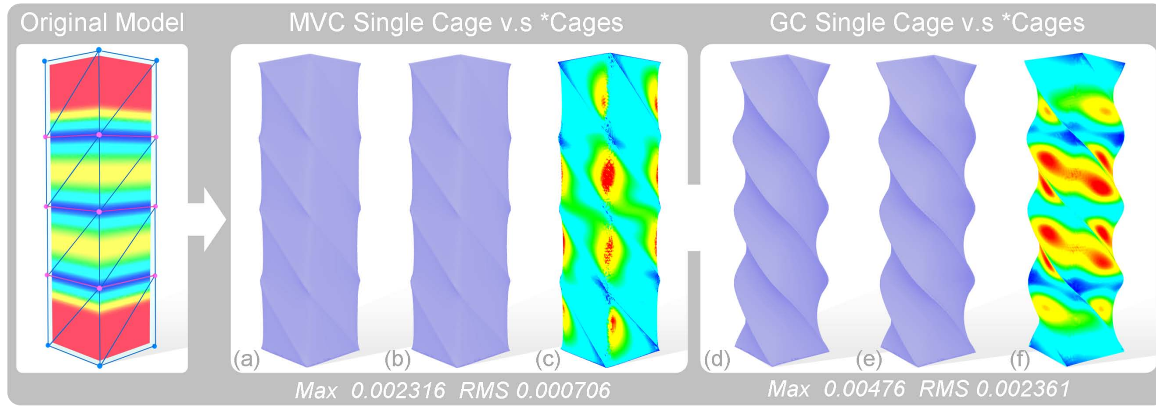


Figure 54: Twisting a prism using MVC (left) and GC (right). See the similarity between a single cage (a,d) and *Cages (b,e). (c,f) show the corresponding difference maps. Red means higher error and blue means lower error.

CPU-based system, carefully following the pseudocodes provided in the respective papers.

To show that the deformations produced by *Cages result in deformations with a level of quality on par with common single cage-based approaches, in Figure 54 and Figure 55 we present some comparisons. First, the prism model (206312 triangles) is enclosed using four cages (20 vertices) by *Cages and the union of cages by the single cage-based methods. The deformation is obtained by twisting the prism (each cage is rotated by $\pi/2$ with respect to the previous one). Note the large similarity between the results obtained by MVC and GC (Figures 54(a) and 54(d)) and *Cages (Figures 54(b) and 54(e)), even at the boundaries between cages where the new *Join Transformation* is fully applied. In the figure, both cage ($T_{c_i}(p)$) and *join transformations* ($J_{c_i}(p)$) have been computed with the same coordinate systems (MVC/GC). The corresponding difference maps are shown in Figures 54(c) and Figure 54(f). The maximum and RMS difference values shown are computed with respect to the bounding box of the model. Second, the camel model (see Figure 53) has been enclosed in four cages (36 vertices) by *Cages and again the union of cages by the single cage-based methods. See how similar are the results obtained by single cage approaches and our multi-cage technique. In the last column of Figure 55 we show the difference maps corresponding to each deformation. Observe the great similarity and how the differences are located in regions near the boundaries between cages, where our new *join transformation* is fully applied.

One important feature that *Cages has, in contrast to previous single cage-based methods, is the degree of locality obtained in the deformations. As a way of illustrating that important contribution, in Figure 57 we compare the locality of the deformations achieved by *Cages and single cage-based methods. For the former we have enclosed the train model (60052 triangles) in 4 cages (28 vertices) (see Figure 56), while for the latter we have used their union as a global cage for MVC and GC. As can be seen in the figure, single cage approaches (see Figure 57(a) and Figure 57(c)) cannot perform very localized deformations as all mesh vertices are deformed with respect to all cage vertices.

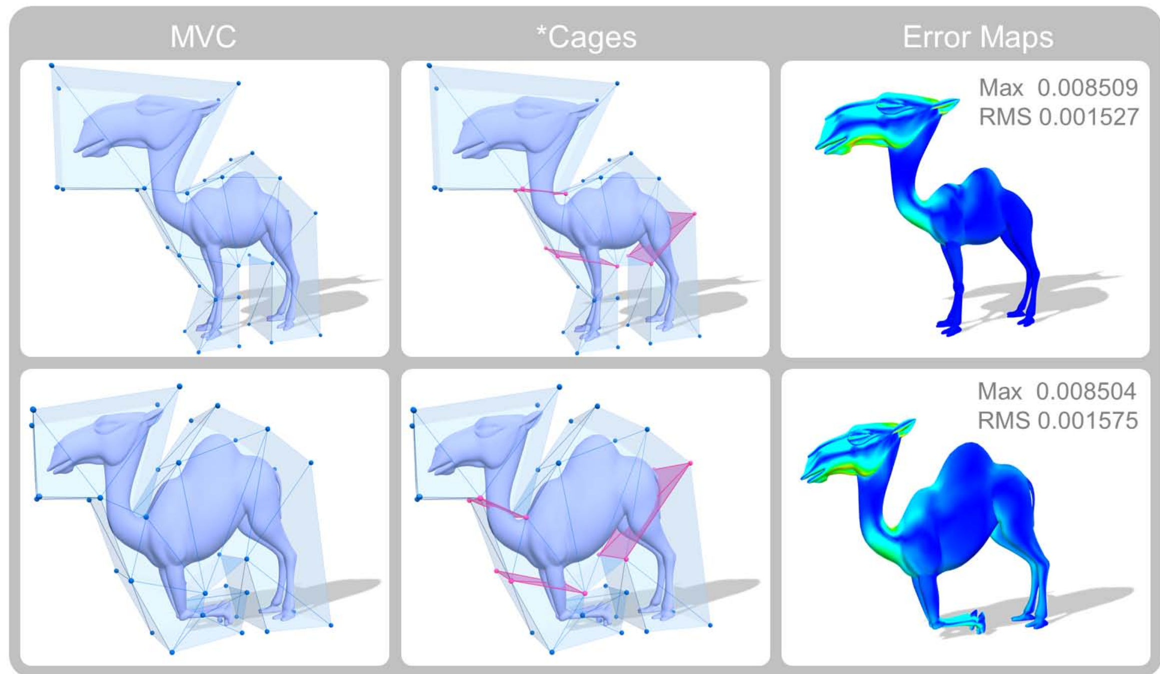


Figure 55: Two diferent deformations on the camel model with MVC (first column) and *Cages (second column). Third column shows the difference maps. Red means higher error and blue means lower error.

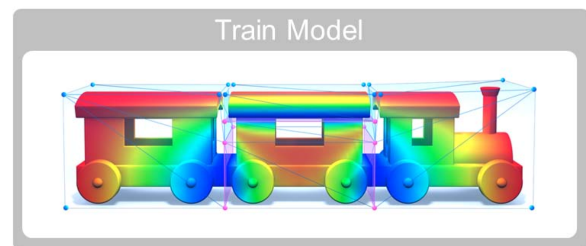


Figure 56: Influence map of the train model at binding time.

For instance, if the cage vertices of the top of the central wagon are moved, the single cage version of MVC (see Figure 57(a)) or GC (see 57(c)) deforms the head and tail wagons too, while *Cages (see Figure 57(b) and Figure 57(d)) only affects the center wagon, as one would expect. Also, notice the unsatisfactory deformations produced by MVC in Figure 57(a) at the sides of the wagon roofs due to the presence of negative coordinates produced by non-convex cages (the joining part between wagons). Let us note that, even though some coordinates like HC and GC present some kind of local control over the deformation, they have some issues we should discuss:

- HC uses *Interior Cages* to reduce the coordinate influence at the expense of building an extra interior cage, which leads to more memory and time consumption during the deformation. Moreover, the users need to be careful when creating such cages, because if the mesh goes through them, discontinuities will appear.

- In the case of GC, we could use the so called *Partial Cages* as a tool to provide local control. As discontinuities would arise at cage boundaries, the authors propose smoothing the deformation by extending the local deformation performed inside a partial cage to the rest of the mesh through some selected faces. This operation is not always possible and also results in more computation time.

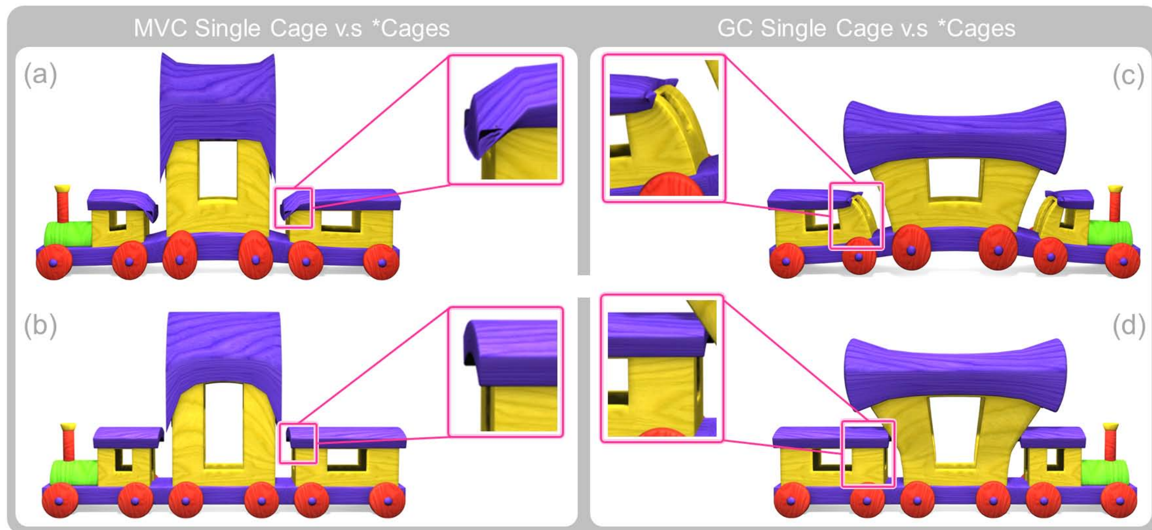


Figure 57: Locality: Comparison of single cage approaches and *Cages(a) single cage MVC, (b) *Cages whith MVC, (c) single cage GC, (d) *Cages with GC.

It is well known [26] [38] [54] that not only each of the existing coordinates produces different deformation results, but also that they have different properties and computational resource needs (e.g. GC produces more time consuming deformations as they take into account the faces of the cages, and HC needs much more time to compute coordinates for each vertex than MVC). Thus, the combination of different coordinate types in a unique framework is a useful feature that allows users to freely choose between coordinates and, as a result, they are be able to produce different deformations with the same cage configuration. Furthermore, coordinate selection allows to benefit from their good properties and concrete computational resource needs depending on the situation:

- As can be seen in Figure 58 and Figure 59, *Cages is able to smoothly combine different coordinate types between different cages. First, the butterfly model (61366 triangles) has been enclosed in 6 cages (114 vertices): two for each wing, one for the lower body, and another one for the head. At the right-most column, we can observe the combination of GC for the top wings and MVC for the bottom ones. Full MVC and GC deformation using *Cages can be seen at the second and third columns, respectively. Observe the difference between the deformations even when they use the same cage configuration, and how *Cages smoothly glues them. Second, another example of coordinates combination is shown over the elk model in Figure 59. There

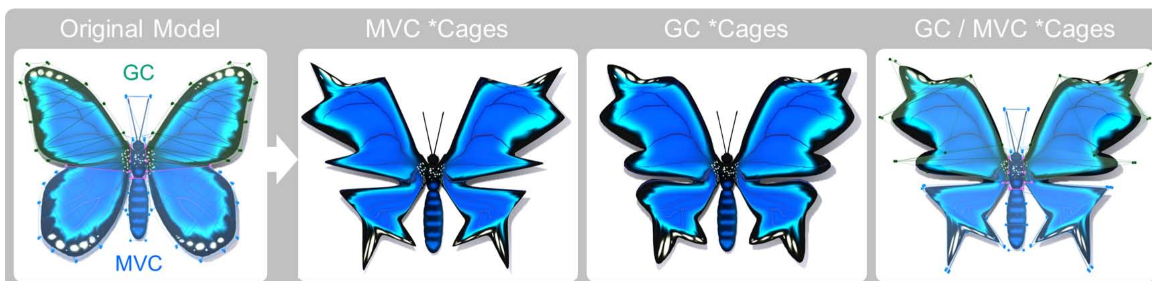


Figure 58: From left to right: the cages at binding time on the butterfly model, two different deformations using MVC and GC with *Cagesand a *Cagescombined GC/MVC deformation.

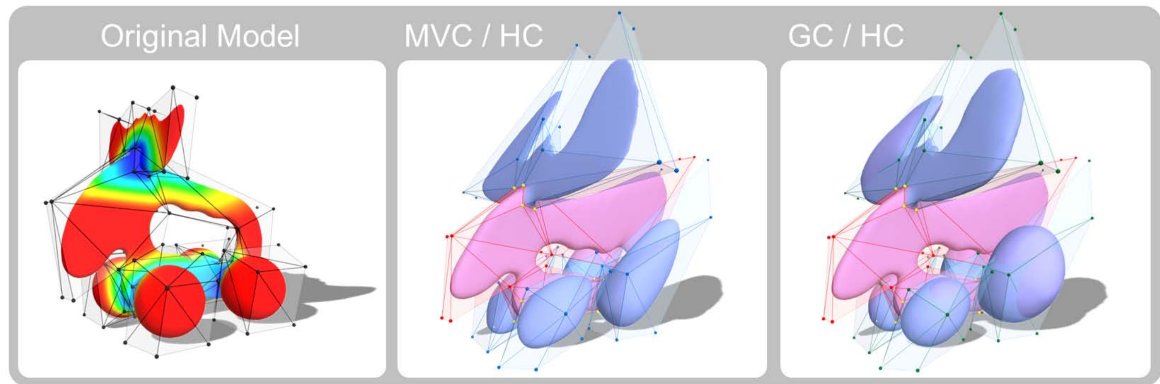


Figure 59: Combined deformations on the elk model. Left: Elk model at binding time with its influence map. Middle: *Cages combined MVC/HC deformation. Right: *Cages combined GC/HC deformation.

we can see again the differences of results when applying one coordinates or another with the same deformation allowing the user to obtain even more results simply by choosing one coordinates or another.

- *Cages also allows to benefit from the good properties of each coordinate type depending on the situation. In Figure 60 we show an example in which we can see the deformations produced by a single-cage with MVC on the hand model. As it is well-known, MVC has problems with concave cages because they result in negative coordinates (first column in the figure). By default, *Cages reduces this problem (second column) to regions near the boundaries when MVC is used to compute the *join transformations*. Here, the negativity still does not completely disappear because even when each single cage is convex, the resulting join cages are still concave. That is, cage transformations $T_{c_i}(p)$ do not have negative coordinates but *join transformations* $J_{c_i}(p)$ continue to be affected by the MVC negative behavior. However, the fact that *Cages is able to combine different coordinates, allows us to completely solve the problem (third column) by using HC to compute only the *join transformations* when concave join cages are detected. Although this situation is not solved in a direct way by our approach, the combination of coordinates localize the usage of HC to only the places where it is needed, and so consuming less preprocessing time and memory to produce a deformation free of negative coordinates.

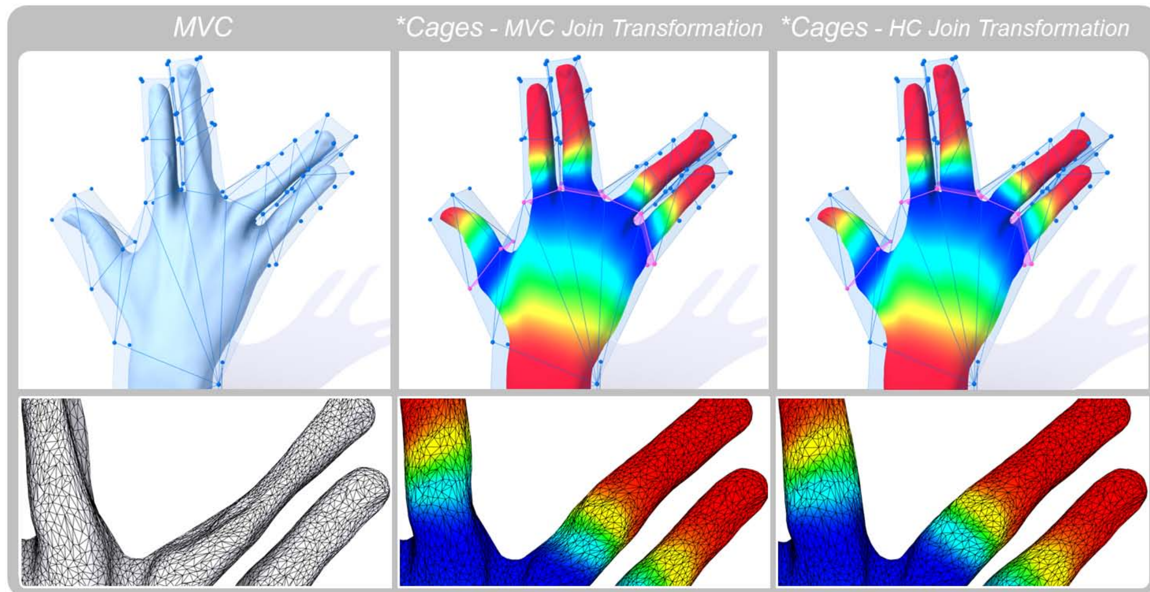


Figure 60: Deformations on the hand model. Columns from left to right: Deformation using MVC (observe the effect of negative coordinates of non-convex cages), and *Cages using MVC (third column) and HC (fourth column) as join transformations, respectively.

In situations where a combination of coordinates is used, an important aspect is how fair the resulting deformation is, especially in regions near the boundaries, which show a transition between different coordinate types. Given the nature of the propose blending scheme, we can guarantee that the resulting deformations will produce correct results, as we smoothly shift from one deformation type to another. As an example, in Figure 61 we show results of the fairness of the deformations produced by our approach when different coordinates are combined. In the top row, we can see the bumpy surface and its influence map at binding time. Three cages have been used: left and middle cages use MVC, while the right cage has GC. The join transformations applied between cages are of type MVC. In the bottom row we show a deformation using single cage approaches with MVC (left) and GC(middle), as well as the result of applying *Cages. See how the bumps in the model follow the faces of the cage when GC are used (see Figure 61, middle). On the other hand, if we perform the same deformation using MVC, the bumpy details get stretched (see Figure 61, left). This behavior can also be observed in the deformation produced by *Cages as well as the fairness in the deformation over the transition between cages of different coordinate types (see the insets in the bottom row in Figure 61). Observe how the GC deformation shifts to the MVC one.

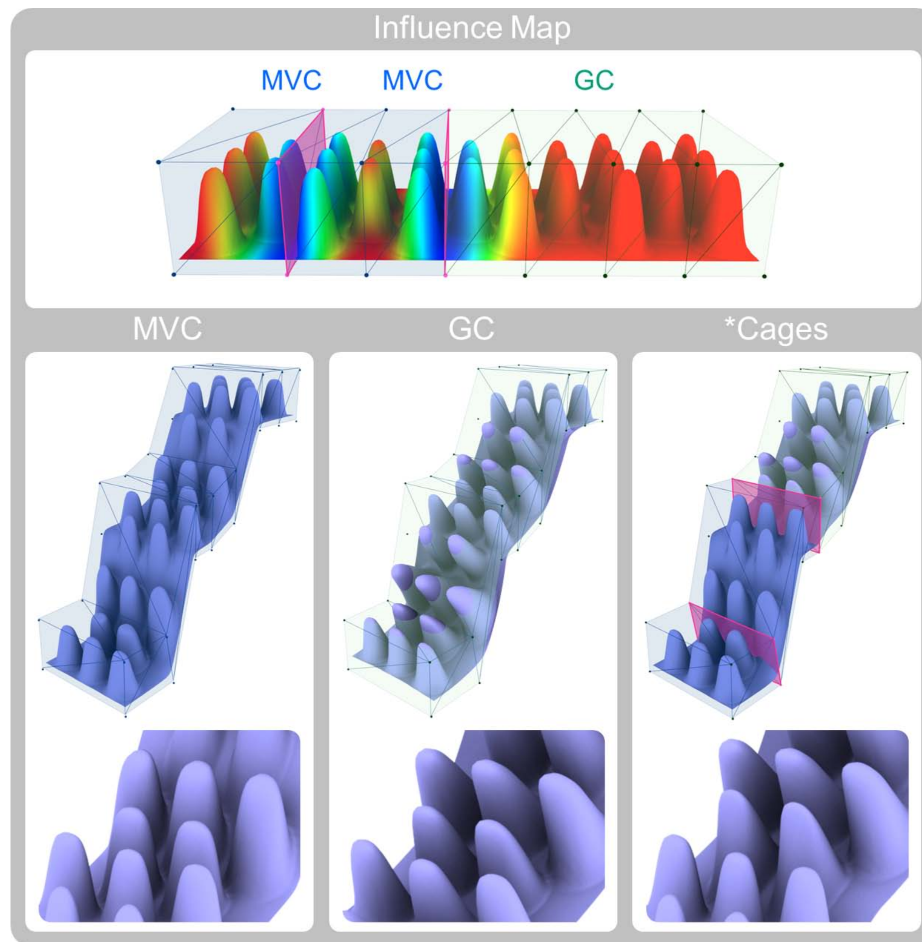


Figure 61: Fairness of the resulting MVC (left), GC (middle) and MVC/GC with *Cages (right) deformations. The top row shows the influence map and the cages at binding time.

*Cages is able to handle any number of cages meeting at a boundary cage vertex. Figure 62 shows a deformation obtained from a flower model (21903 triangles) enclosed in 13 cages (88 vertices) using different coordinate types. Here it is important to observe the correct behavior of the method even when cage vertices with more than two incident cages exist.

Two different multi-level deformations (see Section 4.2.4) can be seen in Figures 63 and 64. On one hand, in Figure 63, the squirrel head model (19552 triangles) has been enclosed by four leaf-cages (66 vertices): teeth, face, left and right ear. There are also two internal-cages (16 vertices), which are colored in grey: The global ears cage, which encloses some vertices of the left and right ear cages, and the head cage, which encloses all unbinded vertices of the previous cages (see Figure 63(a)). The sequence of deformations in the figure is as follows: First the teeth have been deformed in Figure 63(b), second the ears in 63(c), then the entire head in 63(d) and finally the face in 63(e). On the other hand, we have also applied our multi-level approach over the whole squirrel model (37588 triangles), as can be seen on the right of Figure 64. The model has been enclosed in 12 *leaf-cages* (132 vertices) and 3 *internal-cages* (24 vertices), shown at the top-left side of the image. Also, observe the usage of different coordinates for different cages in addition to the deformations produced at different levels of detail. As can be seen, we are able to perform

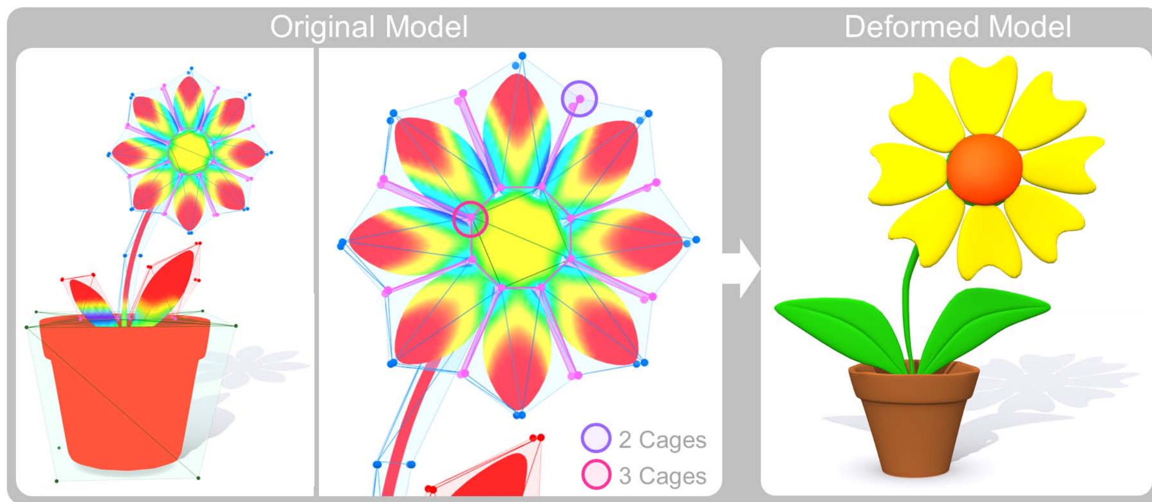


Figure 62: Multiple cages meeting at a cage vertex. Left: Original model with 13 cages using different coordinates. Middle: close view. Right: Deformation with *Cages.

deformations at different levels of detail without recomputing any coordinate for the mesh vertices, which is something that none of the existing single-cage-based approaches allows. As an example, imagine we have a *partial cage* with HC for the ear of the squirrel model in Figure 63 and a single cage containing the whole model. One could compute coordinates for each cage, then deform them using the *partial cage* and then the single cage. What happens here is that, when deforming using the later, the coordinates won't be valid as the vertices affected by the partial cage have seen modified their binding positions. So, we must recompute coordinates each time we go from the usage of partial (local) to single (global) cages and vice versa.

In contrast, *Cages can change the level of detail over the deformation without the need for any coordinate recomputation for mesh vertices, as they are controlled by the leaf-cages. Finally, in Figure 66 we show an scene resulting from applying a multi-level deformation over squirrel and chinchilla models with the use of different coordinates.

As has been explained in Section 4.2, *Cages naturally supports the presence of *interior cage vertices*. In Figure 65 (left) we show a "Easter egg" model enclosed in a grid of twelve cages. This set of cages generates two interior vertices, one at the top and the other at the

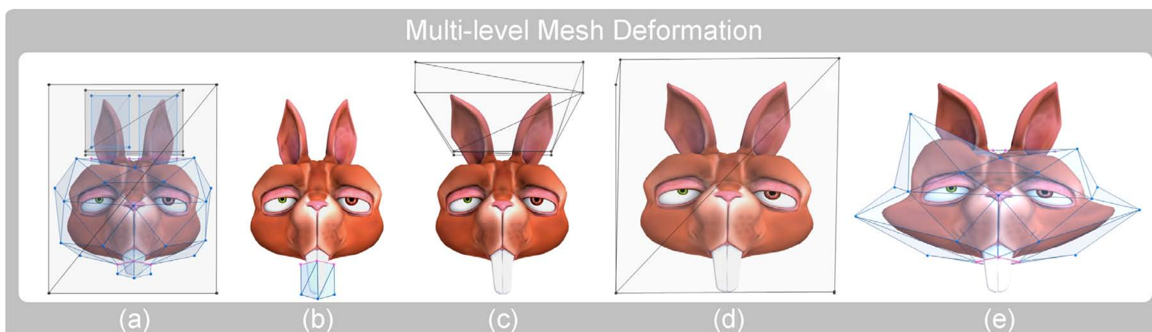


Figure 63: Multi-level deformation of the squirrel model. (a) Multi-level cages. (b) Leaf deformation: teeth cage. (c) Internal deformation: ears' cage. (d) Internal deformation: head cage. (e) Leaf deformation: face cage.

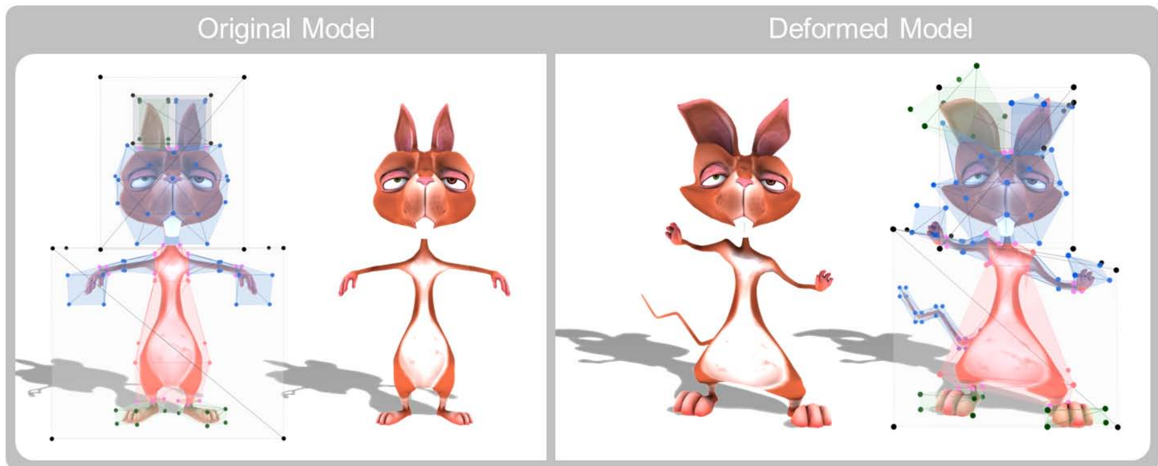


Figure 64: Deformation of the squirrel model using *CagesĿeft: The model and its multi-level cages at binding time. Right: Composition of a pose.

bottom of the "surprise" mesh. As a way to demonstrate the good behavior of deformations when these kinds of vertices are involved, we have generated two different deformations: the first deformation has been obtained by moving up the top interior vertex and leaving the rest of cage vertices stationary (Figure 65(a)), while the second deformation has been achieved by moving the top row of cage vertices up and leaving the rest in their original positions (Figure 65(b)).

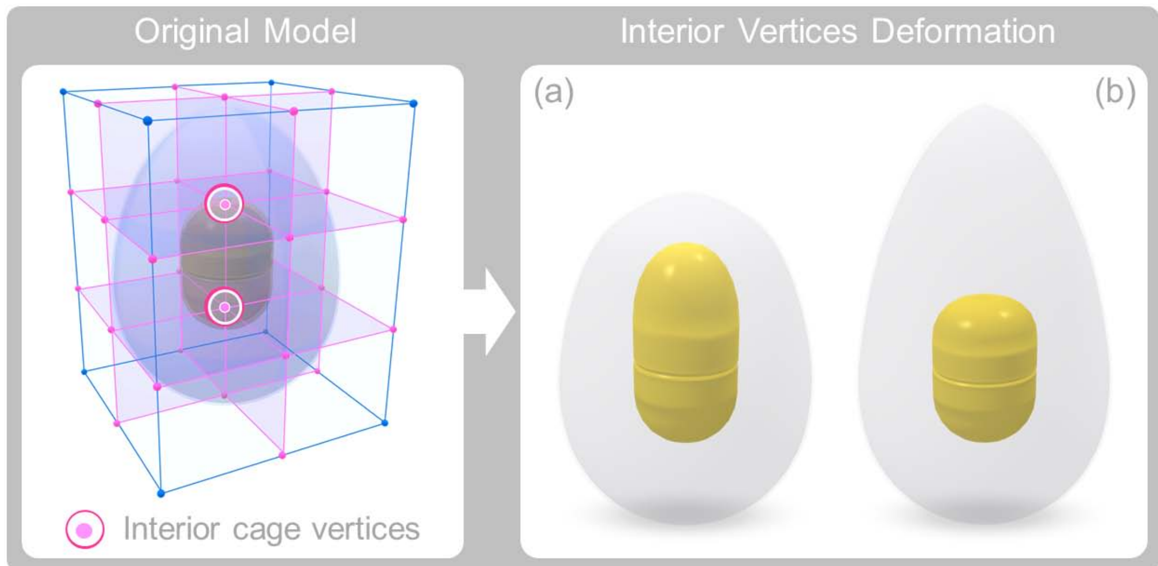


Figure 65: Deformation involving interior points of the "Easter egg" model using *CagesĿeft: The model and the grid of cages at binding time. Highlighted vertices are interior points. Right: Composition of two different deformations.

Figure 66: Scene showing the multi-level deformations on the Squirrel and Chinchilla models. Left: Cages at binding time with different coordinates (Blue - MVC, Green - GC, Red - HC, and pink cage boundaries). Right: Composition of different poses.

Given the multi-cage nature of *Cages it allows to naturally reduce the number of weights stored for each mesh vertex and, as a consequence, to obtain faster evaluations. The computational and memory costs become optimal when the number of cages used is high and they have a small adjacency degree, that is, when one cage is connected to a reduced number of neighboring cages, as in Figure 67. The small adjacency degree gives as a result a small number of *join transformations* involved in the final *smooth transformation* of every single cage, and consequently faster and lower memory-consuming deformations.

Despite this, it is important to mention that the user can control the influence of *join transformations* inside a cage by adjusting the influence map parameter h_{c_i}. In Table 1 we show the memory and time requirements depending on the influence of the parameter h_{c_i} in the boundary weight function $b_{c_i}(p)$ (see Section 4.2.2). We compare the results obtained for three different h_{c_i} values for the set of 9 cages used (134 vertices) in the chinchilla model (140126 triangles), both for MVC and GC. In these cases, cage and *join transformations* have been computed with the same coordinate types. Observe that the memory usage and the computational cost are nearly proportional to h_{c_i}. This is because, as the h_{c_i} values decrease, transformations $T_{c_i}(p)$ are fully applied on more mesh vertices, and for these the *join transformations* $J_{c_i}(p)$ do not need to be stored and computed. As can bee seen, the parameter h_{c_i} has a drastic impact on *Cages requirements, but using an insufficient value for h_{c_i} could introduce visible non-smooth transitions in extreme deformation conditions. Also, it should be noted that the value for h_{c_i} can be set in an easy and independent manner for each border, for each cage or for the whole model. In Figure 51 we have used the second approach, while for all the tables we have used a single h_{c_i} value for the entire model to make comparisons fairer. In our system, the user is provided with a simple slider to control this parameter independently for each selected cage.

In Figure 67 we show three different deformations applied over the Sintel model (66845 triangles), which has been used in real film production [28]. Note that the model has 15 leaf-cages (193 vertices), as can be seen on the left part of the image. Different coordinates have been used for the different cages to obtain the deformations shown. Note the good results obtained by our approach even when several types of coordinates are used in such complex deformations.

In Table 2 we compare *Cages with MVC and GC on the Sintel model of Figure 67 and on the squirrel model (12 leaf-cages with 132 vertices and 3 internal-cages with 24 vertices) of Figure 64. Observe that *Cages consumes less than half the memory for the squirrel model and 4 times less than the memory for the Sintel model (column 2). The total time required for the preprocess is shown in column 3, specifying the amount of time dedicated to compute the coordinates with respect to the parent cages. Also, *Cages takes much less time to compute cage coordinates because each of the cages used are simpler and smaller than a whole single cage. The rest of the time is needed to compute join cages and the coordinates with respect to them. In the case of using GC, *Cages requires even less preprocessing time because of the nature of their computations [54]. The deformation times (column 4) are the averages of the times needed for the deformation of a cage vertex. Observe that our approach is significantly faster for both models, where we achieve between 3 and 5 times the speed of MVC, and between 7 and 18 times that of GC.

Figure 67: Deformations of the Sintel model (66845 triangles) using *CagesLeft: Cages at binding time with different coordinates (Blue - MVC, Green - GC, Red - HC, and pink cage boundaries). Right: Composition of different poses.

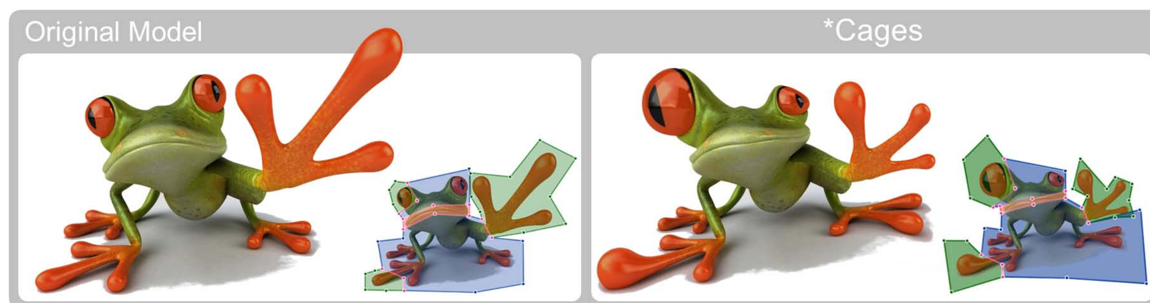


Figure 68: Deformation of the frog model. Left: Cages at binding time and original model. Right: Deformed cages and model using *Cages.

We would like to emphasize that, even our code is unoptimized and CPU-based, *Cages allows for a more GPU-friendly implementation than single cage-based approaches do, as it has a much lower number of weights to store for each mesh vertex. Moreover, unlike the technique presented by Landreneau and Schaefer [44], we don't need to be constrained by having to create new deformations that must be similar to an initial range of predetermined poses to be able to reduce memory and time consumption. Instead, we give the user the freedom to perform any type of deformation while also keeping the memory and time requirements small, as well. Let us note that *Cages is fully compatible with the work by Landreneau and Schaefer [44], and our computational requirements could be reduced even more if used together: Their compression could be used for both cage and join transformations. The latter case would benefit *Cages the most, as join transformations are more computationally demanding to evaluate than regular cage transformations.

*Cages is not related with the modeling of the cages themselves. As the examples throughout the paper have shown we use a set of individual cages, the union of which result in a single cage for the entire model. This has been done as a way to make comparisons to previous single cage-based approaches fairer. With *Cages we don't need to create the whole set of cages that are equivalent to a single cage. For instance, if we want to deform only the head of the Sintel model shown in Figure 67, we are not required to build all the cages shown there, we only need to model the ones needed to make this task simpler. Moreover, the modeling of cages used to deform a small region is usually easier and faster and, as a consequence, the use of many cages to deform a mesh can result in a more user-friendly element for the cage-modeling phase.

As a space deformation approach, *Cages can be used in the same domains as previous methods. For instance, the lowest-level cages of our hierarchy could be deformed by a simple skeleton, as Ju at al. [40] did. Thanks to the local behavior of our approach, we could provide a finer degree of control over the skeleton and, as a result, a smoother final animation. *Cages also can be used to perform deformations in 2D (see Figure 68), as long as the cages satisfy the requirements described in Section 4.2. *Cages is a cage-based method that can be also integrated with other deformation techniques that uses other types of handles, as the one proposed by Jacobson et al. [37]. For instance, our approach can be used in a certain region of the model to perform local and hierarchical deformations with MVC/HC or GC. Then, on the rest of the model, the bounded biharmonic weights could be used with point and bone handlers. *Cages would be responsible for smoothly gluing

the deformations provided by cage-based methods (MVC/HC/GC) with those produced with the technique of Jacobson et al. [37], thus avoiding the discontinuities that would appear through cage boundaries.

Chinchilla	**Memory**	**Preprocess**		**Deform**
	(MB)	Cage Coord.	Total	(sec)
MVC *Cages				
$h_{c_i} = 1.0$	45.45	3.8045	29.4625	0.3308
$h_{c_i} = 0.6$	29.75	3.7958	17.3009	0.1409
$h_{c_i} = 0.2$	14.05	3.8034	9.5280	0.0662
GC *Cages				
$h_{c_i} = 1.0$	131.27	10.1461	134.8266	1.0725
$h_{c_i} = 0.6$	84.35	10.3721	73.5973	0.4460
$h_{c_i} = 0.2$	37.43	10.2712	49.1273	0.1888

Table 1: Memory and time requirements for several h_{c_i} values for the chinchilla model.

Model	**Memory**	**Preprocess** (sec)		**Deform**
	(MB)	Cage Coord.	Total	(sec)
Squirrel				
MVC	25.55	7.4815	7.4815	0.0904
MVC *Cages	11.26	3.4861	6.6197	0.0372
GC	63.30	39.7976	39.7976	0.7095
GC *Cages	30.39	8.3214	28.9178	0.0969
Sintel				
MVC	52.52	20.1988	20.1988	0.2105
MVC *Cages	13.54	8.3469	13.7599	0.0427
GC	155.41	107.4353	107.4353	1.8824
GC *Cages	34.94	16.3395	47.9066	0.1074

Table 2: Memory and time requirements for the Sintel and Squirrel models using *Cages and standard single cage approaches.

4.4 CONLUSIONS

In this fourth chapter we have presented *Cages a multi-level (i.e., hierarchical) cage-based system for spatial mesh deformations. It allows the combination of heterogeneous sets of coordinates, allowing the user to define different coordinates for different neighboring cages and smoothly use them together in combination while preserving their properties (e.g. linear precision and boundary interpolation). With *Cages any change the user makes in one cage is kept local to the cage being modified. This is one of the main advantages with respect to other mechanisms that try to obtain more localized deformations. Moreover, *Cages allows the local use of any coordinate, even those that do not allow such usage when used in isolation, as long as they are defined inside the cage. Thus, in this sense, we can consider single cage approaches as a particular case of *Cages.

5 CONTINUITY AND INTERPOLATION IN SCREEN SPACE

Vive! Disfruta! Avanza! Decide! Y sonríe!

In previous chapters we have introduced two techniques that allowed to avoid discontinuities in both texture (2D) and object spaces (3D), giving as a result new and better ways to generate smooth deformations of continuous textured meshes. Once we have a well textured model in the right pose for a certain scene, what remains is to render it, preferably faster than current state of the art approaches while holding the image quality. In Computer Graphics, acceleration techniques for Rendering in general, and Ray Tracing in particular, have been subject of much research. Most efforts have been focused on new data structures for efficient ray/scene traversal and intersection with the scene. In this chapter we will present a two-stage rendering acceleration technique called *I-Render*: First, a pre-processing clustering stage that builds upon information theoretic channels to group triangles by their similar features. The clusters of triangles will create regions of smooth variations and, as a consequence, regions that will be candidate to be interpolated when rendered. Boundaries between clusters will define sharp transitions between features and so regions that need to be accurately preserved. Second, an approximate rendering stage that uses the clustering information to decide which areas of the final image could be interpolated and which require more involved calculations. This process will be carried out in a iterative way, starting on a low-resolution render, and then successively refining it up to the desired resolution (final image). That way we will reuse previous results and avoid costly computations. As we will show in the results obtained, we are be able to speed up the rendering up to 8 times. The actual improvement will depend on the complexity of the per-pixel calculations, the screen-size of the objects and the number of clusters. We will also provide some parameters to fine-tune the rendering quality of the final image. Moreover, we will show that our technique supports a range of popular and costly techniques, going from texture mapping up to complex ambient occlusion and soft shadow calculations, and even it can be used in conjunction with more traditional acceleration methods, making it a flexible and easy approach for being integrated in current rendering pipelines.

5.1 introduction

The problem of efficient image generation has been a cornerstone in research since the earliest days in Computer Graphics [27]. Ray Tracing is one of the most popular techniques when generality, quality and ease of implementation comes into account, being able to handle most optical effects [64]. Most of the acceleration methods in Ray tracing involve *primary rays*, because they present a high level of coherence that can be exploited to speed up ray/scene traversal/intersection and thus, the final image generation. That is, given a scene, a pixel and its associated ray, its neighboring pixels will probably hit a similar region in that scene. However, other complex shading operations, e.g. ambient occlusion or soft shadows, can be really expensive to compute because they involve *secondary rays*, which usually lose most of the coherence that primary rays have, considerably hindering rendering performance. As a result, sometimes rough approximations, simplified calculations or other tradeoffs need to be used to accelerate computations.

If we observe most of the scenes used in production, we can arrive to the realization that in general there are low-variability regions that share common characteristics or information, an thus they could be computed more efficiently than calculating each pixel from scratch. Taking into account this observation, in this chapter we are going to build a new rendering strategy that uses the similarity between the elements of a scene to provide an approximate and fast rendering approach. We call this technique *I-Render* and it will consist of two main stages:

- **Clustering stage**. First, a pre-processing stage will use information theoretic tools to define channels that will allow clustering the scene geometry by their most relevant features like visibility, orientation, or texturing.
- **Rendering stage**. Once the geometry has been clustered, the runtime part will use the clustering information to approximately compute the final image in a series of passes. It will begin with a low-resolution buffer and iteratively it will increase the resolution up to the final image size. At each pass, low-resolution samples from previous passes will be reused in order to obtain, wherever possible, an interpolated value for the samples in the new resolution image. If it is not possible, for instance at cluster boundaries where a discontinuity between features exist, the samples will be evaluated and reused farther on in future passes. This process will result in a considerable reduction in the number of computed evaluations. All the intermediate images that we will use until we reach the final one, are given the name of *I-Buffer*, a pyramid-like data structure that will store the information used for interpolation at each resolution, and hence the name of the general technique: *I-Render* (Interpolation of elements sharing similar information provided by the clusters).

As we will show, one important feature that we will introduce is an automatic control mechanism of the number of passes (intermediate images), such that performance of our technique has a lower bound in the performance of traditional Ray Tracing, resulting in a win-win situation. As an example, that way, we will be able to avoid situations in which we could have a low level of interpolation ratio due to having very complex geometries with no smoothness in any of their features or having a model projected by a small set of pixels in the screen. From the point of view of quality, the user will be able to have a fine-grained control of the final rendering by selecting the features involved in the clustering

and their degree of fidelity during rendering, by easily controlling the clustering thresholds provided in the preprocessing stage. Finally, but not less important, we will show that memory usage requirements for our approach are on the same level as traditional Ray Tracing techniques.

In summary, in this chapter we present a novel technique that provides a number of important contributions:

- A mesh-clustering framework based on Information theoretic tools. In particular, we define clustering criteria based on geometry visibility, orientation and texture stretching, but other user-definable parameters could be used, too.
- A multi-pass progressive rendering strategy based on the reuse and interpolation of previously-computed results.
- A controlling mechanism to guarantee traditional Ray Tracing as a lower bound to the rendering speed.
- Our technique can accommodate both static and animated scenes as well.

5.2 CLUSTERING

The objective of this pre-processing stage is to group the input geometry according to a user-defined set of features. Our input mainly consists of a triangle soup with connectivity information (i.e., we do not require any special structuring). For each triangle we may have any number of associated attributes (e.g., color, normal, etc.).

We first define a number of information theoretic channels, each one representing a user-defined feature (See Section 5.2.1). The algorithm for each feature uses its respective channel plus a clustering strategy to group triangles into an homogeneous cluster of similar elements. Observe that this "homogeneity" is only with respect to its given feature, and that the resulting triangle groups might not be homogeneous with respect to a different feature (See Section 5.2.2).

The algorithm performs an iterative hierarchical processing over the mesh, progressively obtaining finer clusterings after each step: the clusters generated at the previous step using a given channel are fed as independent meshes to the clustering algorithm in the next step (See Algorithm 3). It is easy to realize that, after each iteration, the input geometry is partitioned into a set of clusters, and each cluster is at most as large as the input. Thus, each iteration operates over smaller sets of triangles, which results in a reduced cost and faster computations. Finally, as the resulting clusters present jaggy edges, we apply a boundary-smoothing algorithm respecting the feature, improving rendering quality and speed (See Section 5.2.3).

5.2.1 *Information Theoretic Channels*

As mentioned above, our clustering strategy consists of detecting homogeneous and smooth areas with respect to some user-defined criteria or feature. In order to evaluate the similarity among triangles using that criteria, we use information theoretic tools. Let us briefly

Algorithm 3 Clustering algorithm.

```
1: clusters = [mesh]
2: for each channel ch in channels do
3:   newClusters = [ ]
4:   for each cluster cl in clusters do
5:     ch.init(cl)
6:     ch.cluster()
7:     ch.smoothBoundaries()
8:     newClusters += ch.getClusters()
9:   end for
10:  clusters = newClusters
11: end for
12: return clusters
```

emphasize in some key concepts about information theory have been previously discussed in Chapter 2.

Let us define X and Y both as a discrete random variables with their respective probability distributions. Let $p(y|x) = Pr[Y = y|X = x]$ be the conditional probability of an element $y \in Y$ given and element $x \in X$. Then, we can define an information channel $X \to Y$ [74] by:

- The input distribution $p(X)$, which represents the probability of selecting each element in X, which also can be considered as the importance given to each x.
- The transition probability matrix $p(Y|X)$. Conditional probabilities represent the probability of seeing an element from Y once we have seen a given element from X and fulfill $\sum_{y \in Y} p(y|x) = 1$.
- The output distribution $p(Y)$, given by

$$p(y) = \sum_{x \in X} p(x)p(y|x) \tag{21}$$

  representing the average probability of seeing each element in Y.

The elements defining the information channel can be represented in matrix form as:

$$\begin{array}{ccc} P(X) & & P(Y|X) \\ \begin{bmatrix} p(x_1) \\ p(x_2) \\ \vdots \\ p(x_N) \end{bmatrix} & \rightarrow & \begin{bmatrix} p(y_1|x_1) & p(y_2|x_1) & \cdots & p(y_M|x_1) \\ p(y_1|x_2) & p(y_2|x_2) & \cdots & p(y_M|x_2) \\ \vdots & \vdots & \ddots & \vdots \\ p(y_1|x_N) & p(y_2|x_N) & \cdots & p(y_M|x_N) \end{bmatrix} \\ & & \downarrow \\ & & \begin{bmatrix} p(y_1) & p(y_2) & \dots & p(y_M) \end{bmatrix} \\ & & P(Y) \end{array}$$

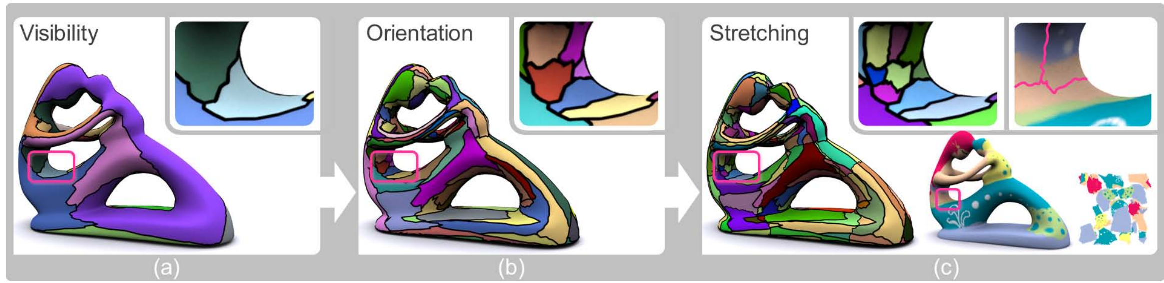


Figure 69: Iterative application of channel clustering: visibility, orientation and texture stretch.

In some cases (e.g., the visibility channel, see below), it could be more practical to compute $Y \to X$ and then invert the channel to get $X \to Y$. This can be easily done with the help of Bayes' theorem:

$$p(x, y) = p(x)p(x|y) = p(y)p(y|x)$$

We classify channels that require an inversion as *indirect* channels, while the ones that do not as *direct* channels.

The mutual information $I(X; Y)$ between two random variables X and Y is defined by $I(X; Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$ and represents the degree of correlation or dependence between X and Y.

Finally, the Jensen-Shannon (JS) divergence is defined as:

$$JS(\pi_1, ..., \pi_n; p_1, ..., p_n) = H\left(\sum_{i=1}^{n} \pi_i p_i\right) - \sum_{i=1}^{n} \pi_i H(p_i)$$

with p_i are the probability distributions defined with normalized weights π_i, $i \in [1, n]$. We use this divergence as a measure of the similarity between the triangles to cluster. For practical reasons, we introduce the following shorthand notation for this last expression for the case of two probability distributions a and b and an output variable Y involved in the channel:

$$JS(Y|a, b) = JS(\pi_1 = p(a), \pi_2 = p(b); p(Y|a), p(Y|b))$$

As we will show, throughout this chapter for each feature to cluster, X is the set of model triangles, except in the visibility channel which is an indirect channel and in this case X will be the set of viewpoints.

5.2.1.1 *Visibility Channel*

Visibility is the first feature that we want to preserve from the scene to be rendered. If two triangles share the same visibility information, that means that these triangles can be considered to be roughly seen with the same occlusions from every possible viewpoint. As we are in the pre-processing stage of our technique, the idea here is to decouple the visibility information from the viewpoint itself by making estimations of how visible triangles are for all viewpoints. Thus, for the visibility feature, we define an information channel between the set of input viewpoints V and the set of output polygons of a mesh T, as $V \to T$. Later, we invert it ($T \to V$) to obtain a measure of how each triangle "sees"

the viewpoints. In the case of the visibility we choose to use an *indirect channel* because it is simpler to compute the visibility of the scene polygons from every viewpoint using rasterization than computing it from the polygon point of view.

The $V \to T$ channel was previously introduced in [23] as the viewpoint channel and conceptually expresses, for any given triangle, how all viewpoints "see" it. There, the viewpoints where given the same importance $p(v_i) = 1/N$, with N the number of viewpoints and the transition probability matrix $p(T|V)$ was defined from the projected areas of polygons $t_j \in T$ at each viewpoint $v_i \in V$. With that definition, the authors took into account not only occlusions but also orientation differences in the same channel. As we will show, we modify this definition by making it only sensible to the occlusions produced in our scene, not considering orientation in this channel. The reason behind that choice is that focusing on only one feature for each individual channel will allow us to have a finer degree of control over the clustering process. Also, note that the traditionally definition of the viewpoint channel is completely resolution dependent, which will not happen in our case, which greatly benefits the clustering process, and as a consequence, the final quality of the rendering.

Thus, let's redefine the $V \to T$ channel to take into account only the visibility criteria: We set the probability for each viewpoint as $p(v_i) = 1/N$, with N the number of viewpoints, all having the same "preference". We define the *occlusion ration* as $OR(t_j, v_i) = a_{v_i}(t_j)/u_{v_i}(t_j)$, where $a_{v_i}(t_j)$ is the projected area (in pixels) of triangle $t_j \in T$ at viewpoint $v_i \in V$, and $u_{v_i}(t_j)$ is the same projected area without taking into account occlusions. Observe that $OR(t_j, v_i) <= 1$, with the equality meaning that t_j is fully visible from all v_i, and 0 that it is fully occluded. We define the normalized transition probability matrix $p(T|V)$ as $p(t_j|v_i) = OR(t_j, v_i)/\sum_t OR(t, v_i)$. The output distribution $p(t_j)$, is computed as the averaged occluded area of t_j from all viewpoints $p(t_j) = \sum_{v_i \in V} p(v_i)p(t_j|v_i)$. To accelerate computations, we have implemented this channel calculations entirely on the GPU, being an order of magnitude faster than previous approaches [23].

5.2.1.2 *Orientation Channel*

This *direct* channel accounts for strong differences in the polygon orientations, even if they have a similar visibility, like sharp edges or strong curvatures. It goes from the 3D triangles $t \in T_{3D}$ to themselves, which means that the channel is $T_{3D} \to T_{3D}$. Here the input probabilities $p(t)$ are set to the constant value $1/M$, with M the total number of triangles to be processed, $p(t_i|t_j) = (1 - n(t_i) \cdot n(t_j))/\sum_t(1 - n(t_i) \cdot n(t))$, with $t_i \in T$, $t_j \in T$, $n(t)$ the normal of triangle t, and we calculate $p(t_j)$ with Eq. 21.

5.2.1.3 *Texture Stretching Channel*

This *direct* channel is designed to account for stretching in the textures, which should be preserved if we are going to do an interpolated up-sampling process of the texture coordinates (see below). Conceptually, given a triangle t_i in 3D space, it explains how is its stretching with respect to all other triangles t_j in T_{3D}. It is a channel $T_{3D} \to T_{3D}$. Again, we set $p(t_j) = 1/M$, $p(t_j|t_i) = (1 - |S(t_i) - S(t_j)|)/\sum_t(1 - |S(t_i) - S(t)|)$ with $t_i \in T$ and $t_j \in T$, and we calculate the $p(t_j)$ for the 3D triangles with Eq. 21. Here, $S(t)$ is the stretching of triangle t computed as described by Degener et al. [22] and normalized to 1.

5.2.1.4 *Other Possible Channels*

Although we have not implemented them, it is easy to think of other clustering criteria. Our previous channels already took into account visibility, orientation and texture stretching effects. Complementary channels could be reflections, refractions and other optical effects. For instance, effects like specular reflections might require a carefully tuned channel. However, we do not expect these channels to introduce large changes with respect to our current implementation. For instance, soft shadows or ambient occlusion are somewhat already included in the visibility channel, as the quality of our results shows. For instance, Figure 82 contains both, relying only on the above defined channels.

5.2.2 *Clustering a Single Channel*

In general, each cluster represents an homogeneous continuous area in the model, and a cluster boundary reflects an abrupt change (e.g., discontinuity) in any relevant parameter (e.g., visibility). For any channel, the clustering process proceeds in four basic steps: First, the seeds are selected. Then, two clustering stages are performed in sequence: first, a parallel clustering grows clusters starting from the seeds, making them as big as possible, and second a sequential stage groups the triangles that might remain from the parallel step. Depending on the model, the thresholds used and the selected seeds, we may have too small clusters from the sequential step, which we try to merge them in a final step.

5.2.2.1 *Clustering Initialization*

The first step of the clustering algorithm is the initialization step. This initialization involves two parts:

- **Information channel computation**. First of all, we need to initialize the data needed for the clustering algorithm to work, that means computing all the elements of the current information channel. This implies computing the probabilities $p(x)$, $p(y)$ and $p(Y|X)$ as described in previous sections.
- **Channel denoise**. The evaluations that feed an information channel can be the result of a possible noisy measure, like the one done for the visibility channel in Section 5.2.1.1. There, measuring means rendering the triangles with OpenGL, which provides a good but not exact estimation. To prevent future problems, we *denoise* it by replacing the value for each probability $p(Y|t_i)$ by the average of the probability over the immediate similar triangle neighbors by:

$$p(Y|t_i) = \frac{\sum_{j \in SN(t_i)} p(Y|t_j)}{||SN(t_i)||}$$

being $SN(t_i)$ those neighbors that are similar enough in the Jensen-Shannon sense $JS(Y|t_i, t_j) \leqslant Th$, $||SN(t_i)||$ the number of neighbors of t_i and Th is a threshold specifically set for this channel. Observe that triangles that are not similar will not be taken into account during the denoise process, avoiding situations in which two neighbor triangles have strong differences in the evaluated feature.

5.2.2.2 *Selecting Seeds*

Once we have the current information channel ready to work with, we begin the clustering process by looking for a set of representative triangles from our scenes that we will call seeds. We start with an empty set of seeds and then progressively fill it until adding a new seed would not produce any improvement on the quality of the representativeness of the set. That is, first, we add the most representative triangle, which is the one that has the minimum $I(x, y)$ value. This means that this triangle is the one that has the most balanced information concerning the feature that the current channel encodes. Then, while we are not set and there are candidate triangles to select, we choose the next triangle M with the maximum difference with all the other seeds by:

Seed candidate selection:

$$M = max_{t \in T} \sum_{s \in seeds} JS(Y|t, s)$$

such that, for every already selected seed s, M verifies that is different enough from s, i.e., $JS(Y|M, s) > Th$. In that case, we add M to the set.

Finally the process stops when there is no such M to be selected. The final set of seeds can be understood as the set of triangles that would represent the scene given the feature of the current channel.

As can be seen, every time a seed needs to be determined, we need to compute a large number of similarities with the Jensen-Shannon inequality. This means that, for each pair of triangles (i.e. the seed and a candidate triangle), we need to perform a loop of as many elements as triangles in the scene (this coincides with the size of the conditional probability distributions $p(Y|X)$) to obtain their similarity value. Actually, for large meshes, this can be a serious bottleneck, so we decided to do a parallel GPU-based implementation that distributes among kernels the calculation of the needed JS divergence of a given seed with respect to the rest of triangles of the scene. We would like to emphasize that this is done lazily and only for the requested seeds.

5.2.2.3 *Clustering strategy*

After the representing triangles of the scene have been selected for a given feature, we perform a clustering approach composed of three steps:

- **Parallel clustering**. First, we grow the clusters from the seeds in parallel. A triangle can be in a given cluster if its Jensen-Shannon divergence with respect to the seed is smaller than the user-provided threshold Th. All the seeds grow in a greedy way, trying to grab one ring of triangles at each iteration.
- **Sequential clustering**. Once the parallel clustering is finished, there might be small groups of triangles that were not added to any cluster, mainly because another cluster "barred" the "right" one to grow in their direction. Then, the sequential clustering selects a new seed, growing the cluster until no more triangles can be added. This process is iterated until no unclustered triangles remain.

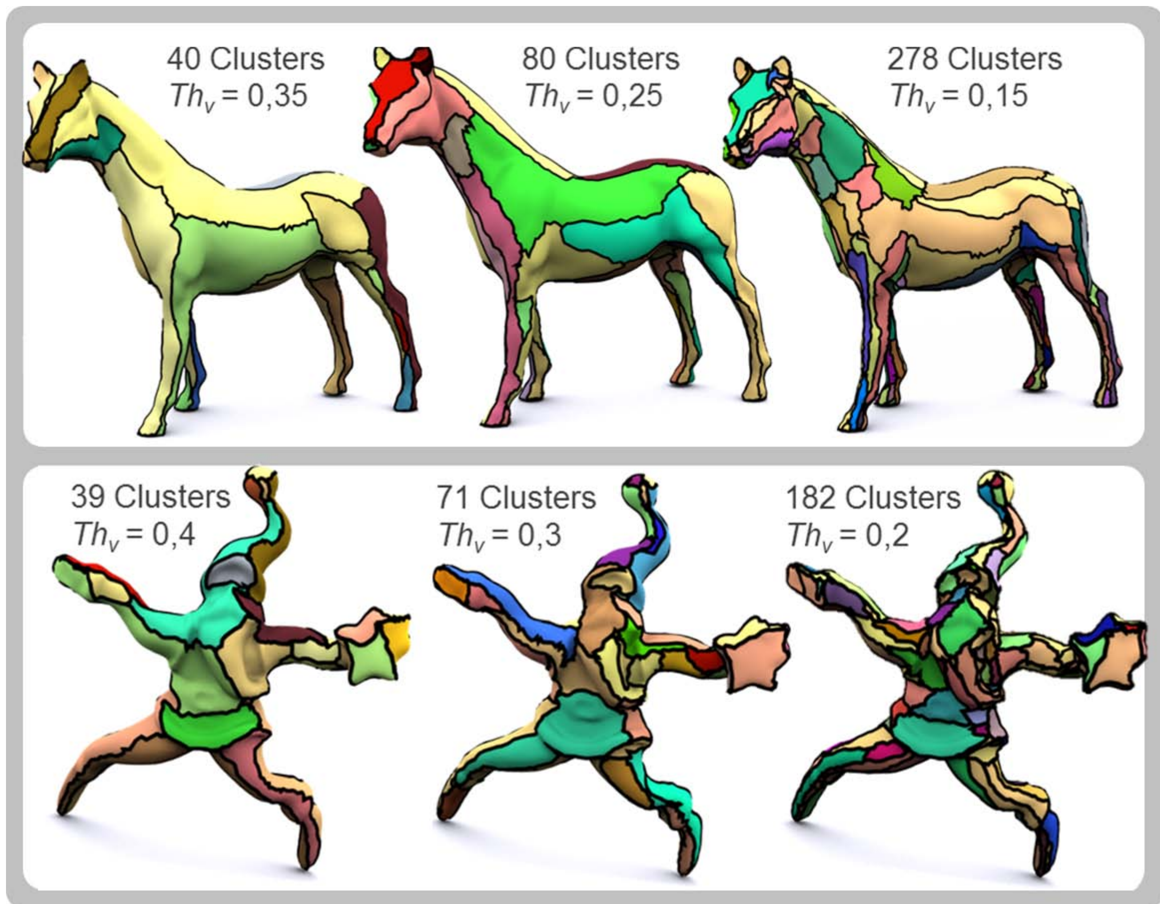


Figure 70: Variation of clustering threshold Th.

- **Merging**. After the clustering process, it can happen that some clusters coming from the sequential step consist in a really small number of triangles. As the number of clusters may slow down the runtime rendering stage, we correct this in this step by merging them with the most similar neighboring cluster. The cluster chosen to merge the small cluster with is the one whose seed has the smallest JS divergence.

In Figure 70 we show the effect of selecting different thresholds, shown on the top of the figure, and applying the described clustering strategy to the horse (top) and santa (bottom) models. In this figure we have taken into account only the visibility channel for illustration purposes. As can be seen, the clustering results in somewhat different clusters being built with different seeds also. This is completely correct: first of all, a tight threshold produces a set of seeds that is a subset of the set produced by a more relaxed one. Second, the clustering process itself will stop growing sooner if the threshold is tighter.

5.2.3 *Smoothing*

The resulting clusters from the previous steps can have rough boundaries, which would result into a poor interpolation at certain views. This is mainly because rough boundaries could be lost in the initial low-res passes of our rendering algorithm, just because they could be consider as sharp features in that resolution (too much detail for such a low resolution). See Figure 71, top row. In the boundary regions there are triangles that could

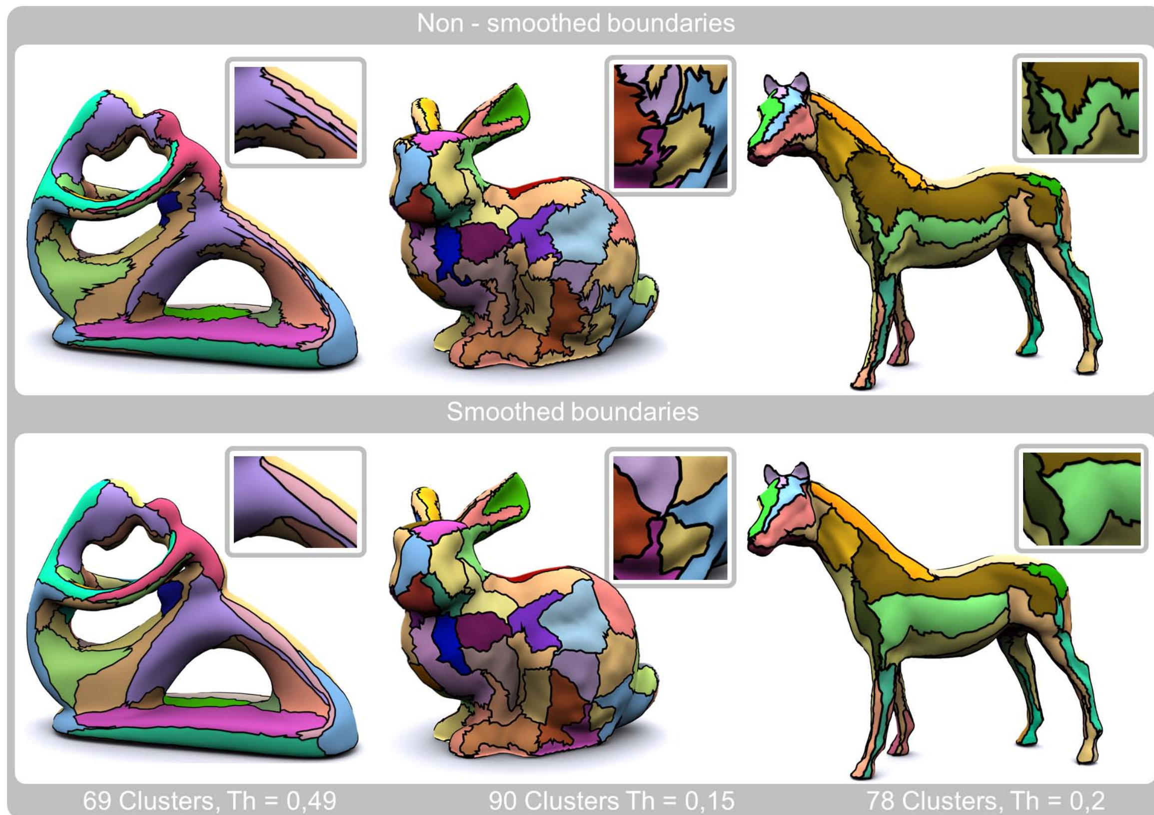


Figure 71: Cluster smoothing.

have been clustered with the seeds of both clusters sharing the boundary, and that were assigned to one given cluster just because the growing process made this cluster to "arrive first". We call the set of all these triangles the *boundary band*. Actually, defining the band this way can produce too broad bands, so we restricted the definition to those triangles that were clustered with a seed but that are more similar (in the Jensen-Shannon sense) to the other seed (See Figure 72). The actual boundary between two given neighboring clusters can be any line inside (and along) the *boundary band*. To compute this smooth new boundary, we follow a procedure inspired in the work by Sander et al. [73].

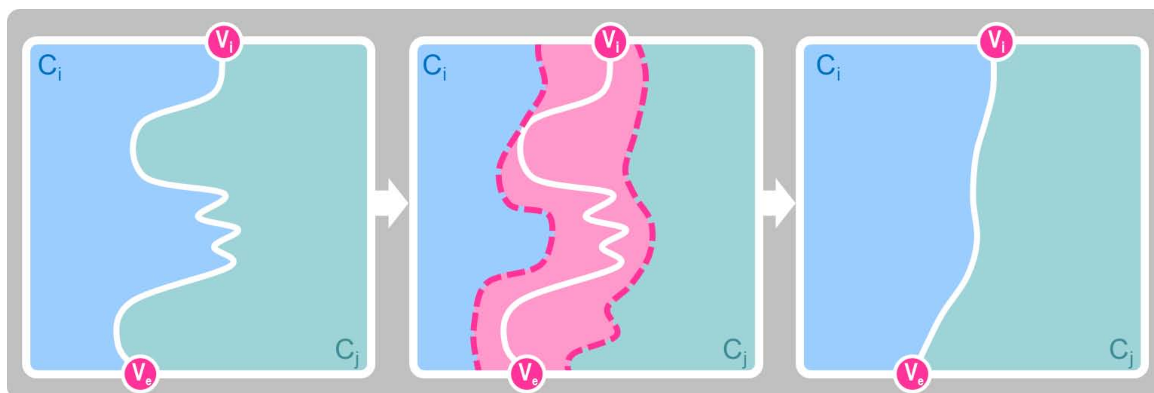


Figure 72: A band between two clusters, C_i and C_j, defined around a boundary from vertex V_i to vertex V_e.

First, for each boundary between clusters we generate the actual *boundary band*. We initialize it with all the edges of the current boundary. Then, for each triangle that could have been more likely clustered with the seed at the other side of the boundary, we add all its edges whose incident triangles are completely inside one of the clusters we are considering. This last condition was introduced to avoid problems by accidentally modifying other clusters.

Once all edges have been selected, we simply compute the shortest path through the band from the starting boundary vertex to the end vertex. In our implementation, we use the well known A^* algorithm. Once the new boundary has been computed, we simply reallocate the triangles to their new clusters. As a positive side-effect of this reordering, the resulting clusters are not only smoother than before, but also more equilibrated in their shapes (See Figure 71, bottom row).

5.2.4 *Animated Scenes*

We want to take into account animated scenes or characters with our technique. For that purpose, we repeat the clustering strategy, previously seen, for each keyframe in the animation of the model. Usually, an animation consists of a series of keyframes which are interpolated to get the final postures of the characters for in between frames. We rely on the same idea and use these keyframes to iteratively perform the clustering for each one. Starting from the initial keyframe, we repeat the clustering pipeline for each successive keyframe, every iteration working after the result of the previous one. This way, we ensure having a consistent model custerization, which can be reused for every frame of the animation. In Figure 73-left, we show an initial keyframe with its corresponding clustering (visibility, orientation and texture stretching). Then the second keyframe (see Figure 73-right) is clusterized by using the previous computed clusters as input meshes for the clustering algorithm.

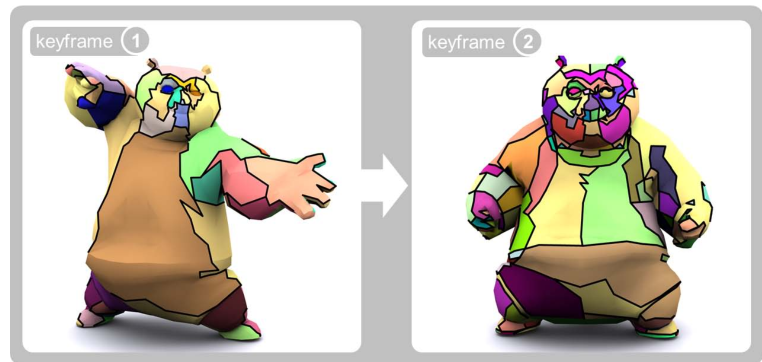


Figure 73: Iterative clustering refining for different model keyframes (left and right).

5.2.5 *Threshold Selection*

Although the only parameters that the user is expected to select are the channel thresholds, selecting a good threshold can be challenging for an inexperienced user. To provide an intuitive interface, we use the upper and a lower bounds to the JS divergence, which can be demonstrated to be $JS \in [0, 1]$ [52]. This way, the user can use a simple slider to control the clustering by setting any value between $Th = 0$, resulting in a cluster for each triangle, and $Th = 1$, resulting in a single cluster for the whole model. This way, threshold selection becomes a simple and easily configurable task.

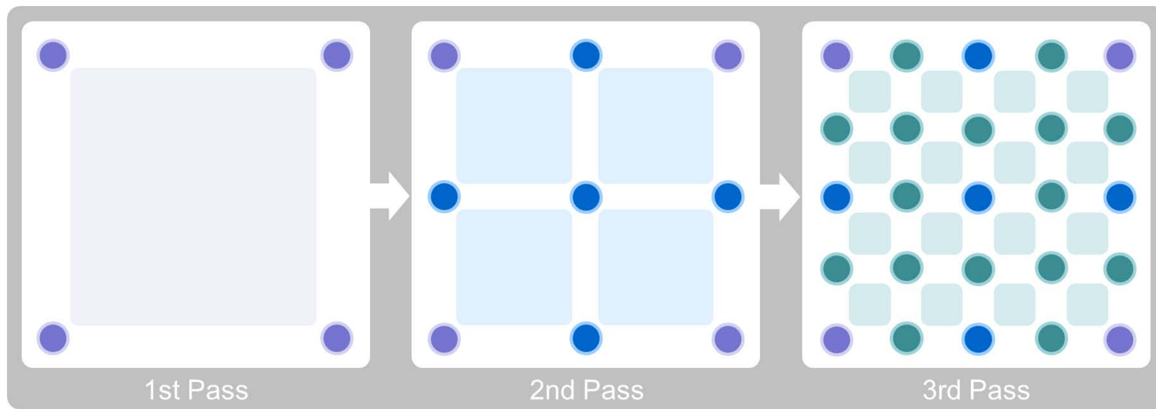


Figure 74: Rendering pattern for three passes starting at a 2×2-pixel image.

5.3 RENDERING BY UPSAMPLING

Here we focus on the rendering stage, which consists of a multi-pass approximate strategy that progressively computes the final image from the information gathered at previous passes. Based on this information, the runtime shaders decide whether to reuse the previously computed values, to interpolate a new value from them, or simply to perform a full computation to guarantee the final image quality. For each pass of the algorithm a different resolution is used, going from the lower one to the hi-res final image and for each pixel there, we store the set of values that we want to evaluate/interpolate:

- **Position**. It is one of the most important values to be interpolated, as it will allow us to obtain the visualization results of the primary rays with our scene, that is, the hit point for every pixel.
- **Texture coordinates**. As we want to apply common texture mapping to our models we will compute/interpolate the texture coordinates for each pass.
- **Normal**. To apply several illumination effects (common shading, ambient occlusion, soft shadows, ...), we store the normal of the position of the surface hit.
- **Ratio of occluded samples**. For each pixel we want to compute/interpolate its occlusion factor.
- **Ratio of shadowed samples**. Also, to render our scene with shadows computations, for each pixel we will evaluate/interpolate the amount of shadow produced there.
- **Cluster ID.** One of the most important values is the cluster ID as it will drive the decision on whether to interpolate or to perform a full evaluation of the pixel.

At each pass of our algorithm, samples are evaluated by examining the samples computed at the previous pass. In Figure 74, left we can see a simple 2×2-pixel image generated in the first pass (lilac dots). Then, at the second pass, new samples are added in between the previous ones (the blue dots between the lilac ones) and so on. The key to our up-sampling algorithm is that, even when different passes generate different resolution images, the samples evaluated at each pass are the same, and thus reused at the next passes. Observe that even this pattern is really well suited for interpolation purposes, it

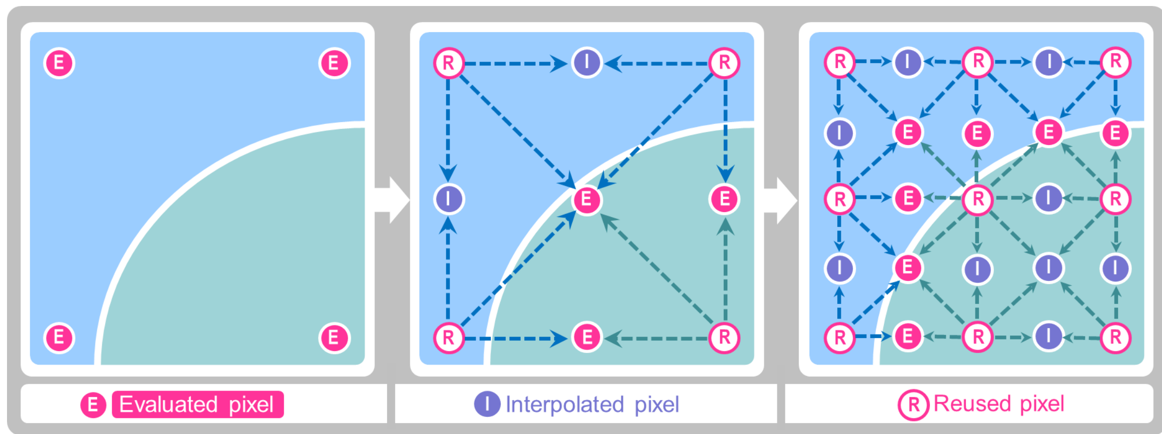


Figure 75: Re-used (R), interpolated (I) and evaluated (E) samples in the successive passes.

implies that each pass should have a size of $(2N-1) \times (2M-1)$ if the previous one was $N \times M$, but we can render any resolution by just starting from correctly chosen N and M and discarding a thin border of unused pixels at the end.

The first option to implement this pattern is to use a single image of the final resolution, but to work only with the samples corresponding to the respective pass, ignoring the others. Although this approach has the advantage of requiring no extra space, if this is going to be implemented in a GPU-based ray tracer, as in our case, it produces highly incoherent memory accesses. As a result, it will strongly reduce the performance and will make useless the improvements achieved by avoiding evaluations. The second option is to make each pass to work on a different target image, effectively resulting in an image pyramid. The memory requirements are the same as for Mip Mapping, about one third of the final image size. Also, considerable speedup is obtained thanks to the increased coherent access patterns to GPU memory. In our case we have used the later option to store all the evaluations/interpolations at each pass. We have called it the *I-Buffer* (from interpolated information).

As mentioned, each pass computes the new samples by reusing the results of previous passes. As one can realize, the first pass has no previous reference image, so all samples must be fully evaluated, as shown in Figure 75-left. Then, from our sampling pattern, the coincident samples at the new resolution can be directly copied, as shwon in Figure 75-middle with the samples marked as "R" (Re-used). On the other hand, if the cluster IDs of the previous samples are equal, new samples in-between can be interpolated (marked with "I" in the figure). If they are different, they need a full evaluation (marked with "E"). The arrows in the image show which previous samples are used for each new sample. Figure 75-right shows the third pass in the process.

5.3.1 *Hard Shadows and Higher-Frequency Signals*

The described interpolation scheme provides good results for low-frequency signals in a natural way. However, higher-frequency signals, like hard shadows, require some extra work. To avoid missing details, we add an extra shadow check at the pixel program together with the cluster ID verification. For a single light, a simple boolean suffices to know

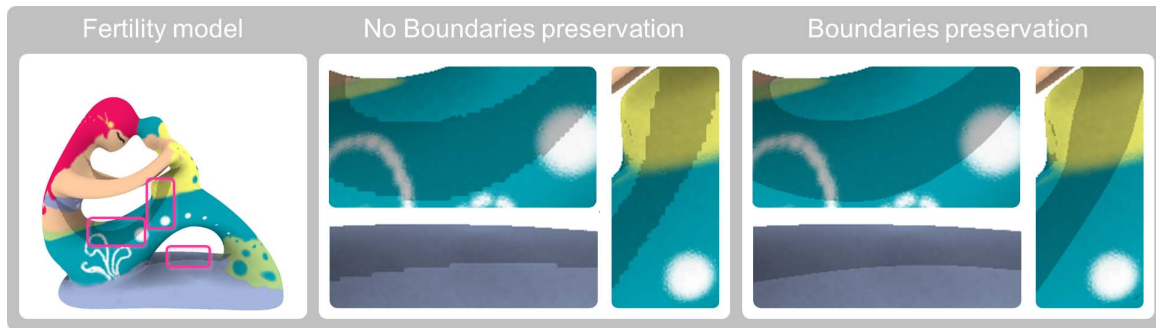


Figure 76: Hard shadow boundary preservation. Center: I-Render with no boundary preservation. Right: I-Render with boundary preservation.

whether a sample is illuminated or not, and from there the check for the shadow boundary is immediate. More lights would require multiple booleans, but they can be "compacted" in a single integer value, thus requiring a single equality verification in the shader. The results can be seen in Figure 76. Other high frequency signals can be guaranteed in a similar way, as long as their frequency is not as high as to require a per-pixel evaluation, which would render our technique useless (e.g., hair, grass, fibers).

5.3.2 *Automatic Pass-Controlling Mechanism*

One of the main factors that affect the performance of our technique is the number of passes used (See Section 5.4). If we visualize a scene from a viewpoint that results in a small ratio of interpolated pixels, the extra cost of our multi-pass algorithm will overshadow the gain produced by avoiding computations and hence it will produce worst performance than common Ray Tracing. Optimizing it is a complex task, as it strongly depends on the scene itself, the distance from the observer, and the complexity of the shading, among other factors. To avoid cumbersome manual trial and error tests, we introduce an easy yet powerful automatic controlling mechanism that selects the number of passes trying to guarantee a user-defined upper and lower framerate bounds. We perform as follows: If the current average framerate is smaller than the lower bound, the number of passes is increased. If it is above the upper bound, the number of passes is reduced. In practice, we have observed that, for medium and far distances, more than 5 passes provide too blurry results and some perceivable "swimmering" for moving objects. This is because the initial resolution is too low compared to the hi-res one, and this produces that the content of the pixels from one frame to the other differs greatly (some important features may be missing because of the low-initial resolution). Observe that this approach does not strictly guarantee the framerate: if the user selects an extremely complex shading plus real-time bounds, the system will increase the number of passes up to the maximum allowed, never achieving the required bound.

5.4 Results and Discussion

We integrated Aila and Laine's [2] GPU ray tracer into our application, which can currently be considered as the state of the art in Ray Tracing algorithms (see Appendix B). All renderings are generated using a viewport of 1025×1025 pixels, and we have used 150

Figure 77: Quality depending on the number of clusters. Left: the clusters, Right: renderings showing details and error values in false color (inset).

samples for both, ambient occlusion and soft shadow computations. The times to generate the clusterization of the models used in the chapter range from several seconds up to 5 minutes. It depends on the number of polygons of the mesh, the channels involved, and their associated thresholds. Once the clusters have been computed, we store the cluster ID of each triangle as a third texture coordinate for each vertex of the scene. That way, we merge the texturing parameterization of the mesh with the information-theoretic clusters, resulting in a new set of charts. Thus, we easily integrate this information to be used in runtime in any rendering pipeline.

As can be seen in Figures 77, 78 and 79 the two main factors that affect quality and performance of our approach are the number of passes and, less significantly, the number of clusters. In Figure 77 we can see that the more clusters, the better the quality of the final image. Observe how textures are well preserved (insets), but a correct shape interpolation of the fertility model requires more clusters, as can be seen in the error values (bottom). In this case, this is because the visibility threshold has been set too low for the fertility model and hence, from some viewpoints we merge parts with different visibility. Our experience shows that the more channels are added to the preprocessing, the more relaxed the thresholds can be, still obtaining good quality results. In both cases framerates are similar: the model with 110 clusters requires 427.4 msec (thresholds: Visibility=0.54, Orientation=0.3, Stretching=0.1), while the one with 181 clusters requires 500.81 msec (Visibility=0.48, Orientation=0.02, Stretching=0.1).

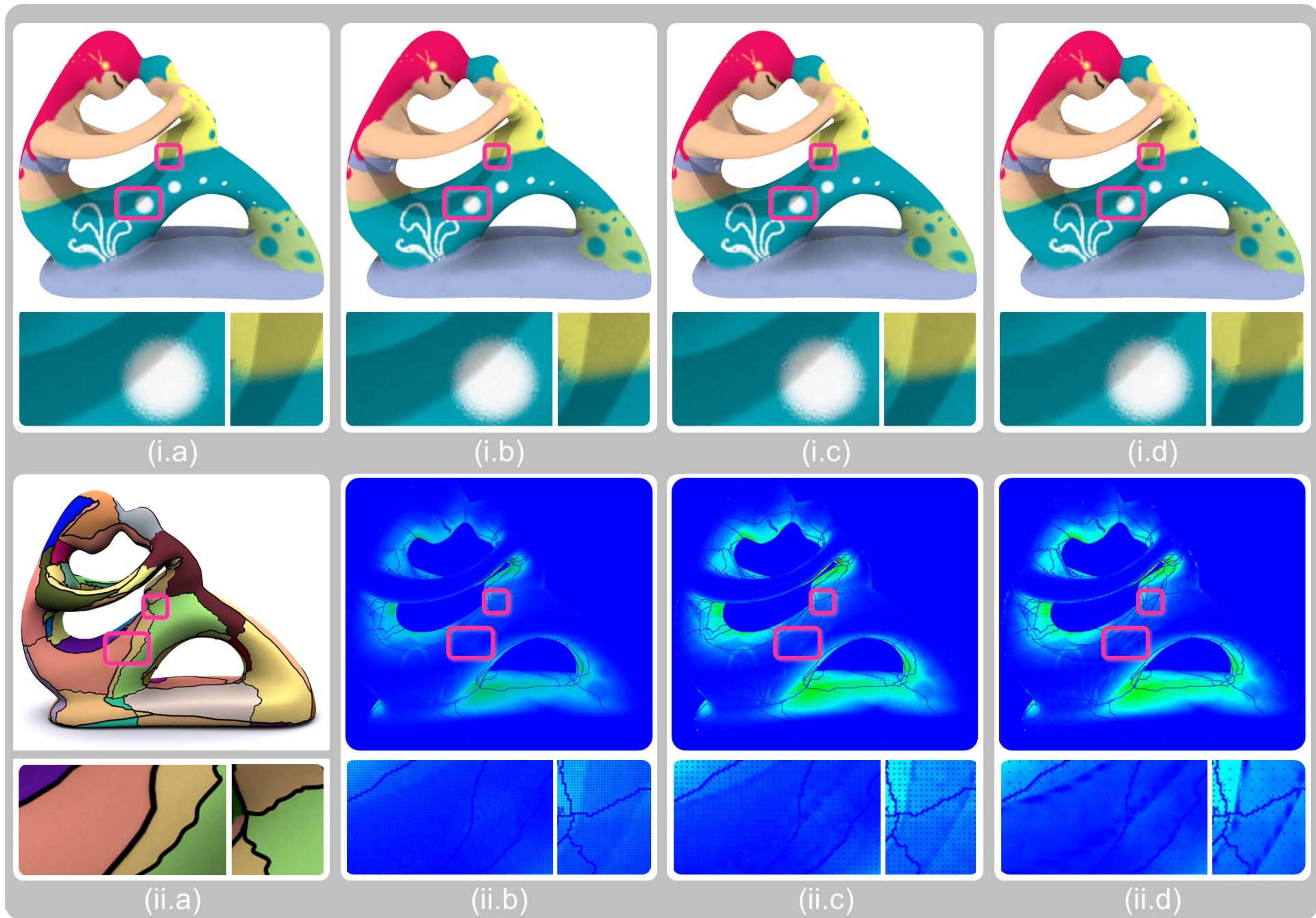


Figure 78: Quality depending on the number of passes. Top row: Ray Tracing reference and 2, 3 and 4 passes respectively. Bottom row: the clusters and the respective error images. The insets show details of the texturing and soft shadows.

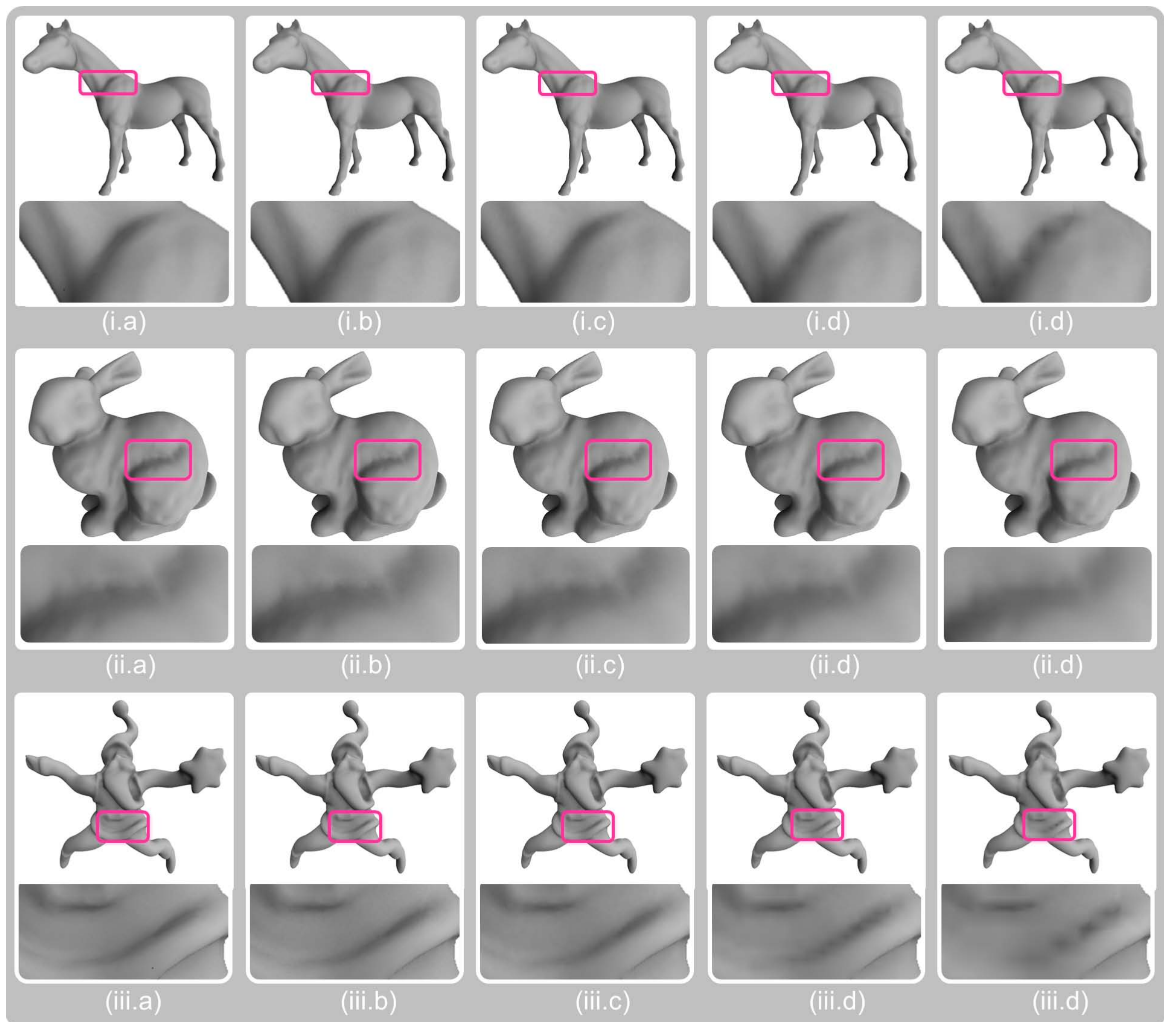


Figure 79: Quality depending on the number of passes. Top row: Ray Tracing reference and 2, 3 and 4 passes respectively. Bottom row: the clusters and the respective error images. The insets show details of the texturing and soft shadows.

In Figures 78 and 79 we can see the dependence of the rendering quality on the number of passes. In the first figure the fertility model is used. The first row shows the rendering quality using one pass (first column), two passes (second column), three passes (third column) and four passes (fourth column). At the bottom an inset showing two different details of the model. The second row shows the clustering used as well as the error maps respect to common Ray Tracing (top row, first column). As expected, the more passes, the more approximated the rendering is. This is more noticeable in the zoomed views (insets), which show a good quality for the textures but some artifacts in the soft shadow details, which is reflected in a low error (bottom). This happens when shadow boundaries are interpolated as they fall in the same cluster. Please, note in the error maps how cluster boundaries produces low-error values as they are fully evaluated. Also, see how the maximum error is located in regions where the ambient occlusion signal is. This is because ambient occlusion can be considered a smooth signal even if it is computed with a discreet number of samples (200 for each pixel) which introduces a small degree of noise. Observe how some texture details cause high error values as well, as our interpolation does not map them as in the original ray-traced rendering. In Figure Figure 79, several models have

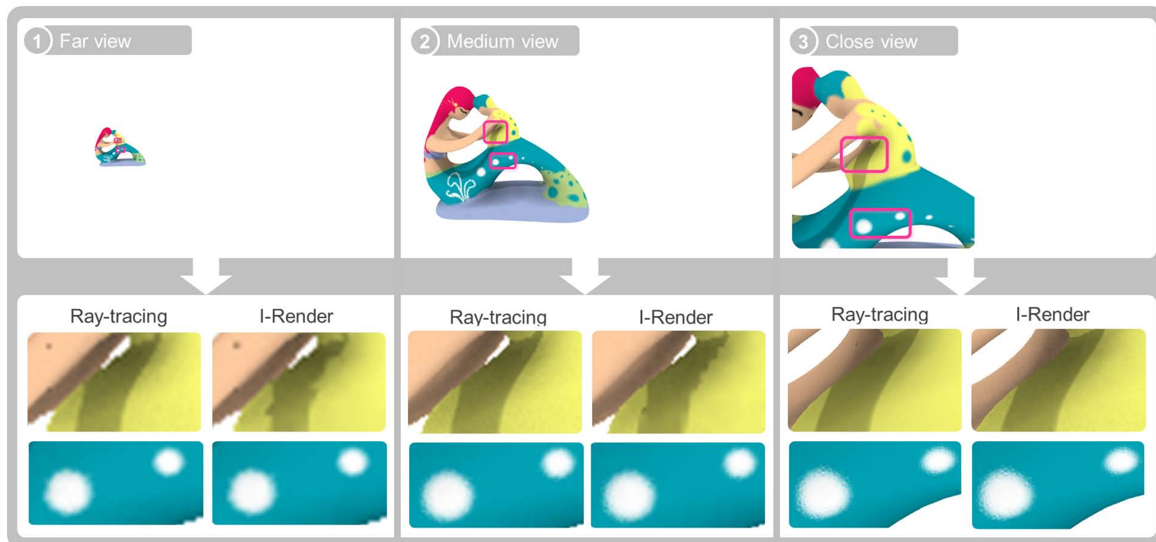


Figure 80: Quality depending on the observer distance. Timings in Figure 84.

been used: the horse (first row), the bunny (second row) and the santa (third row). Each column shows the results generated by I-Render from one pass until five passes. Each image has its corresponding inset showing a certain part of the mesh to better appreciate the image quality produced by our technique. See again the high level of fidelity obtained with respect to the original image (first column).

In Figure 80, given a fixed number of passes (3), we can observe the quality of the render depending on the distance to the observer: At short distances more samples are evaluated over the model projection in screen space, resulting in a better quality image affecting mainly the high-frequency shadows. However, as the model size on the screen gets smaller, the overall perceived quality is not reduced, but sharp features see their quality decreased. Also, it is important to notice that the texture signal is correctly preserved at any distance, as we interpolate texture coordinates instead of directly the texture color.

In Figure 81 we can appreciate that *I-Render* is fully compatible with the use of textures, as we use it to interpolate *texture coordinates* and not the textures themselves. As texture coordinates (lower insets) have a lower frequency than the textures themselves, their interpolation results in high quality interpolated renderings (upper insets), even with high frequency details. All the models used in the paper has been textured using multi-chart parameterizations. When texture coordinates are interpolated with our technique, it may happen that, for some convex charts, we could obtain interpolated points that fall outside. To avoid such situations we have used padding, but in extreme cases the technique described by González and Patow [32] could be used to sample the correct texture coordinates avoiding possible artifacts, i.e., color from outside the charts or from other charts leaking into the atlas.

In Table 3 we can see a comparison of the times needed to evaluate and to interpolate a single pixel (in ms), averaged over a number of viewing angles and distance ranges. As we can see, the evaluation cost is several orders of magnitude slower than interpolation.

Figure 81: Correct texture rendering. Left: clusters, Middle: Ray Tracing, Right: I-render. The lower insets show the smooth interpolation of the texture coordinates.

All the models used in the paper has been textured using multi-chart parameterizations. When texture coordinates are interpolated with our technique, it may happen that, for some convex charts, we could obtain interpolated points that fall outside. To avoid such situations we have used *Traveler's Map* (see Chapter 3) to sample the correct texture coordinates avoiding possible texturing artifacts-color from outside the charts or from other charts leaking into the atlas).

Figure 82 presents another example with 387 clusters (left) and a quality comparison of Ray Tracing (middle) with our technique (right). Again observe the great similarity between both even in the details from the insets. *I-Render* can visualize the scene up to 8 times faster and, thanks to the pass-controlling mechanism we never fall under the Ray Tracing performance, even when a low interpolation ratio is produced.

Models	Method	Evaluation	Interpolation
Dancing Kids	Ray Tracing	21.7969	0
	I-Render	22.3699	0.0142
Fertility	Ray Tracing	20.85121	0
	I-Render	24.6957	0.0140
Chinese Vase	Ray Tracing	6.3608	0
	I-Render	6.4115	0.0184

Table 3: Table showing the differences between the cost of evaluated and interpolated pixels for Ray-Tracing and I-Render for several models shown in the paper.

Figure 82: Our system first performs a feature-based clustering of the object (left) and then reconstructs the final image by an approximate interpolated approach (right). Quality is comparable with Ray Tracing (middle), observe the texture details, shadow boundaries and correct visibility. Our technique can render the scene up to 12 times faster than Ray Tracing.

Figure 83: Animation sequence of the panda model using our technique.

In Figure 83 we show the results of applying *I-render* to an animation produced over the panda model (3191 triangles). From left to right we present four different frames between two animation keyframes (first and fourth columns). The final refined cluster used for the whole set of poses can be seen in Figure 71-right. Note the good quality of the images obtained, even for the second and third poses, which are not directly taken into account during the clusterization step.

In Figure 84 we can se a graph showing the relation between the rendering time and the distance to the observer for the Fertility model (25K triangles). Observe how *I-Render* outperforms Ray Tracing in close views as the number of interpolated pixels increase, and how, at large distances, our pass-controlling mechanism guarantees, at least, the performance of Ray Tracing.

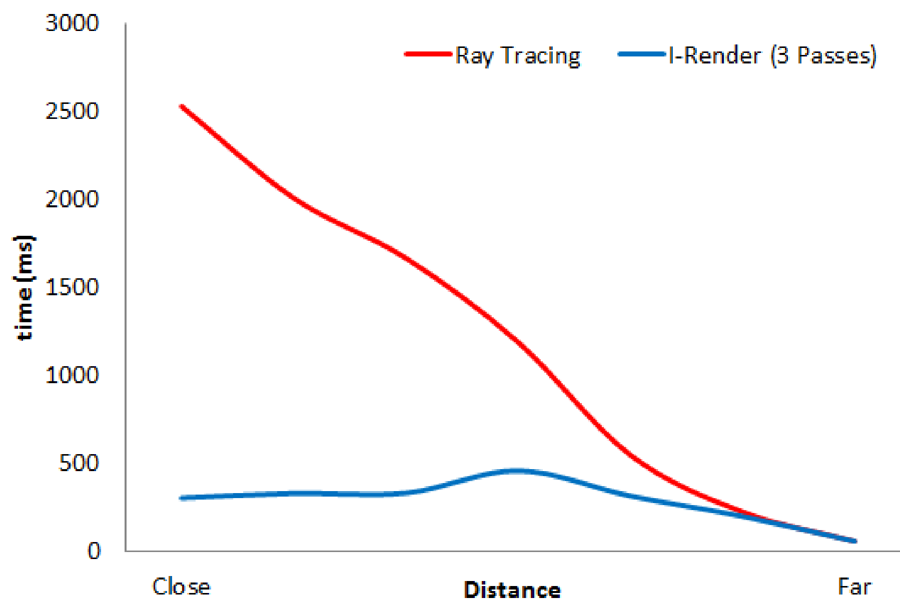


Figure 84: Graph showing the relation between rendering time (in ms) and distance to the observer (in arbitrary units), for the model in Figure 80.

Rendering with Ray Tracing an image is, in general, $O = n^2E$, with n the side of an $n \times n$ image and E the cost of evaluating an average sample. Our technique changes this to be $O = nE + n^2I$, where we assume that the cost of reused samples (R) is roughly equal to the cost of the interpolated ones (I). The reason of this behavior is that chart boundaries represent a 1D space embedded in the screen 2D space (actually, they are embedded in the 2D projection of the 3D object space), so they grow linearly while screen resolution grows quadratically. This can be appreciated in Figure 85. Let us remark that this trend is exactly the same that the one followed by chart boundaries in texture space in *Continuity Mapping* (see Chapter 3). Of course, as every pixel must have a value in the end, the interpolated (or reused) samples also need to be considered, resulting in an asymptotic quadratic behavior. However, the multiplicative constant of interpolated samples can be orders of magnitude smaller than the one for evaluated samples. As a result, the more complex the computation to perform, the better for our algorithm. Of course, this difference is completely dependent on the specific elements involved in computing a particular sample. If the exact evaluation of a sample is very cheap, then probably plain Ray Tracing is a better option. This is the reason of our pass-controlling mechanism, explained before. On the other hand, if too many passes are selected, the constant C can grow large enough to erase any advantage obtained with our reusing mechanism. All this behavior can be observed in Figure 85 where we show the dependence between the rendering time (in ms) and the resolution of the final image (target resolution). Observe that the higher the resolution is, the the lower the time needed. Also note that more passes in our technique results in less rendering time.

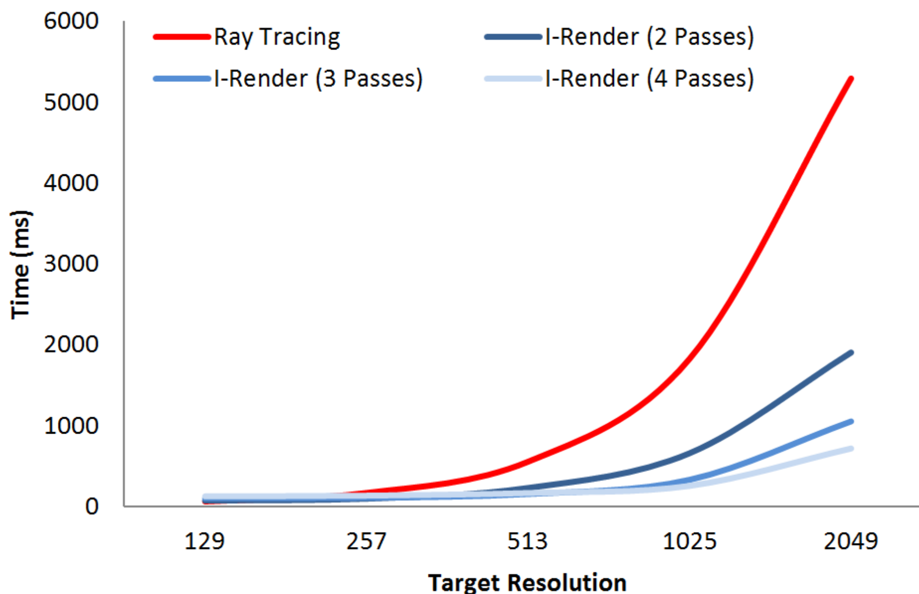


Figure 85: Graph showing the relation between rendering time (in ms) and the number of passes for different resolutions, see Figure 78.

However, in the case of a GPU-based implementation, as in our case, there are additional overheads introduced by the device. On one side, there is a constant overhead produced by the multiple kernel calls, although we found this to be negligible for our scenes. On the other, as threads in a GPU are executed in scheduling units called *warps* that execute

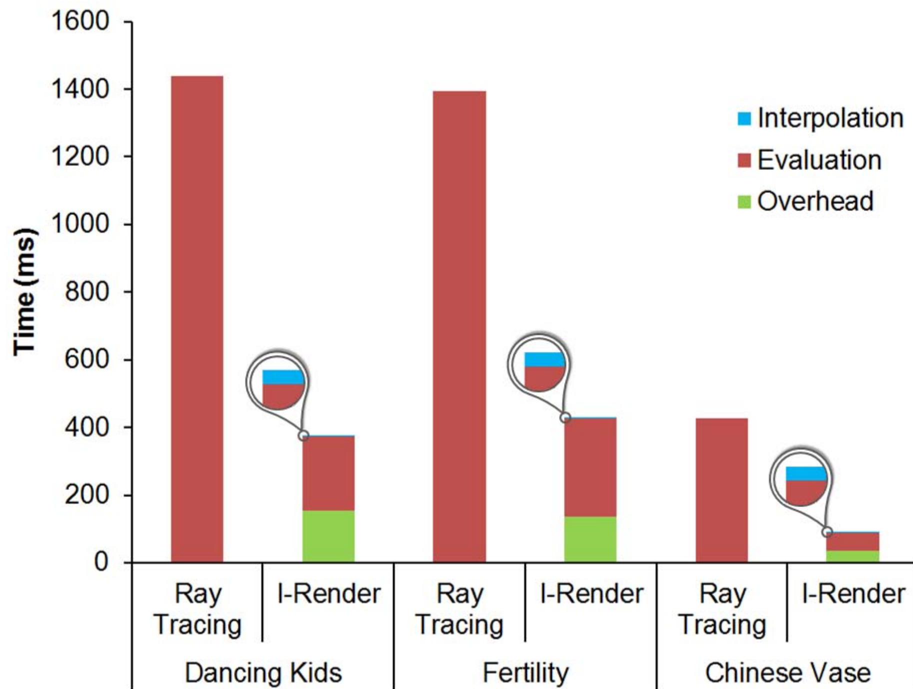


Figure 86: Graph comparing the mean rendering time (in ms) between Ray Tracing and I-Render for several views and models.

in SIMD, there is an overhead that we can easily identify with the control divergence between evaluated and interpolated paths. As we have seen, evaluation is the dominant part of the computations, and the control divergence overhead can be associated to the samples that need to be evaluated (even if one sample in a warp needs evaluation, this cost will drive the warp's total cost), thus resulting in an effective behavior of $O = nE' + n^2 I$, with $E' = E + W$ and W the overhead introduced by the divergence between the threads. Figure 86 illustrates the costs of each term for a number of scenes and averaged over a number of different viewpoints, ranging from close to far distances. To make comparisons fairer, our figures do not include background computations. It is important to remark that this extra W cost does not change the overall behavior of the system, as Figure 85 shows.

In all of the examples shown in the paper, the visibility channel has been involved in the clustering process. As we have explained, we have followed the approach described in [23] to define the set of viewpoints. It is most suitable for only one model or a small set of elements. So, if a large scene is used, even with closed regions (a room for instance), another approach may be needed. For instance, we could define the set of viewpoints as a grid over the scene. Then, we could compute the corresponding conditional probabilities of the channel by taking into account the projected area of the triangles from the set of rays centered at the viewpoint itself and pointing outwards a surrounding sphere.

Finally, we think that the proposed model can be used for general rendering, so it could be applied to rasterization-based rendering, too. However, this would imply rasterizing the whole geometry each time to find the intersection points, which could reduce its benefit. Further research is needed to assert the effectiveness of *I-Render* with rasterized solutions.

5.5 CONCLUSIONS

In this fifth chapter, we have presented *I-Render*, a multi-pass acceleration technique that approximates rendering samples at one pass by reusing values obtained at previous passes. Instead of a full evaluation for every pixel of the final image, *I-Render* takes benefit by the regions of the scene with low variability, that is, they share the same features, and interpolates as much as it can. We found that feature coherence of the low-variability regions in the scene, when projected in screen space, can be exploited to drastically accelerate ray-tracing-based rendering. Our findings show a dramatic acceleration in rendering speed of up to an order of magnitude, and with a quality similar to that of ray tracing. We identify regions that share a common set of user-defined features (defined in a broad sense), and that allows us to compute in a more efficient way (i.e., linear cost) than traditional approaches that compute each pixel from scratch (i.e, quadratic cost). Our multi-pass acceleration technique approximates rendering samples at one pass by reusing values obtained at previous passes. One important feature of this technique is that it is completely independent, and can be used in combination with any existing acceleration technique. Also, I-Render is pixel-bound, which makes it independent of the tessellation of a given model. A conceptually similar screen-space strategy can be found in other works [3, 4, 1]. However, here we explore a new path by letting the user to define clustering features with a completely general framework, incorporating even complex properties like visibility, which has never been unified with other features before, as we do here. A drawback of this technique is that clustering has to be done in a pre-processing stage, which requires knowing in advance if not all, most of the properties of our scene. Future studies should explore whether this stage can be further parallelized in order to be able to perform this in real-time. As our implementation uses a reduced-cost, lazy-evaluation approach, we foresee this as a feasible objective.

Here we have presented a clustering approach based on information theoretical tools, using visibility, texturing and illumination as case-studies. We believe that these tools can be successfully employed for all frequency features, and we hope that there must exist a general and hopefully elegant structure that encompasses all kind of frequencies in a single framework. We also think that the proposed technique can be used for general rendering, so it could be applied to rasterization-based rendering, too. However, this would imply rasterizing the whole geometry each time to find the intersection points, which can reduce its benefit. Further research is needed to assert the effectiveness of I-Render with rasterized solutions.

Feature-based coherence exploitation is a powerful and promising avenue for rendering acceleration, and information theoretic tools appear as an excellent path to unleash this power. It is clear that techniques like the one presented in this chapter actually *benefit* from evaluation complexity: the more complex the computations performed for each pixel, the larger the acceleration ratio. We think that these techniques will increase in relevance as shaders continue to grow in complexity and size.

6 CONCLUSIONS AND FUTURE WORK

Paso a paso te acercas a tu felicidad.
Abre los ojos y los brazos y tu eligirás.
Nuevas oportunidades se abren a ti.
Todo cuanto te propongas vas a logarlo.

In this final chapter we are going to review all the work presented, as well as detail the general conclusions that we have obtained during the thesis. Finally, we will provide hints of future research that could be done after the experience obtained in this thesis.

6.1 Conclusions

In this thesis we have presented techniques that embrace many important areas in Computer Graphics: mesh texturing, mesh deformation and rendering. In each of them we have presented its continuity problems, that imposed limitations to the end of creating computer-generated synthetic images. First, we have presented a robust and practical solution, called *Continuity Mapping*, to avoid discontinuities that appear when multi-chart parameterizations are used. Modelers do not need to take care of how they parameterize the models, i.e., they do not need to create a parameterization that hides explicitly the seams. In contrast, they can concentrate all their efforts in the texturing process itself, as we provide the continuous mapping between the parameterization in texture space and the 3D mesh in object space. *Continuity Mapping* is efficient, as it has a small memory footprint, fast, and compatible with mesh deformation techniques, due to the fact that it works completely in texture space. Second, we have proposed a new multi-cage and multi-level deformation technique that allows to perform more localized deformations with a level of detail that can be determined by the user. This new method is much faster, consume less memory and is more flexible than current cage-based approaches. To provide all these new features we have had to solve continuity problems in object space that current coordinates present at cage boundaries. As a result, we obtained a smooth and continuous deformed mesh. Finally, we have demonstrated how to detect high frequency regions, that is, regions where an abrupt change is produced, on meshes that can be characterized by a given set of user-predefined features. Then we have shown how to use that information to improve the rendering performance. For that purpose, we have first introduced a new clustering algorithm that joins triangles by its level of similarity using information-theoretic tools. Then, we have proposed a new multi-pass technique that generates the final high-resolution image by a set of intermediate passes, evaluating the final pixel values from images of lower size. This technique uses the set of previously created clusters to decide where to perform a full evaluation (a region with discontinuities in a certain feature) and where to perform cheaper interpolation procedures (similar regions). All this process is carried out completely in screen space and, as a consequence, our approach can be used in addition to common acceleration data structures.

As a final conclusion, we can say that in this thesis we have shown how to solve discontinuities in some contexts like texturing or mesh deformation, in a robust and practical way, while in others like rendering, we have taken advantage from the ability to detect them and improve the visualization performance through and intens use of cheaper interpolation. Thus, in the end and with the results provided in this thesis, we have contributed with several techniques that allow to produce better content in a more robust, fast and flexible way than current state of the art. Moreover, all the work done can be directly applied to important and growing industries like videogames, cinema and virtual reality.

6.2 Future Work

We expect that, in a near future, mesh parameterization, mesh deformation and Ray Tracing techniques are going to continue to evolve, giving as a result new and exiting approaches. Some of the opportunities could lay on some of the following ideas:

- **Information-theoretic mesh parameterization**. In the field of mesh parameterization we believe that we could merge our knowledge in parameterizations and information-theoretic tools to improve the construction of new parameterizations. These tools would allow to produce better mappings of the meshes in texture space than current techniques. Then, once the mesh is parameterized, we need to place or pack each individual chart in a common parameter domain (i.e., texture space) to have and efficient storage. We think we could also investigate how to apply all those information-theoretic tools to this packing problem.

- **Automatic cage creation**. This is still an open area for research. Some work has been done in automatic generation of polyhedron (cages) that are similar to its mesh counterparts. But creating a cage for later cage-based deformation it's a much more complex task: When modelers want to deform a mesh using cage-based techniques, they need to know in advance exactly how the final deformation should look like, as it will affect the complexity and modeling of the cage. The final deformation limits the cage construction and hence later modifications with the same cage can be impossible to reach. Also, no universal cage can be used for all the deformation needs on a given model. We can imagine that an adaptive approach may begin with an initial low-resolution cage computed automatically, to later refine cage regions where the modeler needs more accuracy. To make this approach efficient and usable in deformation pipelines, research must be done on how to efficiently recompute weights while adding detail on a region of the cage, and thus, maintain smoothness over the resulting deformation. Probably, the GPU capabilities to generate geometry could be useful for this purpose.

- **Compression of large models**. Given a cage and a set of values associated to its vertices, we can use them to fill with interpolated values the whole volume within the cage. We think that we could use this property as a way to encode information about the mesh. This information would allow us to reconstruct it later on without having the mesh itself and only with the usage of the related cage and the stored information. All this information (cage+data) could be encoded with much less memory consumption than the original mesh and could be used, for instance, for being sent by internet-driven applications in a fast and light way.

- **Automatic semantic crowd generation**. We would like to apply cage-based approaches as a way to generate a similar but different family of models from a given mesh. We think that we could introduce semantic parameters as well as restrictions on cages that would describe the main characteristics of a certain enclosed mesh. As cages are much simpler than the model within, it would allow us to propose some easy-to-use tools to the final user. Then we would automatically generate many new instances of that mesh which would be similar but, different, to the initial mesh, by modifying the semantic parameters of the cage. Later, given two similar exemplar models with similar cages but different semantic attributes and restrictions, we could even think in how to transfer those attributes from one cage to the other to be able to cross characteristics from several different models.

- **I-Render on general scenes**. As has been told in previous chapters, *I-Render* can be used for single isolated models as well as for open and closed scenes. One of the features that we have used to clusterize is visibility, and the way we have faced its

computation may be much more well suited for single objects or open scenes with a low number of elements. So, a solution for a more general environment should be presented (i.e., closed scenes or open scenes with many elements). A way, that we hinted in previous chapters, is to define a grid of viewpoints over the scene, instead of a sphere surrounding the model, and then compute visibility in a similar way as we proposed in the thesis. Many challenges appear and still need to be addressed, for instance, we have to ensure that all the polygons that are visible from any possible viewpoint are taken into account and not are skipped by the incorrect or insufficient placement of the viewpoints.

- **Real-time mesh clustering**. Our rendering acceleration technique is based on a set of pre-computed clusters. Even this technique is also applicable to animated characters by blending the clusterization at each key frame of the animation, fully dynamic scenes with a high degree of interactivity may not work with the proposed solution. Complex dynamic environments may require the clustering recomputation. We think that a GPU oriented implementation would benefit its use in such a scenario, but we also believe that a GPU-driven implementation would not be enough to meet the demands of interactivity. A possible solution would be to compute an initial clustering of the scene in a pre-processing stage. Then, at run-time we could refine clustering on the models only in places where it is needed. However, the decisions that would guide the refinement have to be fast and simple to be able to meet the time demands. Also, we could study ways to make the clustering simpler for that purpose, while still obtaining good results.

Part I

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Anders Adamson, Marc Alexa, and Andrew Nealen. Adaptive sampling of intersectable models exploiting image and object-space coherence. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 171–178, New York, NY, USA, 2005. ACM. ISBN 1-59593-013-2.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8.

[3] Takaaki Akimoto, Kenji Mase, and Yaushito Suenaga. Pixel-selected ray tracing. *IEEE Computer Graphics and Applications*, 11:14–22, 1991. ISSN 0272-1716. doi: http://doi.ieeecomputersociety.org/10.1109/38.126876.

[4] Kavita Bala, Julie Dorsey, and Seth Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. Graph.*, 18(3):213–256, July 1999. ISSN 0730-0301. doi: 10.1145/336414.336417. URL http://doi.acm.org/10.1145/336414.336417.

[5] Mirela Ben-Chen, Ofir Weber, and Craig Gotsman. Variational harmonic maps for space deformation. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-726-4. doi: http://doi.acm.org/10.1145/1576246.1531340.

[6] David Benson and Joel Davis. Octree textures. *ACM Trans. Graph.*, 21(3):785–790, 2002.

[7] Ioana M. Boier-Martin. Domain decomposition for multiresolution analysis. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 31–40, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-687-0. URL http://dl.acm.org/citation.cfm?id=882370.882374.

[8] Peter Borosan, Reid Howard, Shaoting Zhang, and Andrew Nealen. Hybrid mesh editing. In *Proc. of Eurographics 2010 (short papers)*, 2010.

[9] Mario Botsch and Leif Kobbelt. Real-time shape editing using radial basis functions. In *Computer Graphics Forum*, pages 611–621, 2005.

[10] Jacob Burbea and C. Radhakrishna Rao. On the convexity of some divergence measures based on entropy functions. *IEEE Transactions on Information Theory*, 28(3): 489–495, May 1982.

[11] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.*, 21(2):106–131, 2002.

[12] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Rectangular multi-chart geometry images. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 181–190, 2006. ISBN 30905673-36-3.

[13] Ignacio Castano. Next-generation rendering of subdivision surfaces. ACM SIGGRAPH 2008 presentations, 2008.

[14] P. Castelló, M. Sbert, M. Chover, and M. Feixas. Viewpoint-based simplification using f-divergences. *Inf. Sci.*, 178(11):2375–2388, June 2008. ISSN 0020-0255.

[15] Ying-Chieh Chen and Chun-Fa Chang. A prism-free method for silhouette rendering in inverse displacement mapping. *Comput. Graph. Forum*, 27(7):1929–1936, 2008.

[16] Daniel Cohen-Or. Space deformations, surface deformations and the opportunities in-between. *J. Comput. Sci. Technol.*, 24:2–5, January 2009. ISSN 1000-9000. doi: 10.1007/s11390-009-9200-0. URL http://portal.acm.org/citation.cfm?id=1599254.1599256.

[17] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *ACM Trans. Graph.*, 23(3):905–914, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015817. URL http://doi.acm.org/10.1145/1015706.1015817.

[18] Sabine Coquillart. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. *SIGGRAPH Comput. Graph.*, 24:187–196, September 1990. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/97880.97900. URL http://doi.acm.org/10.1145/97880.97900.

[19] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications, 1991.

[20] Rodrigo de Toledo, Bin Wang, and Bruno Levy. Geometry textures and applications. *Comput. Graph. Forum*, 27(8):2053–2065, 2008. ISSN 0167-7055. doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01185.x. URL http://dx.doi.org/10.1111/j.1467-8659.2008.01185.x.

[21] Rodrigo de Toledo, Bin Wang, and Bruno Levy. Geometry textures and applications. *Comput. Graph. Forum*, 27(8):2053–2065, 2008. ISSN 0167-7055. doi: http://dx.doi.org/10.1111/j.1467-8659.2008.01185.x. URL http://dx.doi.org/10.1111/j.1467-8659.2008.01185.x.

[22] P. Degener, J. Meseth, and R. Klein. An adaptable surface parameterization method. In *In Proceedings of the 12th International Meshing Roundtable*, pages 201–213, 2003.

[23] Miquel Feixas, Mateu Sbert, and Francisco González. A unified information-theoretic framework for viewpoint selection and mesh saliency. *ACM Trans. Appl. Percept.*, 6(1):1:1–1:23, February 2009. ISSN 1544-3558.

[24] Michael S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20(1):19–27, 2003. ISSN 0167-8396. doi: http://dx.doi.org/10.1016/S0167-8396(02)00002-5.

[25] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in multiresolution for geometric modelling*, pages 157–186. Springer Verlag, 2005. URL http://vcg.isti.cnr.it/Publications/2005/FH05.

[26] Michael S. Floater, Géza Kós, and Martin Reimers. Mean value coordinates in 3d. *Computer Aided Geometric Design*, 22(7):623–631, 2005.

[27] James D. Foley, Andries van Dam, Stephen K. Feiner, John F. Hughes, and R. Phillips. *Introduction to Computer Graphics*. Addison-Wesley, 1993.

[28] Blender Foundation. Sintel. http://www.sintel.org/, 2011.

[29] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Trans. Graph.*, 23(3):652–663, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015775. URL http://doi.acm.org/10.1145/1015706.1015775.

[30] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 59–64, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0.

[31] Francisco González García, Miquel Feixas, and Mateu Sbert. View-based shape similarity using mutual information spheres. eurographics'07 short paper, prague, czech republic, 2007, 2007.

[32] Francisco González and Gustavo Patow. Continuity mapping for multi-chart textures. *ACM Trans. Graph.*, 28:109:1–109:8, December 2009. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1618452.1618455. URL http://doi.acm.org/10.1145/1618452.1618455.

[33] Cindy M. Grimm and John F. Hughes. Modeling surfaces of arbitrary topology using manifolds. In *SIGGRAPH '95 Proceedings*, pages 359–368, 1995.

[34] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3):355–361, 2002. ISSN 0730-0301.

[35] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6:56–67, 1986.

[36] Jin Huang, Lu Chen, Xinguo Liu, and Hujun Bao. Efficient mesh deformation using tetrahedron control mesh. *Comput. Aided Geom. Des.*, 26(6):617–626, 2009. ISSN 0167-8396. doi: http://dx.doi.org/10.1016/j.cagd.2008.12.002.

[37] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, 30(4):78:1–78:8, 2011.

[38] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. *ACM Trans. Graph.*, 26(3):71, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276466.

[39] Tao Ju, Scott Schaefer, and Joe D. Warren. Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, 24(3):561–566, 2005.

[40] Tao Ju, Qian-Yi Zhou, Michiel van de Panne, Daniel Cohen-Or, and Ulrich Neumann. Reusable skinning templates using cage-based deformations. *ACM Trans. Graph.*, 27 (5):122:1–122:10, December 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409075. URL http://doi.acm.org/10.1145/1409060.1409075.

[41] Kazufumi Kaneda, Takushi Kagawa, and Hideo Yamashita. Animation of water droplets on a glass plate. In *Proceedings Computer Animation*, pages 177–189, 1993.

[42] Zachi Karni and Craig Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 279–286, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. doi: 10.1145/344779.344924. URL http://dx.doi.org/10.1145/344779.344924.

[43] Vladislav Kraevoy, Alla Sheffer, and Craig Gotsman. Matchmaker: constructing constrained texture maps. *ACM Trans. Graph.*, 22(3):326–333, 2003. ISSN 0730-0301.

[44] Eric Landreneau and Scott Schaefer. Poisson-based weight reduction of animated meshes. *Comput. Graph. Forum*, 29(6):1945–1954, 2010.

[45] Torsten Langer, Alexander Belyaev, and Hans-Peter Seidel. Mean value bézier maps. In *GMP'08: Proceedings of the 5th international conference on Advances in geometric modeling and processing*, pages 231–243, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79245-7, 978-3-540-79245-1.

[46] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009. ISSN 1467-8659.

[47] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM Press, 2007.

[48] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25(3):579–588, 2006.

[49] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Trans. Graph.*, 25(3):541–548, 2006.

[50] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, 2002. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/566654.566590.

[51] Zheng Li, David Levin, Zhengjie Deng, Dingyuan Liu, and Xiaonan Luo. Cage-free local deformations using green coordinates. *Vis. Comput.*, 26(6-8):1027–1036, 2010. ISSN 0178-2789. doi: http://dx.doi.org/10.1007/s00371-010-0438-x.

[52] J. Lin. Divergence measures based on the shannon entropy. *Information Theory, IEEE Transactions on*, 37(1):145 –151, jan 1991. ISSN 0018-9448.

[53] Yaron Lipman, Johannes Kopf, Daniel Cohen-Or, and David Levin. Gpu-assisted positive mean value coordinates for mesh deformations. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 117–123, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-46-3.

[54] Yaron Lipman, David Levin, and Daniel Cohen-Or. Green coordinates. *ACM Trans. Graph.*, 27(3):78:1–78:10, August 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360677. URL http://doi.acm.org/10.1145/1360612.1360677.

[55] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[56] Weiliang Meng, Bin Sheng, Shandong Wang, Hanqiu Sun, and Enhua Wu. Interactive image deformation using cage coordinates on gpu. In *VRCAI '09: Proceedings of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 119–126, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-912-1. doi: http://doi.acm.org/10.1145/1670252.1670279.

[57] Weiliang Meng, Xiaopeng Zhang, Weiming Dong, and Jean-Claude Paul. Multicage image deformation on gpu. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 155–162, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1060-4. doi: 10.1145/2087756.2087777. URL http://doi.acm.org/10.1145/2087756.2087777.

[58] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276484. URL http://doi.acm.org/10.1145/1276377.1276484.

[59] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 25–35, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-625-7.

[60] Jan Novák and Carsten Dachsbacher. Rasterized bounding volume hierarchies. *Comp. Graph. Forum*, 31(2pt2):403–412, May 2012. ISSN 0167-7055.

[61] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 7–14, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-739-7.

[62] M. M. Oliveira and F. Policarpo. An efficient representation for surface details. Technical Report RP-351, Federal University of Rio Grande do Sul - UFRGS, 2005. URL http://www.inf.ufrgs.br/~oliveira/pubs_files/Oliveira_Policarpo_RP-351_Jan_2005.pdf.

[63] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Trans. Graph.*, 24(3):445–452, July 2005. ISSN 0730-0301.

[64] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 012553180X.

[65] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 155–162, New York, NY, USA, 2005. ACM. ISBN 1-59593-013-2. doi: 10.1145/1053427.1053453. URL http://doi.acm.org/10.1145/1053427.1053453.

[66] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of ACM SIGGRAPH 2000*, pages 465–470, jul 2000.

[67] Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. Seamless texture atlases. In *SGP '04: Proceedings of the 2004 Eurographics/SIGGRAPH symposium on Geometry Processing*, pages 65–74, 2004. ISBN 3-905673-13-4.

[68] Nicolas Ray, Vincent Nivoliers, Sylvain Lefebvre, and Bruno Lévy. Invisible seams. In Jason Lawrence and Marc Stamminger, editors, *EUROGRAPHICS Symposium on Rendering Conference Proceedings*. Eurographics, Eurographics Association, 2010.

[69] Jaume Rigau, Miquel Feixas, and Mateu Sbert. Information-theory-based oracles for hierarchical radiosity. In Vipin Kumar, MarinaL. Gavrilova, ChihJengKenneth Tan, and Pierre LÕEcuyer, editors, *Computational Science and Its Applications - ICCSA 2003*, volume 2669 of *Lecture Notes in Computer Science*, pages 275–284. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40156-8.

[70] Jaume Rigau, Miquel Feixas, and Mateu Sbert. Entropy-based adaptive sampling. the 13th eurographics workshop on rendering, poster papers proceedings (pisa, italy), pp. 63-70, 2002., 2003.

[71] Marc Ruiz, Anton Bardera, Imma Boada, and Ivan Viola. Automatic transfer functions based on informational divergence. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1932–1941, December 2011. ISSN 1077-2626.

[72] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/SIGGRAPH symposium on Geometry processing*, pages 146–155, 2003. ISBN 1-58113-687-0.

[73] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 409–416, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X.

[74] Mateu Sbert, Miquel Feixas, Jaume Rigau, Miguel Chover, and Ivan Viola. *Information Theory Tools for Computer Graphics*. Synthesis Lectures on Computer Graphics and Animation. Morgan and Claypool Publishers Colorado, 2009. ISBN 1598299298.

[75] Jörg Schmittler, Sven Woop, Daniel Wagner, and Wolfgang J. Paul. Graphics hardware (2004) t. akenine-möller, m. mccool (editors) realtime ray tracing of dynamic scenes on an fpga chip.

[76] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor – a hardware architecture for ray tracing. In *Proceedings of the conference on Graphics Hardware 2002*, pages 27–36. Saarland University, Eurographics Association, 2002. ISBN 1-58113-580-7. available at http://www.openrt.de.

[77] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20:151–160, August 1986. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/15886.15903. URL http://doi.acm.org/10.1145/15886.15903.

[78] Hyewon Seo and Nadia Magnenat Thalmann. Lod management on animating face models. In *Proceedings of the IEEE Virtual Reality 2000 Conference*, VR '00, pages 161–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0478-7. URL http://dl.acm.org/citation.cfm?id=832288.835768.

[79] Ariel Shamir. A survey on mesh segmentation techniques. *Comput. Graph. Forum*, 27 (6):1539–1556, 2008.

[80] Alla Sheffer and John C. Hart. Seamster: Inconspicuous low-distortion texture seam layout. *Visualization Conference, IEEE*, pages 291–298, 2002. ISSN 1070-2385.

[81] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. 1996.

[82] Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V. Sander, Diego Nehab, and Jiahe Xi. Automated reprojection-based pixel shader optimization. *ACM Trans. Graph.*, 27(5):127:1–127:11, December 2008. ISSN 0730-0301.

[83] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Trans. Graph.*, 30(6):152:1–152:12, December 2011. ISSN 0730-0301.

[84] Noam Slonim and Naftali Tishby. Document clustering using word clusters via the information bottleneck method. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 208–215. ACM Press, 2000. Held in Athens, Greece.

[85] Olga Sorkine, Daniel Cohen-Or, Rony Goldenthal, and Dani Lischinski. Bounded-distortion piecewise mesh parameterization. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 355–362, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7803-7498-3. URL http://dl.acm.org/citation.cfm?id=602099.602154.

[86] Detlev Stalling and Hans christian Hege. Fast and intuitive generation of geometric shape transitions. *The Visual Computer*, 16:241–253, 2000.

[87] Jos Stam. Flows on surfaces of arbitrary topology. *ACM Trans. Graph.*, 22(3):724–731, 2003.

[88] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3):695–704, 2005. ISSN 1467-8659.

[89] Laszlo Szirmay-Kalos, Laszlo Szecsi, and Mateu Sbert. Gpu-based techniques for global illumination effects. In Brian Barsky, editor, *Synthesis Lectures on Computer Graphics and Animation*. Morgan-Claypool, 2008.

[90] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. *ACM Trans. Graph.*, 23(3):853–860, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015810. URL http://doi.acm.org/10.1145/1015706.1015810.

[91] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In Dieter Schmalstieg and Jiří Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007.

[92] Ofir Weber, Mirela Ben-Chen, and Craig Gotsman. Complex barycentric coordinates with applications to planar shape deformation. *Computer Graphics Forum*, 28(2):587–597, 2009. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2009.01399.x. URL http://dx.doi.org/10.1111/j.1467-8659.2009.01399.x.

[93] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *SIGGRAPH '91 Proceedings*, pages 299–308, 1991.

[94] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073211. URL http://doi.acm.org/10.1145/1073204.1073211.

[95] Lei Yang, Pedro V. Sander, and Jason Lawrence. Geometry-aware framebuffer level of detail. *Computer Graphics Forum*, 27(4):1183–1188, 2008. ISSN 1467-8659.

[96] Cem Yuksel, John Keyser, and Donald H. House. Mesh colors. Technical Report tamu-cs-tr-2008-4-1, Department of Computer Science, Texas A&M University, 2008.

[97] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27, January 2005. ISSN 0730-0301. doi: 10.1145/1037957.1037958. URL http://doi.acm.org/10.1145/1037957.1037958.

[98] Youyi Zheng, Hongbo Fu, Daniel Cohen-Or, Oscar Kin-Chung Au, and Chiew-Lan Tai. Component-wise controllers for structure-preserving shape manipulation. In *Computer Graphics Forum (In Proc. of Eurographics 2011)*, volume 30, page to appear, 2011.

[99] Kun Zhou, John Synder, Baining Guo, and Heung-Yeung Shum. Iso-charts: stretch-driven mesh parameterization using spectral analysis. In *SGP '04: Proceedings of the 2004 Eurographics/SIGGRAPH symposium on Geometry processing*, pages 45–54, 2004. ISBN 3-905673-13-4.

[100] Kun Zhou, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. TextureMontage: Seamless texturing of arbitrary surfaces from multiple images. *ACM Trans. Graph.*, 24(3):1148–1155, 2005.

[101] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008. ISSN 0730-0301.

[102] Emanoil Zuckerberger, Ayellet Tal, and Shymon Shlafman. Polyhedral surface decomposition with applications. *Computers and Graphics*, page 2002.

Part II

APPENDIX

CONTINUITY MAPPING PIXEL SHADER CODE

```
/***********************************************************************/
/*                                                                     */
/*                          PIXEL SHADER                               */
/*                                                                     */
/***********************************************************************/

// Define outputs from vertex shader.
struct VertexOut
{
  float4 outPosition : POSITION;
  float2 outTexCoords : TEXCOORD0;
};

// Define outputs from pixel shader.
struct Fragment
{
    float4 color : COLOR0;
};

Fragment main_fp(VertexOut v,
    uniform float4 p_sewingTheSeams,
    uniform float4 p_sewingTriangles,
    uniform float4 p_travellersTextureSize,
    uniform float4 p_transformationTexturesSize,
    uniform float4 p_trianglesTextureSize,ç
    uniform float4 p_verticesTextureSize,
    uniform sampler2D colorMap : register(s0),
    uniform sampler2D travellersMap : register(s1),
    uniform sampler1D transformationMap1 : register(s2),
    uniform sampler1D transformationMap2 : register(s3),
    uniform sampler2D trianglesMap : register(s4),
    uniform sampler2D verticesMap : register(s5))
{

Fragment fOut;
float3 fragmentColor;
float2 uv = float2(v.outTexCoords.x, 1.0-v.outTexCoords.y).rg;

if(p_sewingTheSeams.x || p_sewingTriangles.x)
{
    float4 texelType = tex2D(travellersMap,uv).rgba;
    float displacementTexelX = 1.0 / p_travellersTextureSize.x;
    float displacementTexelY = 1.0 / p_travellersTextureSize.y;

    //Interior fragment = GPU Filtering
    if(texelType.g==-1.0) fragmentColor = tex2D(colorMap,uv).rgb;
    else
    {
        //Sewing The Seams fragment
        float4 subTexelInfo=unpackSubtexelInformation(texelType.a);
```

```
if(fragmentOnSewingTriangles(uv,displacementTexelX,displacementTexelY,
    subTexelInfo))
{
    float2 v1,v2,v3;
    float idxTriangle,currentTriangle,nTriangles;
    float trobat,displacementVertices,displacementTriangles,inTriangle;
    float2 texCoordsCurrentTriangle, texCoordsCurrentVertice;
    float3 coordinates;
    float2 transformedV1, transformedV2,transformedV3;
    float2 color1TexCoords,color2TexCoords,color3TexCoords;

    color1TexCoords=float2(-1.0,-1.0); color2TexCoords=float2(-1.0,-1.0)
        ; color3TexCoords=float2(-1.0,-1.0);

    displacementVertices = 1.0 / p_verticesTextureSize.x;
    displacementTriangles = 1.0 / p_trianglesTextureSize.x;

    //Obtain texture coordinates
    texCoordsCurrentTriangle = get2DCoord(texelType.g,
        p_trianglesTextureSize.x);

    //Get the number of triangles
    nTriangles = tex2D(trianglesMap,texCoordsCurrentTriangle).r;

    updateIndexTexCoords(texCoordsCurrentTriangle,displacementTriangles,
        p_trianglesTextureSize.x);
    currentTriangle = tex2D(trianglesMap,texCoordsCurrentTriangle).r;

    idxTriangle = 0.0;
    trobat = 0.0;

    fragmentColor = float3(0.0,0.0,0.0);

    while((idxTriangle<nTriangles) && (!trobat))
    {
        //Obtain texcoords from the current triangle in the vertices
            list
        texCoordsCurrentVertice = get2DCoord(currentTriangle,
            p_verticesTextureSize.x);

        //Get vertices from the current triangle
        getTraingleVertices(texelType.b,
            verticesMap,transformationMap1,transformationMap2,
            texCoordsCurrentVertice,displacementVertices,
            p_verticesTextureSize.x,p_transformationTexturesSize.x,
            transformedV1,transformedV2,transformedV3,
            v1,v2,v3,color1TexCoords,color2TexCoords,color3TexCoords);

        inTriangle = pointInTriangle(uv,transformedV1,transformedV2,
            transformedV3);

        if(inTriangle)
        {
            trobat = 1.0;

            fragmentColor = evaluateColor(
                uv,v1,v2,v3,
```

```
                                                transformedV1,transformedV2,
                                                   transformedV3,
                            color1TexCoords,color2TexCoords,color3TexCoords,
                            verticesMap,colorMap,p_verticesTextureSize.x,coordinates
                               );

                        if(p_sewingTriangles.x) fragmentColor = coordinates;
                    }
                    else
                    {
                        idxTriangle = idxTriangle + 1;
                        updateIndexTexCoords(
                            texCoordsCurrentTriangle,
                            displacementTriangles,
                            p_trianglesTextureSize.x);

                        currentTriangle = tex2D(trianglesMap,
                            texCoordsCurrentTriangle).r;
                    }
                }

                if(!trobat) fragmentColor = tex2D(colorMap,uv).rgb;
            }
            else fragmentColor = tex2D(colorMap,uv).rgb;
        }
        else fragmentColor = tex2D(colorMap,uv).rgb;

        fOut.color = float4(fragmentColor,1.0);

        return fOut;
}

/**************************************************************************/
/*                                                                        */
/*                              FUNCTIONS                                 */
/*                                                                        */
/**************************************************************************/

float2 get2DCoord(float pos,float texwidth)
{
float2 uv;

// Rectangular coordinate => ( 0..texwidth, 0..texwidth )
uv.y = floor( pos / texwidth );
uv.x = pos - texwidth * uv.y;         // pos % texwidth

// Convert to uniform space / normalize => ( 0..1, 0..1 )
uv = ( uv + 0.5 ) / texwidth;

return uv;
}

float3 getBarycentricCoordinatesColor(
    in float2 point,
    in float2 point1,
    in float2 point2,
    in float2 point3,
    in float3 color1,
```

```
    in float3 color2,
    in float3 color3,
    out float3 coordinates)
{

float A, B, C, D, E, F;
float lambda1, lambda2, lambda3;

A=point1.x-point3.x;
B=point2.x-point3.x;
C=point3.x-point.x;
D=point1.y-point3.y;
E=point2.y-point3.y;
F=point3.y-point.y;

if(A==0.0 && B==0.0)
{
    float aux;

    aux=A;
    A=D;
    D=aux;

    aux=B;
    B=E;
    E=aux;

    aux=C;
    C=F;
    F=aux;
}

lambda1=((B*F)-(C*E)) / ((A*E) - (B*D));
lambda2=((A*F)-(C*D)) / ((B*D)-(A*E));
lambda3=1.0-lambda1-lambda2;

coordinates = float3(lambda1, lambda2, lambda3);

return (lambda1 * color1 + lambda2 * color2 + lambda3 * color3);
}

float2 transformPoint(
    in float texelSeamID,
    in float3 p,
    in sampler1D transformationMap1,
    in sampler1D transformationMap2,
    in float transformationMapsSize)
{

float3 part1, part2;
float2 result;
float3x3 M;

if(p.z!=texelSeamID)
{
    part1 = tex1D(transformationMap1,(p.z + 0.5) / transformationMapsSize).rgb;

    part2 = tex1D(transformationMap2,(p.z + 0.5) / transformationMapsSize).rgb;
```

```
    M=float3x3(part1,part2,float3(0.0,0.0,1.0));

    result = mul(M,float3(p.x,1.0-p.y,1.0)).xy;
    result.y=1.0-result.y;
}
else result = p.xy;

return result;
}

float pointInTriangle(
    in float2 P,
    in float2 A,
    in float2 B,
    in float2 C)
{

//float2 A,B,C;
float2 v0, v1, v2;
float dot00, dot01, dot02, dot11, dot12;
float invDenom;
float u,v;

// Compute vectors
v0 = C - A;
v1 = B - A;
v2 = P - A;

// Compute dot products
dot00 = dot(v0, v0);
dot01 = dot(v0, v1);
dot02 = dot(v0, v2);
dot11 = dot(v1, v1);
dot12 = dot(v1, v2);

// Compute barycentric coordinates
invDenom = 1.0 / (dot00 * dot11 - dot01 * dot01);
u = (dot11 * dot02 - dot01 * dot12) * invDenom;
v = (dot00 * dot12 - dot01 * dot02) * invDenom;

// Check if point is in triangle
if((u > 0.0) && (v > 0.0) && (u + v < 1.0)) return 1.0;
else return 0.0;

}

void updateIndexTexCoords(
    inout float2 currentTexCoords,
    in float displacement,
    in float size)
{

currentTexCoords = currentTexCoords + float2(displacement,0.0);

if(currentTexCoords.x>1.0) currentTexCoords = float2(0.5/size,currentTexCoords.y
    +displacement);
```

```
}

void getTraingleVertices(
    in float texelType,
    in sampler2D verticesMap,
    in sampler1D transformationMap1,
    in sampler1D transformationMap2,
    inout float2 texCoordsCurrentVertice,
    in float displacementVertices,
    in float verticesTextureSize,
    in float transformationTextureSize,
    out float2 transformedV1,
    out float2 transformedV2,
    out float2 transformedV3,
    inout float2 v1Color,
    inout float2 v2Color,
    inout float2 v3Color,
    inout float2 color1TexCoords,
    inout float2 color2TexCoords,
    inout float2 color3TexCoords)
{

float3 v1,v2,v3;

//Vertex 1
v1 = tex2D(verticesMap,texCoordsCurrentVertice).rgb;
v1Color=v1.xy;
if(v1.z>=0.0)
{
    //Shared vertice
    transformedV1 = transformPoint(texelType,v1,transformationMap1,
        transformationMap2,transformationTextureSize);
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color1TexCoords=float2(-1.0,-1.0);
}
else if(v1.z==-1.0)
{
    //Join vertice
    transformedV1 = v1.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color1TexCoords=float2(-1.0,-1.0);
}
else
{
    //Intersection vertex or Seam vertex
    transformedV1 = v1.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color1TexCoords=texCoordsCurrentVertice;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize); updateIndexTexCoords(texCoordsCurrentVertice,
        displacementVertices,verticesTextureSize);
}

//Vertex 2
v2 = tex2D(verticesMap,texCoordsCurrentVertice).rgb;
```

```
v2Color=v2.xy;
if(v2.z>=0.0)
{
    //Shared vertice
    transformedV2 = transformPoint(texelType,v2,transformationMap1,
        transformationMap2,transformationTextureSize);
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color2TexCoords=float2(-1.0,-1.0);
}
else if(v2.z==-1.0)
{
    //Join vertice
    transformedV2 = v2.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color2TexCoords=float2(-1.0,-1.0);
}
else
{
    //Intersection vertex or Seam vertex
    transformedV2 = v2.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color2TexCoords=texCoordsCurrentVertice;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize); updateIndexTexCoords(texCoordsCurrentVertice,
        displacementVertices,verticesTextureSize);
}

//Vertex 3
v3 = tex2D(verticesMap,texCoordsCurrentVertice).rgb;
v3Color=v3.xy;
if(v3.z>=0.0)
{
    //Shared vertice
    transformedV3 = transformPoint(texelType,v3,transformationMap1,
        transformationMap2,transformationTextureSize);
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color3TexCoords=float2(-1.0,-1.0);
}
else if(v3.z==-1.0)
{
    //Join vertice
    transformedV3 = v3.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color3TexCoords=float2(-1.0,-1.0);
}
else
{
    //Intersection vertex or Seam vertex
    transformedV3 = v3.xy;
    updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
        verticesTextureSize);
    color3TexCoords=texCoordsCurrentVertice;
```

```
        updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
            verticesTextureSize);
        updateIndexTexCoords(texCoordsCurrentVertice,displacementVertices,
            verticesTextureSize);
    }

    }

    float3 getWeightedVerticeColor(
        in float2 colorTexCoords,
        in sampler2D verticesMap,in sampler2D colorMap,
        in float verticesTextureSize)
    {
    float3 infoV1,infoV2;
    float3 color1,color2;
    float2 texCoords;

    texCoords=colorTexCoords;

    infoV1 = tex2D(verticesMap,colorTexCoords).rgb;
    updateIndexTexCoords(texCoords,1.0/verticesTextureSize,verticesTextureSize);
    color1 = tex2D(colorMap,infoV1.xy).rgb;

    infoV2 = tex2D(verticesMap,texCoords).rgb;
    color2 = tex2D(colorMap,infoV2.xy).rgb;

    return ((infoV1.z * color1) + (infoV2.z * color2));
    }

    float3 evaluateColor(
        in float2 uv,
        in float2 v1, in float2 v2, in float2 v3,
        in float2 transformedV1, in float2 transformedV2, in float2 transformedV3,
        in float2 color1TexCoords,in float2 color2TexCoords,in float2
            color3TexCoords,
        in sampler2D verticesMap, in sampler2D colorMap, in float
            verticesTextureSize,
        inout float3 coordinates)
    {

    float3 color1,color2,color3;

    if(color1TexCoords.x>0)
    {
        color1 = getWeightedVerticeColor(color1TexCoords,verticesMap,colorMap,
            verticesTextureSize);
    }
    else color1 = tex2D(colorMap,v1).rgb;

    if(color2TexCoords.x>0)
    {
        color2 = getWeightedVerticeColor(color2TexCoords,verticesMap,colorMap,
            verticesTextureSize);
    }
    else color2 = tex2D(colorMap,v2).rgb;

    if(color3TexCoords.x>0)
    {
```

```
        color3 = getWeightedVerticeColor(color3TexCoords,verticesMap,colorMap,
            verticesTextureSize);
}
else color3 = tex2D(colorMap,v3).rgb;

return getBarycentricCoordinatesColor(uv,transformedV1,transformedV2,
    transformedV3,color1,color2,color3,coordinates);
}

float4 unpackSubtexelInformation(float info)
{
int num;
float4 result;

num = (int)info;

result.w=fmod((float)num,2.0);
num=num/2;

result.z=fmod((float)num,2.0);
num=num/2;

result.y=fmod((float)num,2.0);
result.x=num/2;

return result;
}

float fragmentOnSewingTriangles(in float2 texCoords,in float displacementTexelX,
    in float displacementTexelY,float4 subTexelInfo)
{
float2 currentTexel;

int currentTexelX = texCoords.x / displacementTexelX;
int currentTexelY = texCoords.y / displacementTexelY;

currentTexel.x = (currentTexelX * displacementTexelX) + (displacementTexelX /
    2.0);
currentTexel.y = (currentTexelY * displacementTexelY) + (displacementTexelY /
    2.0);

//Determine the cuadrant of the current fragment within the texel it belongs
if(texCoords.x > currentTexel.x)
{
    if(texCoords.y > currentTexel.y) return subTexelInfo.z;        //3rd
        cuadrant
    else return subTexelInfo.w;                                     //4th
        cuadrant
}
else
{
    if(texCoords.y > currentTexel.y) return subTexelInfo.y;            //2nd
        cuadrant
    else return subTexelInfo.x;                                  //1rst cuadrant
}
}
```

B

I-RENDER CUDA KERNELS

```
/***********************************************************************/
/*                                                                     */
/*                       TRAVERSAL KERNELS                        */
/*                                                                     */
/***********************************************************************/

__device__ inline void traceIRender1Pass(
    /* primary / secondary ray */
    int isPrimaryRay,
    /* compute or not shadow rays */
    int computeShadow,
    /* Total number of rays in the batch. */
    int numRays,
    /* Position of the first primary ray.*/
    int firstPrimaryRay,
    /* Number of samples per ray for secondary rays to know how many of them
       belong to the same pixel. */
    int numSamplesPerRay,
    /* False if rays need to find the closest hit. */
    bool anyHit,
    /* For secondary Rays. */
    int randomSeedAO,
    /* For secondary Rays. */
    float radiusAO,
    /* Light position X component */
    float lightPositionX,
    /* Light position Y component */
    float lightPositionY,
    /* Light position Z component */
    float lightPositionZ,
    /* Light radius */
    float lightRadius,
    /* Shadow samples per ray */
    int shadowSamples,
    /* Near plane distance */
    float nearPlaneDistance,
    /* Far plane distance*/
    float farPlaneDistance,
    /* Ray input: float3 origin, float tmin, float3 direction, float tmax. */
    float4* rays,
    /* Ray output: int triangleID, float hitT, int2 padding. */
    int* results,
    /* SOA: bytes 0-15 of each node, AOS/Compact: 64 bytes per node. */
    float4* nodesA,
    /* SOA: bytes 16-31 of each node, AOS/Compact: unused. */
    float4* nodesB,
    /* SOA: bytes 32-47 of each node, AOS/Compact: unused. */
    float4* nodesC,
    /* SOA: bytes 48-63 of each node, AOS/Compact: unused. */
    float4* nodesD,
```

```
    /* SOA: bytes 0-15 of each triangle, AOS: 64 bytes per triangle, Compact: 48
         bytes per triangle. */
    float4* trisA,
    /* SOA: bytes 16-31 of each triangle, AOS/Compact: unused. */
    float4* trisB,
    /* SOA: bytes 32-47 of each triangle, AOS/Compact: unused. */
    float4* trisC,
    /* Triangles indices */
    int* triIndices,
    /* To know the correspondence pixel / taskID */
    int* slotToID,
    /* Scene triangles */
    void* sceneTriangles)
{
int nAOIntersections = 0;
int nShadowIntersections = 0;
F32 epsilon = 1.0e-4f;
Vec3f origin = Vec3f(0.0, 0.0, 0.0);

// Pick ray index.
int rayidx = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * (blockIdx.x
    + gridDim.x * blockIdx.y));
if (rayidx >= numRays) return;

// Get the current pixel in 1D (correspondence taskID / pixel)
int pixel1DCurrentRes = ((const signed int*)slotToID)[rayidx];

// Get current primary ray
float4 o = rays[rayidx * 2 + 0];
float4 d = rays[rayidx * 2 + 1];

// Travers current primary ray
Vec4f traversalResult = rayTraversal(anyHit, o, d, nodesA, trisA, triIndices);

// A triangle has been intersected and we have to compute AO rationIntersections
if(traversalResult.x >= 0)
{
    origin = Vec3f(o.x, o.y, o.z) + Vec3f(d.x, d.y, d.z) * fmaxf(traversalResult
        .y - epsilon, 0.0f);

    if(!isPrimaryRay)
    {
        nAOIntersections = getAOIntersections(pixel1DCurrentRes, o, d,
            traversalResult, randomSeedAO, numSamplesPerRay, radiusAO, nodesA,
            trisA, triIndices, sceneTriangles);
    }
    if(computeShadow)
    {
        nShadowIntersections = getShadowRatio(pixel1DCurrentRes, o, d,
            traversalResult, randomSeedAO, shadowSamples, Vec3f(lightPositionX,
            lightPositionY, lightPositionZ), lightRadius, nodesA, trisA,
            triIndices, sceneTriangles);
    }
}

// Store the result of the traversal
results[rayidx * 9 + 0] = (int)traversalResult.x;
results[rayidx * 9 + 1] = __float_as_int(traversalResult.y);
```

```
results[rayidx * 9 + 2] = __float_as_int(traversalResult.z);
results[rayidx * 9 + 3] = __float_as_int(traversalResult.w);
results[rayidx * 9 + 4] = nAOIntersections;
results[rayidx * 9 + 5] = nShadowIntersections;
results[rayidx * 9 + 6] = __float_as_int(origin.x);
results[rayidx * 9 + 7] = __float_as_int(origin.y);
results[rayidx * 9 + 8] = __float_as_int(origin.z);
}

//------------------------------------------------------------------------

void tracetraceIRenderNPass(
    /* primary / secondary ray */
    int isPrimaryRay,
    /* compute or not shadow rays */
    int computeShadow,
    /* Shading with texture color */
    int texColor,
    /* Shaindg the cluster color */
    int renderCluster,
    /* Total number of rays in the batch. */
    int numRays,
    /* Number of samples per ray for secondary rays to know how many of them
       belong to the same pixel. */
    int numSamplesPerRay,
    /* Position of the first primary ray for batching them. */
    int firstPrimaryRay,
    /* Size of the previous buffer */
    int previousSize,
    /* Size of the current buffer */
    int currentSize,
    /* False if rays need to find the closest hit. */
    bool anyHit,
    /* For secondary Rays. */
    int randomSeedAO,
    /* For secondary Rays. */
    float radiusAO,
    /* Light position X component */
    float lightPositionX,
    /* Light position Y component */
    float lightPositionY,
    /* Light position Z component */
    float lightPositionZ,
    /* Light radius */
    float lightRadius,
    /* Red component of the diffuse color of the current mesh */
    float diffuseColorR,
    /* Red component of the diffuse color of the current mesh */
    float diffuseColorG,
    /* Red component of the diffuse color of the current mesh */
    float diffuseColorB,
    /* Shadow samples per ray */
    int shadowSamples,
    /* Near plane distance */
    float nearPlaneDistance,
    /* Far plane distance*/
    float farPlaneDistance,
    /* Ray input: float3 origin, float tmin, float3 direction, float tmax. */
```

```
    float4* rays,
    /* Ray output: int triangleID, float hitT, int2 padding. */
    int* results,
    /* SOA: bytes 0-15 of each node, AOS/Compact: 64 bytes per node. */
    float4* nodesA,
    /* SOA: bytes 16-31 of each node, AOS/Compact: unused. */
    float4* nodesB,
    /* SOA: bytes 32-47 of each node, AOS/Compact: unused. */
    float4* nodesC,
    /* SOA: bytes 48-63 of each node, AOS/Compact: unused. */
    float4* nodesD,
    /* SOA: bytes 0-15 of each triangle, AOS: 64 bytes per triangle, Compact: 48
         bytes per triangle. */
    float4* trisA,
    /* SOA: bytes 16-31 of each triangle, AOS/Compact: unused. */
    float4* trisB,
    /* SOA: bytes 32-47 of each triangle, AOS/Compact: unused. */
    float4* trisC,
    /* Triangles indices */
    int* triIndices,
    /* Previous color image */
    unsigned int* previousImage,
    /* Current color image */
    unsigned int* currentImage,
    /* Previous clusterID buffer */
    float4* previousClusterID,
    /* Previous clusterID buffer */
    float4* previousClusterID2,
    /* Previous clusterID buffer */
    float4* previousClusterID3,
    /* Current clusterID buffer */
    float4* currentClusterID,
    /* Current clusterID buffer */
    float4* currentClusterID2,
    /* Current clusterID buffer */
    float4* currentClusterID3,
    /* To know the correspondence pixel / taskID */
    int* slotToID,
    /* Scene triangles */
    void* sceneTriangles)
{
uint2 pixel2DCurrentRes;
int pixel1DCurrentRes;
bool interpolatePixel = false;
F32 epsilon = 1.0e-4f;
Vec3f origin = Vec3f(0.0, 0.0, 0.0);

// Pick ray index.
int rayidx = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * (blockIdx.x
    + gridDim.x * blockIdx.y));
if (rayidx >= numRays) return;

// Get the current pixel in 1D (correspondence taskID / pixel)
pixel1DCurrentRes = ((const signed int*)slotToID)[rayidx];

// Get the current pixel in 2D coordinates
pixel2DCurrentRes.x = pixel1DCurrentRes % currentSize;
pixel2DCurrentRes.y = pixel1DCurrentRes / currentSize;
```

```
bool parX = pixel2DCurrentRes.x % 2 == 0;
bool parY = pixel2DCurrentRes.y % 2 == 0;

if(parX && parY)
{
    // COPY directly from previous resolution
    int pixelPreviousRes1D;
    uint2 pixelPreviousRes2D;

    pixelPreviousRes2D.x = pixel2DCurrentRes.x / 2;
    pixelPreviousRes2D.y = pixel2DCurrentRes.y / 2;
    pixelPreviousRes1D = pixelPreviousRes2D.y * previousSize +
        pixelPreviousRes2D.x;

    interpolatePixel = true;

    currentImage[pixel1DCurrentRes] = previousImage[pixelPreviousRes1D];

    currentClusterID[pixel1DCurrentRes] = previousClusterID[pixelPreviousRes1D];
    currentClusterID2[pixel1DCurrentRes] = previousClusterID2[pixelPreviousRes1D
        ];
    currentClusterID3[pixel1DCurrentRes] = previousClusterID3[pixelPreviousRes1D
        ];
}

if (parX && !parY)
{
    int neighbour1High1D, neighbour2High1D;
    uint2 neighbour1High2D, neighbour2High2D;
    int clusterID1, clusterID2;

        //Direct neighbours pattern
        neighbour1High2D.x = pixel2DCurrentRes.x / 2;
        neighbour1High2D.y = (pixel2DCurrentRes.y - 1) / 2;
        neighbour2High2D.x = pixel2DCurrentRes.x / 2;
        neighbour2High2D.y = (pixel2DCurrentRes.y + 1) / 2;

        neighbour1High1D = neighbour1High2D.y * previousSize + neighbour1High2D.
            x;
        neighbour2High1D = neighbour2High2D.y * previousSize + neighbour2High2D.
            x;

        clusterID1 = (int)previousClusterID[neighbour1High1D].x;
        clusterID2 = (int)previousClusterID[neighbour2High1D].x;

        if(clusterID1 == clusterID2)
        {
                interpolatePixel = true;

        // Interpolate information and store the new one in the iBuffer
                currentClusterID[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID);
                currentClusterID2[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID2);
                currentClusterID3[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID3);
```

```
                // Store the color for the current pixel --> Get the pixel
                    interpolated values and compute its shading
                currentImage[pixel1DCurrentRes] = interpolateColorBuffer(
                    clusterID1, isPrimaryRay, computeShadow, numSamplesPerRay,
                    shadowSamples, texColor, renderCluster, Vec3f(1.0, 1.0, 1.0),
                     Vec3f(diffuseColorR, diffuseColorG, diffuseColorB),
                    lightPositionX, lightPositionY, lightPositionZ,
                    currentClusterID[pixel1DCurrentRes], currentClusterID2[
                    pixel1DCurrentRes], currentClusterID3[pixel1DCurrentRes]);
        }
}

if (!parX && parY)
{
        int neighbour1High1D, neighbour2High1D;
        uint2 neighbour1High2D, neighbour2High2D;
        int clusterID1, clusterID2;

        //Direct neighbours pattern
        neighbour1High2D.x = (pixel2DCurrentRes.x - 1) / 2;
        neighbour1High2D.y = pixel2DCurrentRes.y / 2;
        neighbour2High2D.x = (pixel2DCurrentRes.x + 1) / 2;
        neighbour2High2D.y = pixel2DCurrentRes.y / 2;

        neighbour1High1D = neighbour1High2D.y * previousSize + neighbour1High2D.
            x;
        neighbour2High1D = neighbour2High2D.y * previousSize + neighbour2High2D.
            x;

        clusterID1 = (int)previousClusterID[neighbour1High1D].x;
        clusterID2 = (int)previousClusterID[neighbour2High1D].x;

        if((clusterID1 == clusterID2))
        {
                interpolatePixel = true;

                // Interpolate information and store the new one in the iBuffer
                currentClusterID[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID);
                currentClusterID2[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID2);
                currentClusterID3[pixel1DCurrentRes] = interpolateBuffer2(
                    neighbour1High1D, neighbour2High1D, previousClusterID3);

                // Store the color for the current pixel --> Get the pixel
                    interpolated values and compute its shading
                currentImage[pixel1DCurrentRes] = interpolateColorBuffer(
                    clusterID1, isPrimaryRay, computeShadow, numSamplesPerRay,
                    shadowSamples, texColor, renderCluster, Vec3f(1.0, 1.0, 1.0),
                     Vec3f(diffuseColorR, diffuseColorG, diffuseColorB),
                    lightPositionX, lightPositionY, lightPositionZ,
                    currentClusterID[pixel1DCurrentRes], currentClusterID2[
                    pixel1DCurrentRes], currentClusterID3[pixel1DCurrentRes]);

        }
}

if (!parX && !parY)
```

```
{
        int clusterID1, clusterID2, clusterID3, clusterID4;
        int neighbour1High1D, neighbour2High1D, neighbour3High1D,
            neighbour4High1D;
        uint2 neighbour1High2D, neighbour2High2D, neighbour3High2D,
            neighbour4High2D;

        //Direct neighbours pattern
        neighbour1High2D.x = (pixel2DCurrentRes.x - 1) / 2;
        neighbour1High2D.y = (pixel2DCurrentRes.y - 1) / 2;
        neighbour2High2D.x = (pixel2DCurrentRes.x + 1) / 2;
        neighbour2High2D.y = (pixel2DCurrentRes.y - 1) / 2;
        neighbour3High2D.x = (pixel2DCurrentRes.x - 1) / 2;
        neighbour3High2D.y = (pixel2DCurrentRes.y + 1) / 2;
        neighbour4High2D.x = (pixel2DCurrentRes.x + 1) / 2;
        neighbour4High2D.y = (pixel2DCurrentRes.y + 1) / 2;

        neighbour1High1D = neighbour1High2D.y * previousSize + neighbour1High2D.
            x;
        neighbour2High1D = neighbour2High2D.y * previousSize + neighbour2High2D.
            x;
        neighbour3High1D = neighbour3High2D.y * previousSize + neighbour3High2D.
            x;
        neighbour4High1D = neighbour4High2D.y * previousSize + neighbour4High2D.
            x;

        clusterID1 = (int)previousClusterID[neighbour1High1D].x;
        clusterID2 = (int)previousClusterID[neighbour2High1D].x;
        clusterID3 = (int)previousClusterID[neighbour3High1D].x;
        clusterID4 = (int)previousClusterID[neighbour4High1D].x;

        if((clusterID1 == clusterID2) && (clusterID1 == clusterID3) && (
            clusterID1 == clusterID4))
        {
                interpolatePixel = true;

                // Interpolate information and store the new one in the iBuffer
                currentClusterID[pixel1DCurrentRes] = interpolateBuffer4(
                    neighbour1High1D, neighbour2High1D, neighbour3High1D,
                    neighbour4High1D, previousClusterID);
                currentClusterID2[pixel1DCurrentRes] = interpolateBuffer4(
                    neighbour1High1D, neighbour2High1D, neighbour3High1D,
                    neighbour4High1D, previousClusterID2);
                currentClusterID3[pixel1DCurrentRes] = interpolateBuffer4(
                    neighbour1High1D, neighbour2High1D, neighbour3High1D,
                    neighbour4High1D, previousClusterID3);

                // Store the color for the current pixel --> Get the pixel
                    interpolated values and compute its shading
                currentImage[pixel1DCurrentRes] = interpolateColorBuffer(
                    clusterID1, isPrimaryRay, computeShadow, numSamplesPerRay,
                    shadowSamples, texColor, renderCluster, Vec3f(1.0, 1.0, 1.0),
                     Vec3f(diffuseColorR, diffuseColorG, diffuseColorB),
                    lightPositionX, lightPositionY, lightPositionZ,
                    currentClusterID[pixel1DCurrentRes], currentClusterID2[
                    pixel1DCurrentRes], currentClusterID3[pixel1DCurrentRes]);
        }
}
```

```
if(!interpolatePixel)
{
        int nAOIntersections = 0;
        int nShadowIntersections = 0;

        // Get current primary ray
        float4 o = rays[rayidx * 2 + 0];
        float4 d = rays[rayidx * 2 + 1];

        // Travers current primary ray
        Vec4f traversalResult = rayTraversal(anyHit, o, d, nodesA, trisA,
            triIndices);

        if(traversalResult.x >= 0)
        {
                origin = Vec3f(o.x, o.y, o.z) + Vec3f(d.x, d.y, d.z) * fmaxf(
                    traversalResult.y - epsilon, 0.0f);

                if(!isPrimaryRay)
                {
                        nAOIntersections = getAOIntersections(pixel1DCurrentRes,
                             o, d, traversalResult, randomSeedAO,
                            numSamplesPerRay, radiusAO, nodesA, trisA, triIndices
                            , sceneTriangles);
                }

                if(computeShadow)
                {
            nShadowIntersections = getShadowRatio(pixel1DCurrentRes, o, d,
                traversalResult, randomSeedAO, shadowSamples, Vec3f(
                lightPositionX, lightPositionY, lightPositionZ), lightRadius,
                nodesA, trisA, triIndices, sceneTriangles);
        }
    }

    // Store the result of the traversal
        results[rayidx * 9 + 0] = (int)traversalResult.x;
        results[rayidx * 9 + 1] = __float_as_int(traversalResult.y);
        results[rayidx * 9 + 2] = __float_as_int(traversalResult.z);
        results[rayidx * 9 + 3] = __float_as_int(traversalResult.w);
        results[rayidx * 9 + 4] = nAOIntersections;
        results[rayidx * 9 + 5] = nShadowIntersections;
        results[rayidx * 9 + 6] = __float_as_int(origin.x);
        results[rayidx * 9 + 7] = __float_as_int(origin.y);
        results[rayidx * 9 + 8] = __float_as_int(origin.z);
}
else
{
        //STORE_RESULT(rayidx, -2, 0.0f, 0.0f, 0.0f);
        results[rayidx * 9 + 0] = -2;
        results[rayidx * 9 + 1] = 0;
        results[rayidx * 9 + 2] = 0;
        results[rayidx * 9 + 3] = 0;
        results[rayidx * 9 + 4] = 0;
        results[rayidx * 9 + 5] = 0;
        results[rayidx * 9 + 6] = __float_as_int(origin.x);
        results[rayidx * 9 + 7] = __float_as_int(origin.y);
```

```
        results[rayidx * 9 + 8] = __float_as_int(origin.z);
}
}

/***************************************************************************/
/*                                                                         */
/*                              FUNCTIONS                                  */
/*                                                                         */
/***************************************************************************/

__device__ inline Vec4f rayTraversal(bool anyHit, float4 o, float4 d, float4*
    nodesA, float4* trisA, int* triIndices)
{

// Traversal stack in CUDA thread-local memory.
int traversalStack[STACK_SIZE];

// Return the result of the traversal
Vec4f traversalResult;

// Live state during traversal, stored in registers.
float   origx, origy, origz;    // Ray origin.
float   dirx, diry, dirz;       // Ray direction.
float   tmin;                   // t-value from which the ray starts. Usually 0.
float   idirx, idiry, idirz;    // 1 / dir
float   oodx, oody, oodz;       // orig / dir
char*   stackPtr;               // Current position in traversal stack.
int     leafAddr;               // First postponed leaf, non-negative if none.
int     nodeAddr;               // Non-negative: current internal node, negative
    : second postponed leaf.
int     hitIndex;               // Triangle index of the closest intersection,
    -1 if none.
float   hitT;                   // t-value of the closest intersection.

// Fetch ray.
origx = o.x, origy = o.y, origz = o.z;
dirx = d.x, diry = d.y, dirz = d.z;
tmin = o.w;

float ooeps = exp2f(-80.0f); // Avoid div by zero.
idirx = 1.0f / (fabsf(d.x) > ooeps ? d.x : copysignf(ooeps, d.x));
idiry = 1.0f / (fabsf(d.y) > ooeps ? d.y : copysignf(ooeps, d.y));
idirz = 1.0f / (fabsf(d.z) > ooeps ? d.z : copysignf(ooeps, d.z));
oodx = origx * idirx, oody = origy * idiry, oodz = origz * idirz;

// Setup traversal.
traversalStack[0] = EntrypointSentinel; // Bottom-most entry.
stackPtr = (char*)&traversalStack[0];
leafAddr = 0;   // No postponed leaf.
nodeAddr = 0;   // Start from the root.
hitIndex = -1;  // No triangle intersected so far.
hitT     = d.w; // tmax

// Traversal loop.
float outU, outV;
while (nodeAddr != EntrypointSentinel)
{
    // Traverse internal nodes until all SIMD lanes have found a leaf.
```

```
bool searchingLeaf = true;
while (nodeAddr >= 0 && nodeAddr != EntrypointSentinel)
{
        // Fetch AABBs of the two child nodes.
        float4* ptr = (float4*)((char*)nodesA + nodeAddr);
        float4 n0xy = ptr[0]; // (c0.lo.x, c0.hi.x, c0.lo.y, c0.hi.y)
        float4 n1xy = ptr[1]; // (c1.lo.x, c1.hi.x, c1.lo.y, c1.hi.y)
        float4 nz   = ptr[2]; // (c0.lo.z, c0.hi.z, c1.lo.z, c1.hi.z)

        // Intersect the ray against the child nodes.
        float c0lox = n0xy.x * idirx - oodx;
        float c0hix = n0xy.y * idirx - oodx;
        float c0loy = n0xy.z * idiry - oody;
        float c0hiy = n0xy.w * idiry - oody;
        float c0loz = nz.x   * idirz - oodz;
        float c0hiz = nz.y   * idirz - oodz;
        float c1loz = nz.z   * idirz - oodz;
        float c1hiz = nz.w   * idirz - oodz;
        float c0min = max4(fminf(c0lox, c0hix), fminf(c0loy, c0hiy),
            fminf(c0loz, c0hiz), tmin);
        float c0max = min4(fmaxf(c0lox, c0hix), fmaxf(c0loy, c0hiy),
            fmaxf(c0loz, c0hiz), hitT);
        float c1lox = n1xy.x * idirx - oodx;
        float c1hix = n1xy.y * idirx - oodx;
        float c1loy = n1xy.z * idiry - oody;
        float c1hiy = n1xy.w * idiry - oody;
        float c1min = max4(fminf(c1lox, c1hix), fminf(c1loy, c1hiy),
            fminf(c1loz, c1hiz), tmin);
        float c1max = min4(fmaxf(c1lox, c1hix), fmaxf(c1loy, c1hiy),
            fmaxf(c1loz, c1hiz), hitT);

        bool traverseChild0 = (c0max >= c0min);
        bool traverseChild1 = (c1max >= c1min);

        // Neither child was intersected => pop stack.
        if (!traverseChild0 && !traverseChild1)
        {
                nodeAddr = *(int*)stackPtr;
                stackPtr -= 4;
        }
        else  // Otherwise => fetch child pointers.
        {
                int2 cnodes = *(int2*)&ptr[3];
                nodeAddr = (traverseChild0) ? cnodes.x : cnodes.y;

                // Both children were intersected => push the farther
                    one.
                if (traverseChild0 && traverseChild1)
                {
                if (c1min < c0min)
                                ::swap(nodeAddr, cnodes.y);
                                stackPtr += 4;
                                *(int*)stackPtr = cnodes.y;
                }
        }

        // First leaf => postpone and continue traversal.
        if (nodeAddr < 0 && leafAddr >= 0)
```

```
        {
                searchingLeaf = false;
                leafAddr = nodeAddr;
                nodeAddr = *(int*)stackPtr;
                stackPtr -= 4;
        }
        // All SIMD lanes have found a leaf => process them.
        if (!__any(searchingLeaf)) break;
}

// Process postponed leaf nodes.
while (leafAddr < 0)
{
        // Intersect the ray against each triangle using Sven Woop's
            algorithm.
        for (int triAddr = ~leafAddr;; triAddr += 3)
        {
                // Read first 16 bytes of the triangle.
                // End marker (negative zero) => all triangles processed
                    .
                float4 v00 = tex1Dfetch(t_trisA, triAddr + 0);
                if (__float_as_int(v00.x) == 0x80000000) break;

                // Compute and check intersection t-value.
                float Oz = v00.w - origx*v00.x - origy*v00.y - origz*v00
                    .z;
                float invDz = 1.0f / (dirx*v00.x + diry*v00.y + dirz*v00
                    .z);
                float t = Oz * invDz;

                if (t > tmin && t < hitT)
                {
                        // Compute and check barycentric u.

                        float4 v11 = tex1Dfetch(t_trisA, triAddr + 1);
                        float Ox = v11.w + origx*v11.x + origy*v11.y +
                            origz*v11.z;
                        float Dx = dirx*v11.x + diry*v11.y + dirz*v11.z;
                        float u = Ox + t*Dx;

                        if (u >= 0.0f && u <= 1.0f)
                        {
                                // Compute and check barycentric v.
                                float4 v22 = tex1Dfetch(t_trisA, triAddr
                                    + 2);
                                float Oy = v22.w + origx*v22.x + origy*
                                    v22.y + origz*v22.z;
                                float Dy = dirx*v22.x + diry*v22.y +
                                    dirz*v22.z;
                                float v = Oy + t*Dy;

                                if (v >= 0.0f && u + v <= 1.0f)
                                {
                                        // Record intersection.
                                        // Closest intersection not
                                            required => terminate.
                                        hitT = t;
                                hitIndex = triAddr;
```

```
                                                        outU = u;
                                                        outV = v;
                                                        if (anyHit)
                                                        {
                                                                nodeAddr =
                                                                    EntrypointSentinel;
                                                                break;
                                                        }
                                                }
                                        }
                                }
                        } // triangle

                        // Another leaf was postponed => process it as well.
                        leafAddr = nodeAddr;

                        if(nodeAddr<0)
                        {
                                nodeAddr = *(int*)stackPtr;
                                stackPtr -= 4;
                        }
                } // leaf
        } // traversal

        // Remap intersected triangle index, and store the result.
        if (hitIndex != -1) hitIndex = tex1Dfetch(t_triIndices, hitIndex);

        traversalResult.x = hitIndex;
        traversalResult.y = hitT;
        traversalResult.z = outU;
        traversalResult.w = outV;

        return traversalResult;
        }

        __device__ inline int getShadowRatio(int pixel1DPos, float4 o, float4 d, Vec4f
            traversalResult, int randomSeed, int numSamplesPerRay, Vec3f lightPosition,
            float lightRadius, float4* nodesA, float4* trisA, int* triIndices, void *
            sceneTriangles)
        {
        F32 epsilon = 1.0e-4f;
        Vec3f target, direction;
        float directionLenght;
        float4 originShadow, directionShadow;
        Vec4f traversalResultShadow;
        int shadowHits = 0;
        int tri = traversalResult.x;
        Vec3f normal(1.0f, 0.0f, 0.0f);
        F32 u = traversalResult.z;
        F32 v = traversalResult.w;
        F32 w = 1.0 - u - v;
        const Scene::Triangle*  triangles   = (const Scene::Triangle*)sceneTriangles;

        //The interpolated normal in the hit point of the surface
        normal.x = (u * triangles[tri].tri0_normal.x) + (v * triangles[tri].tri1_normal.
            x) + (w * triangles[tri].tri2_normal.x);
        normal.y = (u * triangles[tri].tri0_normal.y) + (v * triangles[tri].tri1_normal.
            y) + (w * triangles[tri].tri2_normal.y);
```

```
normal.z = (u * triangles[tri].tri0_normal.z) + (v * triangles[tri].tri1_normal.
    z) + (w * triangles[tri].tri2_normal.z);

Vec3f origin = Vec3f(o.x, o.y, o.z) + Vec3f(d.x, d.y, d.z) * fmaxf(
    traversalResult.y - epsilon, 0.0f);

// Pick random offset.
U32 hashA = randomSeed + pixel1DPos;
U32 hashB = 0x9e3779b9u;
U32 hashC = 0x9e3779b9u;
jenkinsMix(hashA, hashB, hashC);
jenkinsMix(hashA, hashB, hashC);
Vec3f offset((F32)hashA*exp2(-32),(F32)hashB*exp2(-32),(F32)hashC*exp2(-32));

// Only trace rays for hit positions that are facing the light ...
for (int i = 0; i < numSamplesPerRay; i++)
{
        // QMC.
        Vec3f pos(sobol2D(i),hammersley(i,numSamplesPerRay));            // [0,1]
        pos += offset;
                                                    // Cranley-Patterson
        if(pos[0]>=1.f) pos[0] -= 1.f;
        if(pos[1]>=1.f) pos[1] -= 1.f;
        if(pos[2]>=1.f) pos[2] -= 1.f;
        pos = pos*2-1;
                                                    // [-1,1]

        // Target position.
        target = lightPosition + lightRadius * pos;
        direction = target - origin;
        directionLenght = direction.length();
        direction = direction.normalized();

        if (dot(normal, direction) >= 0.0f)
        {
                originShadow.x = origin.x;
                originShadow.y = origin.y;
                originShadow.z = origin.z;
                originShadow.w = 0.0f;

        // DIRECT NORMALS: BUNNY
                directionShadow.x = direction.x;
                directionShadow.y = direction.y;
                directionShadow.z = direction.z;
                directionShadow.w = directionLenght;

        traversalResultShadow = rayTraversal(true, originShadow, directionShadow
            , nodesA, trisA, triIndices);

        if(traversalResultShadow.x >= 0.f) shadowHits = shadowHits + 1;
        }
}

return shadowHits;
}
```

```
__device__ inline int getAOIntersections(int pixel1DPos, float4 o, float4 d,
    Vec4f traversalResult, int randomSeedAO, int numSamplesPerRay, float radiusAO
    , float4* nodesA, float4* trisA, int* triIndices, void *sceneTriangles)
{
// Create info to generate all the random secondary rays
F32 epsilon = 1.0e-4f;
Vec4f traversalResultAO;
int nAOIntersections = 0;

Vec3f origin = Vec3f(o.x, o.y, o.z) + Vec3f(d.x, d.y, d.z) * fmaxf(
    traversalResult.y - epsilon, 0.0f);

// Lookup normal, flipping back-facing directions.
int tri = traversalResult.x;
Vec3f normal(1.0f, 0.0f, 0.0f);
F32 u = traversalResult.z;
F32 v = traversalResult.w;
F32 w = 1.0 - u - v;

const Scene::Triangle*  triangles  = (const Scene::Triangle*)sceneTriangles;

//The interpolated normal in the hit point of the surface
normal.x = (u * triangles[tri].tri0_normal.x) + (v * triangles[tri].tri1_normal.
    x) + (w * triangles[tri].tri2_normal.x);
normal.y = (u * triangles[tri].tri0_normal.y) + (v * triangles[tri].tri1_normal.
    y) + (w * triangles[tri].tri2_normal.y);
normal.z = (u * triangles[tri].tri0_normal.z) + (v * triangles[tri].tri1_normal.
    z) + (w * triangles[tri].tri2_normal.z);

if (dot(normal, Vec3f(d.x, d.y, d.z)) >= 0.05f) normal = -normal;

// Construct perpendicular vectors.
Vec3f na = abs(normal);
F32 nm = fmaxf(fmaxf(na.x, na.y), na.z);
Vec3f perp(normal.y, -normal.x, 0.0f); // assume y is largest
if (nm == na.z) perp = Vec3f(0.0f, normal.z, -normal.y);
else if (nm == na.x) perp = Vec3f(-normal.z, 0.0f, normal.x);

perp = normalize(perp);
Vec3f biperp = cross(normal, perp);

// Pick random rotation angle.
U32 hashA = randomSeedAO + pixel1DPos;
U32 hashB = 0x9e3779b9u;
U32 hashC = 0x9e3779b9u;
jenkinsMix(hashA, hashB, hashC);
jenkinsMix(hashA, hashB, hashC);
F32 angle = 2.0f * FW_PI * (F32)hashC * exp2(-32);

// Construct rotated tangent vectors.
Vec3f t0 = perp * cosf(angle) + biperp * sinf(angle);
Vec3f t1 = perp * -sinf(angle) + biperp * cosf(angle);

nAOIntersections = 0;

// Generate each sample.
for (int i = 0; i < numSamplesPerRay; i++)
{
```

```
        // Base-2 Halton sequence for X.
        F32 x = 0.0f;
        F32 xadd = 1.0f;
        unsigned int hc2 = i + 1;
        while (hc2 != 0)
        {
                xadd *= 0.5f;
                if ((hc2 & 1) != 0) x += xadd;
                hc2 >>= 1;
        }

        // Base-3 Halton sequence for Y.
        F32 y = 0.0f;
        F32 yadd = 1.0f;
        int hc3 = i + 1;
        while (hc3 != 0)
        {
                yadd *= 1.0f / 3.0f;
                y += (F32)(hc3 % 3) * yadd;
                hc3 /= 3;
        }

        // Warp to a point on the unit hemisphere.
        F32 angle = 2.0f * FW_PI * y;
        F32 r = sqrtf(x);
        x = r * cosf(angle);
        y = r * sinf(angle);
        float z = sqrtf(1.0f - x * x - y * y);

    // Trace secondary ray
        Vec3f direction = normalize(x * t0 + y * t1 + z * normal);

        float4 originAO, directionAO;

        originAO.x = origin.x;
        originAO.y = origin.y;
        originAO.z = origin.z;
        originAO.w = 0.0f;

        directionAO.x = direction.x;
        directionAO.y = direction.y;
        directionAO.z = direction.z;
        directionAO.w = radiusAO;

        traversalResultAO = rayTraversal(true, originAO, directionAO, nodesA,
            trisA, triIndices);

        if(traversalResultAO.x == -1.0f) nAOIntersections = nAOIntersections +
            1;
}

return nAOIntersections;
}

__device__ inline float4 interpolateBuffer2(int neighbour1, int neighbour2,
    float4 *previousClusterID)
{
```

```
float4 interpolatedValue;

interpolatedValue.x = (previousClusterID[neighbour1].x + previousClusterID[
    neighbour2].x) / 2.0f;
interpolatedValue.y = (previousClusterID[neighbour1].y + previousClusterID[
    neighbour2].y) / 2.0f;
interpolatedValue.z = (previousClusterID[neighbour1].z + previousClusterID[
    neighbour2].z) / 2.0f;
interpolatedValue.w = (previousClusterID[neighbour1].w + previousClusterID[
    neighbour2].w) / 2.0f;

return interpolatedValue;
}

__device__ inline float4 interpolateBuffer4(int neighbour1, int neighbour2, int
    neighbour3, int neighbour4, float4 *previousClusterID)
{
float4 interpolatedValue;

interpolatedValue.x = (previousClusterID[neighbour1].x + previousClusterID[
    neighbour2].x + previousClusterID[neighbour3].x + previousClusterID[
    neighbour4].x) / 4.0f;
interpolatedValue.y = (previousClusterID[neighbour1].y + previousClusterID[
    neighbour2].y + previousClusterID[neighbour3].y + previousClusterID[
    neighbour4].y) / 4.0f;
interpolatedValue.z = (previousClusterID[neighbour1].z + previousClusterID[
    neighbour2].z + previousClusterID[neighbour3].z + previousClusterID[
    neighbour4].z) / 4.0f;
interpolatedValue.w = (previousClusterID[neighbour1].w + previousClusterID[
    neighbour2].w + previousClusterID[neighbour3].w + previousClusterID[
    neighbour4].w) / 4.0f;

return interpolatedValue;
}

extern "C"  texture<float4, 2> t_texture;

__device__ inline U32 interpolateColorBuffer(int clusterID, bool isPrimary, bool
     showShadow, int numAORays, int numShadowRays, int texColor, int
    renderCluster, Vec3f clusterColor, Vec3f diffuseColor, float lightPositionX,
    float lightPositionY, float lightPositionZ, float4 iBufferPixel1, float4
    iBufferPixel2, float4 iBufferPixel3)
{
Vec4f color;

if(clusterID == 0) color = Vec4f(1.0f, 1.0f, 1.0f, 1.0f);
else
{
        if(renderCluster == 1)
        {
                color = Vec4f(clusterColor.x, clusterColor.y, clusterColor.z,
                    1.0);
        }
        else
        {
                Vec3f finalDiffuseColor;
```

```
            Vec3f light = Vec3f(lightPositionX, lightPositionY,
                lightPositionZ) - Vec3f(iBufferPixel1.y, iBufferPixel1.z,
                iBufferPixel1.w);

            if(texColor == 1)
            {
                    float4 tColor = tex2D(t_texture, iBufferPixel2.w, 1.0f -
                        iBufferPixel3.x);
                    finalDiffuseColor = Vec3f(tColor.x, tColor.y, tColor.z);
            }
            else
            {
                    // Compute the shading per vertex
                    light = light.normalized();
                    finalDiffuseColor = diffuseColor * (dot(Vec3f(
                        iBufferPixel2.x, iBufferPixel2.y, iBufferPixel2.z),
                        light) * 0.5f + 0.5f);
            }

            if(isPrimary) color = Vec4f(finalDiffuseColor.x,
                finalDiffuseColor.y, finalDiffuseColor.z, 1.0f);
            else
            {
                            color = Vec4f((F32)iBufferPixel3.y);
                            color *= 1.0f / (F32)numAORays;
                            color.w = 1.0;
                            color *= Vec4f(finalDiffuseColor.x,
                                finalDiffuseColor.y, finalDiffuseColor.z, 1.0
                                f);
                            if(color.x > 1.0) color.x = 1.0f;
                            if(color.y > 1.0) color.y = 1.0f;
                            if(color.z > 1.0) color.z = 1.0f;
                            color.w = 1.0;*/
            }
            if(showShadow)
            {
        Vec4f shadowColor = Vec4f((F32)iBufferPixel3.z);
                    shadowColor *= 1.0f / (F32)numShadowRays;
                    shadowColor = Vec4f(1.0f) - shadowColor;
                    shadowColor += 0.75f;
                    if(shadowColor.x > 1.0) shadowColor.x = 1.0f;
                    if(shadowColor.y > 1.0) shadowColor.y = 1.0f;
                    if(shadowColor.z > 1.0) shadowColor.z = 1.0f;
                    shadowColor.w = 1.0;

        if(dot(Vec3f(iBufferPixel2.x, iBufferPixel2.y, iBufferPixel2.z),
            light) >= 0.0f)
                    {
                            if(iBufferPixel3.z >= 1) color *= shadowColor;
                    }
            }
    }
}

return toABGR(color);
}
```