Interactive Mesostructures

A dissertation presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Doctor of Philosophy

Scott L. Nykl

December 2013

© 2013 Scott L. Nykl. All Rights Reserved.

This dissertation titled
Interactive Mesostructures

by
SCOTT L. NYKL

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

David M. Chelberg
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology

ABSTRACT

NYKL, SCOTT L., Ph.D., December 2013, Electrical Engineering and Computer Science
Interactive Mesostructures (134 pp.)

Director of Dissertation: David M. Chelberg

Humans live in a *3D* reality and experience a *3D* world. As a result, *3D* computer graphics is a natural way to present digital data to humans. Virtual worlds of great complexity are naturally perceived and understood by the human visual system. This mechanism offers an efficient pathway to transfer digital data from a display into human knowledge. Computing *3D* rendering output requires a specific computational complexity for a specific scene. The ever increasing demand to expand a scene's boundaries, add additional details, and enhance a scene's behavior present challenges to both the rendering hardware and rendering algorithms. This work presents a set of novel *image-based* algorithmic rendering approaches designed for massive concurrent execution on modern programmable Graphical Processor Units (GPUs). These approaches make use of constructs known as *billboards* and *mesostructures*, their goal is to render as much *interactive* detail as possible while maintaining real-time framerates.

First, billboards are used to dynamically generate image-based *impostors* of computationally expensive objects within a scene. These impostors are subsequently rendered in lieu of the original geometry thus reducing the scene's overall rendering time. Impostors are then applied to the real-time visualization of tens of millions of *3D* Light Detection and Ranging (LIDAR) data points enabling highly interactive visualizations previously unable to be achieved on commodity hardware.

Second, *mesostructures* are used to dynamically generate a *3D* height field on top of an existing geometrical surface via a *2D* texture known as a *displacement map*. This work presents a technique for interactively deforming and colliding with mesostructures at a per-textel level. This technique integrates well with existing physics engines and is able to

reduce traditional 3D geometrical deformations (vertex-based) to 2D image space operations (pixel-based) that are parallelized on a GPU without CPU-GPU data shuffling. Additionally, surface and material properties may be specified at a per-texel level enabling a mesostructure to possess varying attributes intrinsic to its surface and collision behavior; furthermore, this offers an image-based alternative to traditional decals. The technique provides a simple way to make almost every surface in a virtual world responsive to user actions and events. It requires no preprocessing time and storage requirements of one additional texture or less. The algorithm uses existing inverse displacement map algorithms as well as existing physics engines and can be easily incorporated into new or existing game pipelines.

ACKNOWLEDGMENTS

I would like extend my sincere gratitude to my Advisor, Dr. David Chelberg, for the guidance, support, and freedom he has given me throughout this research and over the course of my Ph.D. I would also like to thank Chad Mourning for his help in co-authoring the STEAMiE Engine, as well as for our countless debates regarding the nature of software design.

I would also like to thank my parents and brother, Thomas, Susan, and Erik for their guidance and encouragement as well their support in all my computational endeavours, both early and later.

Finally, I would like to thank Meagan Verbillion for her care, support, and motivation over the past 3 years. Her companionship has helped make my Ph.D. pursuit very enjoyable, rewarding, and meaningful.

TABLE OF CONTENTS

	Page
Abstract	3
Acknowledgments	5
List of Tables	8
List of Figures	9
1 Introduction	13
1.1 Contributions	13
1.1.1 Billboarding Contributions	15
1.1.2 Mesostructure Contributions	18
2 Image-Based Impostors	24
2.1 Literature Review	29
2.1.1 Planar Impostors	29
2.1.2 Layered Impostors	30
2.1.3 Textured Depth Meshes (TDMs)	31
2.2 Mathematical Transformations for a Graphics Pipeline	32
2.3 Impostor Generation	36
2.4 Light Detection and Ranging (LIDAR) and Billboarding Techniques	42
2.4.1 Previous Work	44
2.4.2 Visualization of LIDAR	45
2.4.3 Results	46
2.4.3.1 Experimental Design	47
2.4.3.2 Analysis	51
2.4.3.3 Interactivity	52
2.4.3.4 Graphics Hardware Scalability	54
3 Interactive Mesostructures	56
3.1 Introduction	56
3.2 Related Work	58
3.3 Interactive Mesostructures	60
3.3.1 Per-Texel Collision Detection	62
3.3.2 Dynamic Deformation, Animation, and Decals	72
3.3.2.1 Texture Repetition	78
3.3.2.2 Simultaneous Mesostructure Collisions	78
3.3.3 Dynamic Hole Creation and Constant Genus	79
3.3.4 Surface Properties	79

3.3.5	Applying to Curved Surfaces	81
3.3.5.1	Approximating Local Curvature via Quadrics	85
3.3.5.2	Approximating Curved Ray-Height field Intersection	87
3.4	Results	88
3.4.1	Implementation using CUDA	89
3.4.2	Implementation with OpenGL 4.3 Compute Shaders	91
3.4.3	Scalable Link Interface (SLI)	94
3.5	Conclusion	95
4	Conclusion	97
4.1	Image-Based Impostor Contributions	97
4.2	Interactive Mesostructure Contributions	98
	References	101
	Appendix: Interactive Mesostructure Implementations	107

LIST OF TABLES

Table	Page
2.1 Results for LRU Metric	52
2.2 Results for Weighted Metric	53
2.3 Average Framerrates for Each Test	54
3.1 Performance matrix listing execution time of the collision and deformation kernels on a NVIDIA GTX 580 GPU and NVIDIA GTX 560M GPU using CUDA 4.1. The corresponding host machines included a Core i7-2600K CPU @ 3.4Ghz with 32 GB of RAM and a Core i7-2670QM @ 2.2 Ghz with 16 GB of RAM, respectively.	89

LIST OF FIGURES

Figure	Page
1.1 Domain of research presented in this document. The red area denotes methods of real-time rendering as discussed in Chapters 2 and 3; Fig.1.3 illustrates the composition of the red area.	14
1.2 Domain of the red area denotes the set of parallelizable algorithms that are able to be executed concurrently on parallel hardware.	14
1.3 Chapters 2 and 3 discuss billboard techniques (blue region) and mesostructure techniques (red region), respectively.	15
1.4 Left side: A 3D model of a KC-135 Stratotanker composed of 20,048 triangles. Right side: A <i>billboard impostor</i> of the KC-135 Stratotanker composed of 2 triangles. Rendering the 3D model requires transforming 60,144 vertices per frame. Rendering the impostor requires transforming six vertices per frame. A vertex transformation reduction of 10,024x per frame.	16
1.5 Rotation about a 3D model and its original 2D image-based impostor. The rotations continue in Fig.1.6.	19
1.6 Continued from Fig.1.5, rotation about the <i>KC – 135 Strato-Tanker</i> . Rotation about a 3D model and its original 2D image-based impostor. The left column shows the 3D geometry, the middle column shows the original billboard, and the right column shows the 2D bounding box around the billboard. Each subsequent row, the camera is rotated 60° counter-clockwise about the <i>KC – 135</i> without updating the billboard. The illusion created by the original billboard (top row of Fig.1.5) quickly becomes apparent as the camera moves. .	20
1.7 Two identical billboards oriented vertically and horizontally. The billboards have been augmented with a mesostructure to provide motion parallax and self occlusion. The texture data used is shown in the right column.	21
2.1 An example go-cart park.	25
2.2 Texture Mapping. The green car and the Hummer truck surface colors are achieved via texture mapping. Below each vehicle is its corresponding 2D texture image. Each vertex has a corresponding texture coordinate that maps into the corresponding texture; the area between each vertex is linearly interpolated across the image.	27
2.3 Taken from Luebke [1]. The top shows the original bunny and 3 geometric impostors of decreasing resolution. The bottom shows a perspective rendering of the same 4 models; notice how the bunnies appear convincing at further distances while requiring fewer vertices.	28
2.4 Taken from Ahn [2]. The left shows a truncated frustum labelled with the variables used in Eq. 2.6. The right shows a top down view of the frustum as well as the corresponding near and far planes.	34

2.5	Left) An approximate scene with no camera motion; Right) a scene with cracking caused by camera motion.	47
2.6	The Haar Wavelet based metric (top 4 lines) and Luminosity metric (bottom 4 lines) plots for each frame for 1, 2, 4, and 64 mesh updates per frame using the LRU policy.	52
2.7	The Haar Wavelet based metric (top 4 lines) and Luminosity metric (bottom 4 lines) plots for each frame for 1, 2, 4, and 64 mesh updates per frame using the Weighted policy.	53
3.1	An interactive mesostructure deformed by spherical impacts. Such an approach provides per-texel interactivity and higher fidelity than traditional decals. The right column of the image shows the mesostructure's color map, normal map, and displacement map from top to bottom, respectively.	61
3.2	The collision kernel performs the ray-height field intersection using $\vec{uv}$ (non-displaced initial texture coordinate), $T\vec{c}_{ts}$ (collider's trajectory in tangent space), and any desired mesostructure rendering technique. The kernel returns uv_{disp} (texture displacement offset), $depth_{ws}$ (world space depth into the mesostructure), and Nc_{ms} (model space contact normal) associated with the intersection point between $T\vec{c}_{ts}$ and the height field. The displaced UV coordinate, $\vec{uv} + uv_{disp}$, is denoted by the pink dot; the $depth$ scalar is the distance from the pink dot to the intersection point along the inverse of macrogeometry's normal vector. The overlapping bounding volumes of the two objects are shown in green. Care must be taken to ensure that uv_{disp} and $depth$ are mapped into world space from tangent/texture space prior to world space use; this scalar multiplication can be performed at the end of the GPU Collision Kernel or any other convenient point prior to calculation of $\vec{C}_{ws}$ (world space collision point), as described in item 6 of the Collision Kernel algorithm in Sec.3.3.1.	62
3.3	The right side shows a mesostructure with several current colliders. Each collider is paired with state information managed by the mesostructure. Each collider-state pair contains the information shown in the left box. This state is updated via callbacks within the physics engine when colliders enter or leave the mesostructure's bounding volume. The red box denotes information computed within the GPU kernels. The green box denotes information required prior to the GPU kernel dispatch.	66
3.4	The GPU Deformation Kernel applies an image operator about the collision point, $\vec{uv} + uv_{disp}$, while accounting for collider-specific attributes and per-texel mesostructure surface properties. The image operator deforms the displacement map accordingly; subsequently, the normal map is updated using the newly deformed displacement map, see Fig.3.4a. The before and after images of the deformation kernel are shown in Fig.3.4b.	70

3.5	Normal and displacement maps before and after a collision, respectively. The displacement map texture is manipulated first and the normal map is regenerated from the updated displacement texture. The deformation process only requires a 2D image-based manipulation to approximate 3D deformations. The contact normal on the left side of this image is consumed by the physics engine to compute corrective force vectors.	73
3.6	Identically color textured mesostructures, each deformed differently. The top surface was impacted by several spheres, some regions were struck multiple times accumulating deeper cavities - a useful advantage over traditional decals. The middle surface was deformed via radial sinusoidal waves emanating from differing points causing interference with one another; the corresponding color, normal, displacement, and specular maps are shown on the right side of the image, top to bottom, respectively. The displacement map shows the origins of each wave. The lower surface has not yet been deformed.	75
3.7	A dynamically deformed box punctured by collisions. Notice colliders are able to enter inside the box after breaking through the depth described by the mesostructure's height map.	76
3.8	A stone wall eroded by collisions. The rendering fragment shader discards fragments where $d_{max} > depth_{ws}$. Holes formed by colliders, as well as areas with small height values in the displacement map, appear invisible and interact as empty space with the physics engine.	80
3.9	Colliders have impacted, deformed, and punctured this curved surface.	82
3.10	A dynamically deformable curved mesostructure. Using Oliveira and Policarpo's rendering technique, one may avoid the drawbacks of planar inverse displacement mapping.	83
3.11	A partial sphere rendered using planar inverse displacement mapping on the left side. The same partial sphere rendered (and horizontally mirrored) using curved quadric-based inverse displacement mapping on the right side. Notice the improved silhouette edges on the right half and the flattened edges on the left half.	84
3.12	Top left: Time-animated waveform of displacement map on a sphere. Top Right: Curvature of a relief mapped saddle point on a Gordian Knot. Bottom Left: Parabolic tops of relief mapped Gordian Knot. Bottom Right: Time-animated waveform of displacement map on a sphere.	86
3.13	Taken from Oliveira [3]. The left figure shows a normal displacement map being traversed by a view ray; this is the exact same procedure as illustrated in Fig.3.2. The right figure shows the same displacement map after being curved by the quadric. Mathematically, however, instead of transforming the height field itself, the view ray is instead curved along its trajectory into the mesostructure.	88

- 3.14 CUDA performance and Compute Shader performance on 3 different graphics cards. Each collider inside the mesostructure's volume dispatches its CUDA / Compute Shader GPU Kernel each frame to account for potential changes in collider trajectory or mesostructure motion. 92
- 3.15 Test configuration showing $N = 25$ colliders simultaneously enter a mesostructure's volume, collide, and bounce back out of the intersection volume. 93

1 INTRODUCTION

Within computer graphics, real-time rendering is the art of creating computational imagery such that human observers perceive a fluid, natural view of the information over time. As the time/space complexity of the rendered information grows beyond that of the rendering hardware, the perception of fluidity diminishes, the time required to compute each frame grows, the user can no longer naturally interact with the data, and the real-time aspect is lost. To preserve real-time interactivity, the time/space complexity of the rendered information must be reduced and/or the computational power of the rendering hardware must be increased. In the first approach, a set of algorithms prune the rendered information, reducing its detail and/or size to preserve real-time interactivity. In the second approach, two hardware improvements are possible: *A*) faster hardware may be employed, or *B*) more hardware may be employed, in parallel. Improvement *A* requires no algorithmic changes, but is limited by the state of the art in hardware. Improvement *B* offers vast computational power, however, its implementation places a significant constraint on the available class of usable algorithms - all applicable algorithms must be parallelizable, see Fig.1.2. Chapters 2 and 3 of this work present a set of novel algorithmic rendering approaches designed for massive concurrent execution on parallel hardware. These approaches make use of constructs known as billboards and mesostructures and their goal is to render as much interactive detail as possible while maintaining real-time framerates, see Fig.1.3.

1.1 Contributions

This section briefly introduces the major areas of real-time rendering to which this work contributes, see Fig.1.1. The first and second areas, *billboards* and *interactive mesostructures*, are closely related topics forming the main focus of this work, see Chapters 2 and 3, respectively. Both billboards and mesostructures are a form of

Figure 1.1: Domain of research presented in this document. The red area denotes methods of real-time rendering as discussed in Chapters 2 and 3; Fig.1.3 illustrates the composition of the red area.

Figure 1.2: Domain of the red area denotes the set of parallelizable algorithms that are able to be executed concurrently on parallel hardware.

image-based rendering which may be used to render a complex virtual world at an interactive framerate such that a human viewer is able to observe a smooth scene that responds quickly and naturally. These techniques do not necessarily render an entire scene in a perfectly accurate manner; instead, they trade a small amount of fidelity for an increase in performance. The goal of this trade off is to ensure that the rendered output is

Figure 1.3: Chapters 2 and 3 discuss billboard techniques (blue region) and mesostructure techniques (red region), respectively.

interactive and looks ‘good enough’ to form a plausible scene when observed by a human. Billboards achieve this by replacing a subset of a scene’s complex geometry with a picture of that complex geometry. This façade has several important limitations, as discussed in Chapter 2. These limitations lead to a new form of image-based rendering called Interactive Mesostructures, which are built on the same underlying image-based principle, yet are able to address many of the billboard’s shortcomings and make image-based rendering a practical solution to modeling almost all surfaces in an interactive virtual world, see Chapter 3.

1.1.1 Billboarding Contributions

Rendering a specific 3D scene to a monitor requires a correspondingly specific amount of computational power. Intuitively, the greater the scene’s complexity, the greater the computational expense. Since computational resources are finite, a point exists where a given scene is too complex to be rendered at interactive frame rates. However, instead of rendering the complex scene in its entirety, one could choose to render a *picture* of an area of a scene and insert that picture in lieu of that area. For example, imagine a scene containing a highly detailed 3D model of a large jet aircraft, see Fig.1.4; assume the

Figure 1.4: Left side: A 3D model of a KC-135 Stratotanker composed of 20,048 triangles. Right side: A *billboard impostor* of the KC-135 Stratotanker composed of 2 triangles. Rendering the 3D model requires transforming 60,144 vertices per frame. Rendering the impostor requires transforming six vertices per frame. A vertex transformation reduction of $10,024x$ per frame.

viewer within the scene is stationary and looking at the aircraft from afar. Each frame, the aircraft must be re-rendered, and the graphics hardware must continually perform the same computations on each vertex to repeatedly generate an identical pixel configuration for the output display. The act of *billboarding* takes the final pixel configuration and overlays that imagery onto a flat, transparent quadrilateral, also known as a *billboard*, see Fig.1.4. Subsequently, the billboard is then rendered in place of the 3D model. Since the quadrilateral contains only six vertices (two triangles), the computational cost decreases from transforming N vertices to transforming six vertices. The relevant pixel configuration is *textured* across the billboard as an image. The billboard is now an *impostor* of the original computationally expensive model. Rendering an object by operating on that object's 3D vertices is called *vertex-based rendering* whereas rendering

an object by operating on that object's 2D texture/pixel data is called *image-based rendering*, Fig.1.4 compares a 3D model against its billboarded counterpart. The left side of the figure shows a 3D model of a KC-135 Stratotanker composed of 20,048 triangles. The right side of the figure shows a *billboarded impostor* of the KC-135 Stratotanker composed of 2 triangles. Rendering the 3D model requires transforming 60,144 vertices per frame whereas rendering the impostor requires transforming six vertices per frame - a vertex transformation reduction of 10,024x per frame. The bottom half of the figure shows the bounding boxes around the 3D model and 2D impostor, from left to right, respectively. Notice that the visible difference between the original model and impostor is nearly imperceivable, yet, in this case, a four magnitude vertex transformation reduction per frame has been achieved. Note however, the texture, or imagery that is placed across the vertices must still be computed and stored, this is elaborated upon in Chapter 2.

Based on these ideas, Chapter 2 describes the mathematical transformations commonly used to transform a set of 3D vertices to a 2D pixel array. A method for dynamically generating arbitrary billboard impostors in real-time is then defined. This technique is generalized to arbitrary data and is maintained via the use of an update policy which ensures the visualization's framerate remains smooth and interactive while attempting to minimize errors caused by out of date impostors. Combining an update policy with image-based impostors enables smooth, real-time interaction with large scale data sets that would otherwise be non-interactive. The presented technique is subsequently applied to the large scale, real-time visualization of Light Detection and Ranging (LIDAR) data. LIDAR is a technology similar to radar, except it uses shorter wavelength lasers instead of RADAR's radio waves. LIDAR is widely used in many fields and is especially helpful for gathering high resolution geological data, such as terrain and elevation information. Chapter 2 makes the following contributions:

- An image-based impostor technique to visualize data sets containing tens of millions of exemplars on commodity hardware in real-time.
- An update policy to dynamically balance this data set's visualization *accuracy* against *interactivity*, quantitatively.
- An image-based error metric to quantify the visual error between the impostors and the reference data set.

1.1.2 Mesostructure Contributions

Image-based impostors are essentially static pictures that must be oriented and positioned precisely to create a convincing illusion of a 3D model. The illusion only holds true when the viewer is at a specific location. If the viewer moves laterally from the impostor, the façade quickly becomes obvious and the impostor must be regenerated to account for the viewer's new location. The more often regeneration is required, the lower the computational savings of using billboards. Several billboard limitations exacerbate this minimal lifespan: *A*) a lack of parallax motion, *B*) a lack of self occlusion, *C*) a lack of animation, and *D*) a lack of interactivity. Figures 1.5 and 1.6 illustrates these problems. The left columns show the 3D geometry fully rendered, the middle columns show the original billboard, and the right columns show the 2D bounding box around the billboard. Each subsequent row, the camera is rotated 60° counter-clockwise about the *KC* – 135 without updating the billboard. The façade is quickly recognizable and the impostor no longer accurately portrays the geometry it is intended to replace.

Billboards lack motion parallax and self occlusion because they are, by definition, planar representations of a 3D object. If a notion of depth was applied upon this planar surface, limitations *A* and *B* could be mitigated, thereby extending the impostor's lifespan. One method to integrate depth information is by applying an additional 2D texture, known as a *displacement map* onto the billboard. When rendering the billboard, one must

Figure 1.5: Rotation about a 3D model and its original 2D image-based impostor. The rotations continue in Fig.1.6.

displace each fragment according to the height values within the displacement map; this type of rendering is called *inverse displacement mapping*. When inverse displacement mapping is applied upon faces residing between geometric vertices, a *mesostructure* is formed. In the case of a billboard, applying a displacement map and rendering via inverse displacement mapping creates a mesostructure as shown in Fig.1.7. Geometrically, the two objects shown are perfectly flat and are each composed of one quadrilateral, the *3D illusion* results from the inverse displacement mapping. The right column of the figure shows the three different 2D textures used to create the mesostructure. Since all necessary

Figure 1.6: Continued from Fig.1.5, rotation about the *KC – 135 Strato-Tanker*. Rotation about a 3D model and its original 2D image-based impostor. The left column shows the 3D geometry, the middle column shows the original billboard, and the right column shows the 2D bounding box around the billboard. Each subsequent row, the camera is rotated 60° counter-clockwise about the *KC – 135* without updating the billboard. The illusion created by the original billboard (top row of Fig.1.5) quickly becomes apparent as the camera moves.

data is 2D imagery, mesostructures are also a form of image-based rendering. Notice the motion parallax, self occlusion, and self shadowing.

Unfortunately, mesostructures still lack interactivity and animation. For example, if a ball were thrown at the mesostructure in Fig.1.7, the geometry is a perfectly flat plane and the sphere would bounce off accordingly, ignoring the illusion of the embedded shapes. Instead, it would be desirable for the colliding ball to *interact* with the mesostructure in a

Figure 1.7: Two identical billboards oriented vertically and horizontally. The billboards have been augmented with a mesostructure to provide motion parallax and self occlusion. The texture data used is shown in the right column.

natural way; for example, if the ball collides with the side of the pyramid, the ball's trajectory should respect the pyramid's boundary and bounce accordingly. Furthermore, if the collision with the mesostructure was also able to *animate and deform* about the

collision point in a natural manner, then a complete replacement for highly detailed geometry becomes possible.

Chapter 3 presents a novel image-based rendering technique called Interactive Mesostructures. This technique adds interactivity, animation, and deformation to image-based surfaces. This technique enables one to replace highly detailed geometry with significantly fewer vertices while still offering interactive, physically plausible surfaces. More specifically, the contributions include:

- A technique for interactively deforming and colliding with mesostructures. This technique leverages existing physics engines, such as NVIDIA PhysX, for initial collision information; it subsequently performs per-textel tangent space collisions, and deforms the affected mesostructure in parallel on the GPU without any server-client texture transfers and minimal CPU involvement.
- Each deformation processed by the GPU is able to account for collider-specific physical attributes as well as per-textel surface properties describing the behavior and/or constraints of the impacted mesostructure; therefore, a mesostructure's visual and behavioral dynamics can be tailored to other objects, animations, or time.
- These deformable mesostructure operations are compatible with many existing inverse displacement mapping rendering techniques including relief mapping and parallax occlusion mapping, among others, lending themselves to easy integration into existing game pipelines and applications that use GPU-based inverse displacement mapping techniques.
- Replaces 3D vertex-based deformations with 2D image-based operations performed solely on the GPU, in parallel, and free from CPU-GPU data shuffling.

- Provides a per-textel alternative to traditional decals. This alternative offers enhanced visual cues, requires no additional geometry, and persists for the life-time of the application.

2 IMAGE-BASED IMPOSTORS

Prior to introducing billboards in depth, the general mechanism by which modern computer rendering occurs is explained. Rendering commonly draws a given scene to some output device, such as a monitor/display. The rendered scene is commonly a *3D* volume containing a set of *3D* objects. For example, a scene depicting a *go-cart park* may contain a race track, ramps, obstacles, walls, and race cars atop a grassy field lit by a sunny blue sky, see Fig.2.1. Each object is modeled mathematically via a set of *3D* vertices defined within a continuous coordinate system (commonly cartesian). Each vertex may also have a corresponding color or texture coordinate; such a correspondence enables a *texture* to be wrapped around the *3D* model enhancing the geometric surface with visual detail, see Fig.2.2. Given a viewing location and direction, the scene is rendered through the viewport of a pin-hole camera (*virtual camera*); each object's vertices are transformed through a series of mathematical equations that ultimately determine the color of each pixel on the output device. The resulting pixel configuration attempts to faithfully draw a *2D* representation of the *3D* scene at the given viewing location and direction. This method of rendering is a form of *vertex-based rendering*; that is, rendering is accomplished via operating on each object's vertices.

A major drawback of vertex-based rendering involves a model's vertex complexity - the more vertices that are required to define a model, the longer it takes to render. Highly detailed models may look realistic but they have a higher space/time rendering complexity. Imagine that the pink go-cart in Fig.2.1 is a highly detailed model comprised of millions of vertices and takes n msec to render. Now imagine that the pink go-cart drives to the far wall of the park while the viewing location remains the same. At the far side of the park, the go-cart still takes n msec to render; however, the pink go-cart may only occupy several pixels out of the 2 million+ pixels on a common 1920x1080 pixel display. This means that during each frame, n msec are still required to draw the go-cart,

Figure 2.1: An example go-cart park.

yet only a minuscule fraction of the final pixels are changed as a result of that work. In general, as the distance from the viewer increases, an object's relative size, or number of pixels consumed on the display, decreases¹ while the object's render time remains constant. If the pink go-cart is far from the viewer and will ultimately influence a small number of pixels on the screen, then rendering *only a subset* of the go-cart's vertices may yield *the same result, or a very similar result*, as rendering all of the go-cart's vertices, but at a reduced rendering time. Rendering fine-grained details, such as seat-belt buckles, dashboard controls, hood ornaments, or the sparkplugs inside the engine block will have a negligible affect on the resulting frame; concordantly, rendering these details needlessly

¹ Assuming a perspective projection instead of an orthographic projection.

wastes computation time. Clark's seminal 1976 work [4] on visible surface algorithms described this fundamental problem and introduced an algorithmic framework to manage scene complexity given finite computational resources. This foundational framework included hierarchical bounding volumes, logarithmic clipping acceleration, variable surface detail, and frame-to-frame coherency. Today, each of these methods has become a fundamental tool in modern rendering.

The process of determining how and when to reduce an object's rendering time is known as *level of detail (LOD)* and may utilize any combination of Clark's approaches. From a geometric perspective, this can be done by manipulating the object's rendered vertex set [1, 5–7]. Commonly, the rendered vertex set is manually reduced by an artist to several predefined levels of decreasing resolution, see Fig.2.3. Each level is a *geometric impostor*, that is designed to serve as a substitute for the object's complete vertex set at a particular range from the viewer. As the object moves further from the viewer an algorithm decides to render a specific geometric impostor instead of the original object. The algorithm must continually try to minimize rendering time while preserving the fidelity of the final output. In Fig.2.3, the original 3D model and three geometric impostors are shown at the top of the image. Each of these models is then rendered at increasing distances from the camera as shown at the bottom of the image. Notice how as the distance from the camera increases, the required fidelity to convincingly render the bunny decreases. Geometric impostors, such as these, are an example of *vertex-based level of detail*.

Regardless of how an algorithm reduces a 3D model's vertex set, the *ultimate goal* is to determine what *2D pixel configuration* to display to the output. The notion of *operating on vertices* to manipulate the 2D visual display is in some ways, an indirect means to an end. Instead of beginning with vertices and ending with a 2D display, one may choose to *begin* with the 2D pixel configuration and work backwards. Using this approach, the

Figure 2.2: Texture Mapping. The green car and the Hummer truck surface colors are achieved via texture mapping. Below each vehicle is its corresponding 2D texture image. Each vertex has a corresponding texture coordinate that maps into the corresponding texture; the area between each vertex is linearly interpolated across the image.

number of pixels an object will occupy on the 2D output determines how much time to spend rendering that object. In other words, the time spent rendering an object is made *proportional to* the number of pixels that object will occupy within the *final* image.

Operating on an object's screen-space area instead of its geometric vertex set is a form of

(a)

(b)

Figure 2.3: Taken from Luebke [1]. The top shows the original bunny and 3 geometric impostors of decreasing resolution. The bottom shows a perspective rendering of the same 4 models; notice how the bunnies appear convincing at further distances while requiring fewer vertices.

image-based rendering. In some sense, image-based rendering is an intuitive notion - the more area an object consumes within the final output, the more time ought to be spent rendering that object. Similarly, the less area an object covers in the final output, the less

time ought to be spent rendering that object. With respect to the aforementioned example of the pink go-cart far away from the viewer, if it is known that the pink go-cart will occupy a small area, a more intelligent rendering decision can be made. By taking this notion of image-based rendering to an extreme, one can reduce any arbitrary geometric vertex set to a constant size of six vertices (two triangles), see Fig.1.4. The largest advantage of an image-based impostor (an image-based LOD approach) is that its render time is a function of the number of pixels (fill rate limited) it occupies on the display as opposed to the number of vertices composing the object; this is the opposite of geometric LOD approaches (vertex-rate limited).

2.1 Literature Review

2.1.1 Planar Impostors

Dynamic billboard generation, as described above, was first introduced in 1995 by Schaufler and is the canonical example for image-based impostor generation [8]. This relatively simple, yet effective idea generates a textured quadrilateral that displays the perspective projection of the corresponding geometry across its surface. Areas of the quadrilateral not covered by the original geometry are textured as transparent. This textured quadrilateral is rendered in place of the original geometry. In other words, a *picture* of the original geometry is taken; subsequently, this picture is positioned at and takes the place of the original geometry. As the viewer changes position, this impostor becomes incorrect and artifacts emerge. The artifacts are the same as those that would arise when trying to perform this substitution within the real world. Impostor update policies, such as Funkhouser's 1993 adaptive display algorithm [9], have been applied to mitigate these errors and find the optimal impostor update methods given a scene [10]. Typically a user modifiable error threshold, ϵ , is used to determine how much error may accumulate within the output pixel configuration before forcing a regeneration of the

impostor. A reasonable lower bound may govern that only sub-pixel errors accumulate. In other cases, ϵ may need to be increased to reduce the impostor update frequency.

Other research, including Aliaga's work, focused on smoothly morphing geometry into an impostor without incurring popping artifacts [11, 12]. He also extended this concept to use image-based impostors for generating guaranteed framerates of static scenes [13]. Shade's 1996 work with Microsoft Research introduced a BSP-tree based hierarchical image cache to accelerate walk throughs of complex worlds using image-based impostors [14].

One commonality among the above approaches is that in each case, the impostors were used in the far field; that is, impostors were generated to replace distant objects far away from the viewer. In some cases, such as Maciel's and Shade's work, the image-based impostors initially generated did not exactly match the replaced object; while this is not necessarily an issue for far-field uses, these imprecisions are exaggerated when impostors are applied to the near field. Furthermore, generating an impostor of a partially off screen object is a recurring issue neglected by previous works. Shade discussed this issue and recommended rendering the object's vertex set (vertex-based rendering) in lieu of generating an impostor. The difficulty in generating partially off screen impostors is explained in Sec.2.3.

2.1.2 Layered Impostors

A single planar impostor does not support parallax motion. By generating several planar impostors for a single object and spacing these out along the view direction, parallax motion can be approximated, prolonging the lifetime beyond that of a single planar impostor. This method also helps improve 'cracking' artifacts [15–17], and has been successfully used with vegetation such as trees [18]. Jeschke extended Schaulfer's work to improve upon cracking artifacts as well as parallax motion [19]. This method's

cost grows linearly with the number of layered impostors that are generated, whereas a planar impostor uses only one. Thus, memory and generation time are increased to provide a longer lifetime for the impostor.

2.1.3 Textured Depth Meshes (TDMs)

The goal of a textured depth mesh (TDM) is to create an impostor which produces the correct parallax motion for multiple objects within the background of a given scene. A TDM essentially renders a set of objects for which a single TDM will be generated. The Z buffer provides the necessary depth information to generate a mesh that is textured according to the actual geometry it is replacing. TDMs were first introduced by [20, 21].

TDMs require only slightly more memory than a planar impostor given that the mesh is not overly dense; however, generating a mesh of the proper density is a non-trivial issue; the majority of the TDM work has been on generating a 'properly dense' mesh which has enough detail to provide sufficient realism without making the mesh too dense and therefore requiring excessive memory. TDMs typically provide reasonable parallax motion but are susceptible to disocclusion artifacts (also known as rubber sheeting) since texture information on faces nearly parallel to the view direction is unavailable during generation. When the viewpoint changes such that those faces become visible, the missing texture information appears as an improperly colored extrusion.

Darsa et al. [20, 22, 23] proposed using the dual of a Voronoi diagram (Delauney triangulation) to aid in TDM generation; the input vertices are obtained via sparse ray casting onto the depth array and recording the intersections. This solution also handles proper texturing of the mesh. Further contributions to TDMs have focused on improving the viewing angle, removing disocclusion artifacts, and an off-line preprocessing scheme combined with dynamic use of this preprocessed data during runtime to reduce load times [5, 21, 24–27].

2.2 Mathematical Transformations for a Graphics Pipeline

This section describes the foundational mathematics used to transform a vertex from its original pose (position/orientation) to its final location on the screen given a specific viewing location and direction as well as the screen resolution. These equations and corresponding matrices are used throughout the sections discussing impostor generation and also describe the fundamental linear transformations performed by a canonical graphics pipeline such as OpenGL.

Let v_{obj} be a 3 dimensional column-major vertex extended to a homogeneous coordinate $[x, y, z, w]^T$ in object space. The w coordinate is set to 1 for a vertex (point) and set to 0 for a vector. Assume v_{obj} is placed within a local reference frame relative to its own origin. In this configuration, v_{obj} only has position and orientation relative to its local origin and has no position or orientation within the global or world coordinate frame. To specify a transform from this local object space to world space, a 4x4 column major homogeneous transformation matrix M is defined.

$$M_{trans} = \begin{pmatrix} 1 & 0 & 0 & tx_w \\ 0 & 1 & 0 & ty_w \\ 0 & 0 & 1 & tz_w \\ 0 & 0 & 0 & 1 \end{pmatrix}, M_{rot} = \begin{pmatrix} x_x & y_x & z_x & 0 \\ x_y & y_y & z_y & 0 \\ x_z & y_z & z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.1)$$

$$M = M_{trans} \times M_{rot} \quad (\text{Column major} \rightarrow \text{post multiplication}) \quad (2.2)$$

As shown in Eq. 2.1, the upper left 3x3 submatrix of M_{rot} forms an orthonormal matrix, defining a local coordinate system used for transforming the orientation of v_{obj} from object space to world space via post multiplication with M_{rot} . This transformation matrix is often referred to as a direction cosine matrix (DCM). Matrix M_{trans} specifies the translation applied to v_{obj} , which describes the vertex's position in world space. The

canonical display matrix, M , is the accumulation of M_{rot} post multiplied with M_{trans} .

Therefore, world space vertex v_{world} is computed via post multiplication of v_{obj} and M :

$$\begin{pmatrix} v_{world} \cdot \mathcal{X} \\ v_{world} \cdot \mathcal{Y} \\ v_{world} \cdot \mathcal{Z} \\ v_{world} \cdot \mathcal{W} \end{pmatrix} = M \times \begin{pmatrix} v_{obj} \cdot \mathcal{X} \\ v_{obj} \cdot \mathcal{Y} \\ v_{obj} \cdot \mathcal{Z} \\ v_{obj} \cdot \mathcal{W} \end{pmatrix} \quad (2.3)$$

v_{world} is now in world space. World space provides a consistent reference frame in which all 3D objects are embedded. This reference frame, in general, is where the 3D description of the virtual world takes place. However, the world space reference frame must always be *viewed* from some position looking in some direction. In other words, the vertex, v_{world} , needs to be transformed relative to a desired viewing position/orientation (also known as the *eyeball*). Such a transformation takes v_{world} from world space to *eye space*, v_{eye} (also known as view space). To convert into eye space, v_{world} is transformed by the 4x4 *view* matrix, V , defined in Eq. 2.5. The view matrix transforms the scene making it relative to the eyeball's position/orientation. In OpenGL [28], for example, the eyeball is placed at $[0, 0, 0]$ looking down the $-Z$ axis with the $+Y$ axis defining the up vector. All world space vertices, including v_{world} , are transformed relative to the eyeball's pose, via post multiplication with V . Afterwards, vertex v is in *eye space*, v_{eye} , as defined in Eq. 2.4.

$$\begin{pmatrix} v_{eye} \cdot \mathcal{X} \\ v_{eye} \cdot \mathcal{Y} \\ v_{eye} \cdot \mathcal{Z} \\ v_{eye} \cdot \mathcal{W} \end{pmatrix} = V \times \begin{pmatrix} v_{world} \cdot \mathcal{X} \\ v_{world} \cdot \mathcal{Y} \\ v_{world} \cdot \mathcal{Z} \\ v_{world} \cdot \mathcal{W} \end{pmatrix} \quad (2.4)$$

$$V = \begin{pmatrix} -cam_{relY} \cdot x & -cam_{relY} \cdot y & -cam_{relY} \cdot z & 0 \\ cam_{relZ} \cdot x & cam_{relZ} \cdot y & cam_{relZ} \cdot z & 0 \\ -cam_{relX} \cdot x & -cam_{relX} \cdot y & -cam_{relX} \cdot z & 0 \\ pos \cdot cam_{relY} & -pos \cdot cam_{relZ} & pos \cdot cam_{relX} & 1 \end{pmatrix} \quad (2.5)$$

Now that the eyeball's position and orientation have been defined and v has been transformed into eye space, one must define other attributes about the eyeball. For example, one must determine the eyeball's field of view (horizontal and vertical), the maximum distance the eyeball can see, the minimum distance the eyeball can see, and how objects far away appear relative to nearby objects. All of these attributes can be defined within a 4×4 matrix, P , called the *projection* matrix, defined in Eq. 2.6. The projection matrix defines a clipped frustum as shown in Fig. 2.4. Within Eq. 2.6, r, l, t, b are the right, left, top, and bottom scalars defining the dimensions of the *near plane*, respectively. Variables n, f are the scalar distances from the eyeball to the near and far planes, respectively. This clipped frustum defines the remaining characteristics of the eyeball².

Figure 2.4: Taken from Ahn [2]. The left shows a truncated frustum labelled with the variables used in Eq. 2.6. The right shows a top down view of the frustum as well as the corresponding near and far planes.

² Projection matrix, P , assumes a perspective projection which is most commonly used for rendering 3D virtual worlds such as games. Alternatively, one can define an orthographic projection within a similar 4×4 matrix. The main difference between the two approaches is that the first offers a perspective-correct depth; that is, objects in the distance appear smaller than objects in the fore front, similar to that of the human visual system. The second approach does not account for this and therefore less depth information can be discerned. The first approach is more relevant to this work and is concordantly elaborated upon.

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (2.6)$$

The vertex in eye space, v_{eye} , is then transformed via post multiplication with the projection matrix, P , yielding a coordinate in clip space, v_{clip} , as defined in Eq. 2.7. Once in clip space, v_{clip} can easily be checked for inclusion within the frustum. If v_{clip} does not lie within the frustum, it can be discarded since v_{clip} is out of the eyeball's viewing frustum. This inclusion test simply involves ensuring that the x, y, z components of v_{clip} reside between $[\pm v_{clip}.w]$; if not, v_{clip} can be discarded.³

$$\begin{pmatrix} v_{clip}.x \\ v_{clip}.y \\ v_{clip}.z \\ v_{clip}.w \end{pmatrix} = P \times \begin{pmatrix} v_{eye}.x \\ v_{eye}.y \\ v_{eye}.z \\ v_{eye}.w \end{pmatrix} \quad (2.7)$$

Clip coordinate v_{clip} has now been projected onto the 2D projection plane; note however, that $v_{clip}.z$ still denotes its depth between the near and far planes. This coordinate still needs to be normalized relative to the near plane's dimensions so that it can finally be mapped to a location on the rendering screen/window. This normalization is called the *perspective divide* and is simply scalar division as defined in Eq. 2.8. The resulting *normalized device coordinate*, or NDC space coordinate, v_{ndc} , is a 3D vector $[x, y, z]^T$ where each component resides within $[\pm 1]$.

³ Care must be taken to ensure that polygons containing both included and discarded vertices are properly *split* along the edges of the frustum to ensure the polygon is properly clipped by the frustum.

$$\begin{pmatrix} v_{ndc}.x \\ v_{ndc}.y \\ v_{ndc}.z \end{pmatrix} = \begin{pmatrix} v_{clip}.x/v_{clip}.w \\ v_{clip}.y/v_{clip}.w \\ v_{clip}.z/v_{clip}.w \end{pmatrix} \quad (2.8)$$

At this point v_{ndc} is now able to be mapped into *screen space* and placed on the rendering screen via the final viewport transformation. v_{ndc} is scaled by the viewport size and translated by the viewport position yielding a *screen space* coordinate as defined in Eq. 2.9. The resultant screen space coordinate defines the (x, y) location on the screen and the screen space z value is the depth *into* the screen at which the coordinate exists.

$$\begin{pmatrix} v_s.x \\ v_s.y \\ v_s.z \end{pmatrix} = \begin{pmatrix} \frac{w}{2}(v_{ndc}.x + 1) + x \\ \frac{h}{2}(v_{ndc}.y + 1) + y \\ \frac{f-n}{2}v_{ndc}.z + \frac{f+n}{2} \end{pmatrix} \quad (2.9)$$

In Eq. 2.9, x and y define the position of the viewport's lower left corner, and n and f map the depth values from NDC space to screen space; n specifies the near plane while f specifies the far plane, these values are commonly 0 and 1, respectively. $v_s.x$ and $v_s.y$ correspond to the current pixel location of v_{world} and $v_s.z$ maps to the current depth into the screen from the near plane to v_{world} . In summary, a vertex v_{obj} is converted to a screen space (pixel location) via the following equation:

$$\begin{pmatrix} v_s.x \\ v_s.y \\ v_s.z \end{pmatrix} = NDC \leftarrow \begin{pmatrix} \frac{1}{w} & 0 & 0 & 0 \\ 0 & \frac{1}{w} & 0 & 0 \\ 0 & 0 & \frac{1}{w} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times P \times V \times M_{trans} \times M_{rot} \times \begin{pmatrix} v_{obj}.x \\ v_{obj}.y \\ v_{obj}.z \\ v_{obj}.w \end{pmatrix} \quad (2.10)$$

2.3 Impostor Generation

Impostor generation involves creating a picture of a complex object, obj , and rendering that picture in lieu of obj , thus reducing rendering time. More specifically obj

is painted into a texture, tex , based on the relative position and orientation between the camera and obj at time t_0 . Once tex has been created, a quadrilateral, q , is then positioned at obj 's location and tex is mapped across q . At this point, obj need not be rendered; in its place, the four vertices of q are rendered displaying tex to the viewer. Assuming q is positioned and sized properly and tex is properly sized / filtered, a viewer cannot distinguish between obj and q . Figure 1.4 illustrates an original object and its corresponding impostor.

Determining q 's optimal position, orientation, and size as well as the optimal texture dimensions that provides a pixel-accurate recreation are now described using the equations and matrices described in Sec.2.2.

To find the optimal texture dimensions, such that one screen space pixel maps to exactly one texel of tex , we take the 8 world space vertices comprising obj 's current oriented bounding box, $objBndBox_{ws}$, and transform these 8 vertices into screen space, using Eq. 2.10. Once in screen space, a tight axis aligned 2D bounding box, $objBndBox_{ss}$, is generated using the maximal and minimal x and y extrema of these 8 vertices; the z components, representing depth into screen space are ignored; thus $objBndBox_{ss}$ contains 4 vertices. The width and height of this axis aligned 2D bounding box correspond to the maximum number of pixels (± 0.5 pixels) that obj will occupy, in the X and Y screen space directions, respectively. Therefore, q and tex must match this width and height as well. In other words, each texel in tex will map to exactly one screen space pixel when rendering obj into tex , assuming tex is rendered parallel to the frustum's near plane, see Fig.2.4. This guaranteed one-to-one pixel-to-texel mapping ensures pixel accurate reconstruction and is why one may consider this an optimal texture size. Since the dimensions of tex - w and h , in pixels, have been determined, we can now render obj into tex thus generating the desired texture for q .

To render obj into tex , one may reapply Eq.2.10 with the output window set to tex . However, in this case, the viewport dimensions should be restricted to the width and height of tex ($w \times h$ pixels). In other words, the render target (output window) is set to the dimensions of tex , see Eq.2.9. Furthermore, the viewing frustum defined by projection matrix P , must be restricted to view the desired area about obj , otherwise, the entire scene would be rendered into tex instead of only obj . This can be accomplished in two steps. First, after setting the viewport size to $w \times h$, recompute $objBndBox_{ss}$ to accommodate the new viewport values. Secondly, using the center point and lengths from the updated $objBndBox_{ss}$, modify P 's r, l, t, b values accordingly, see Eq.2.6. Finally, obj can be rendered into tex using Eq.2.10 with the updated P and viewport values. These steps set the viewing frustum and viewport such that obj is properly drawn into tex while preserving position, orientation, and size of obj w.r.t the viewer.

Now that tex is of the ideal size and has the proper contents, all that remains is to find the proper size, orientation, and position for q . We know the orientation of q needs to be parallel to the view plane to ensure each texel of tex maps to one pixel in screen space; therefore, q needs to be oriented opposite to that of the camera's pose. This is achieved by taking the camera's current normalized DCM (see Sec.2.2 and Eq. 2.1) and negating the X and Y components. This preserves the handedness and ensures texture mapping tex from $[0..1]$ is not flipped or upside down when applied to q in both the u and v directions; this technique also avoids recomputing an orthonormal basis set for q 's DCM.

As for finding the proper size and world space position of q such that one texel will map to exactly one pixel (assuming q is parallel to the view plane), we take the 8 vertices in $objBndBox_{ws}$ as well as the oriented bounding box's world space center point, $bboxCenter_{ws}$ and convert these 9 vertices into screen space using the original viewport and projection matrix (prior to generating the contents of tex). Once again we update $objBndBox_{ss}$; however, this time, we overwrite the z coordinate (depth from the near

plane in screen space) of each of the 4 vertices to $bboxCenter_{ss}.z$, and we also record the perspective divide value, w , of $bboxCenter_{clip}$. Overwriting the z values enforces that the screen space axis aligned bounding box is parallel to the viewing plane and it also defines the depth of q in screen space; this depth in screen space is used to compute the distance from the camera that q will be positioned in world space. Using these 4 newly computed vertices we compute $objBndBox_{ss}$'s center point.

At this point we have 5 screen space vertices, 4 defining the edges and 1 defining the center of the screen space bounding box; all 5 have a z value of $bboxCenter_{ss}.z$. Prior to further transformation, one may observe it is only necessary to represent the bounding box using two corner vertices, in this case we arbitrarily chose the upper left and lower right vertices (since we already have the center point, we really only need one corner vertex). We can now transform these 3 vertices (upper left, lower right, and center) from screen space to world space by performing the opposite of Eqs.2.4,2.7,2.8,2.9; we use the perspective scalar w , from $bboxCenter_{clip}$ mentioned above for the inverse of the perspective divide (perspective multiply). The following equation defines this transformation; note that NDC^{-1} rewrites Eq. 2.9 solving for v_{ndc} given v_s :

$$\begin{pmatrix} v_{world}.x \\ v_{world}.y \\ v_{world}.z \end{pmatrix} = (PV)^{-1} \times \begin{pmatrix} w & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times NDC^{-1} \times \begin{pmatrix} v_s.x \\ v_s.y \\ v_s.z \\ v_s.w \end{pmatrix} \quad (2.11)$$

Eq. 2.11 requires inverting the $P \times V$ matrix; fortunately, the upper left 3x3 submatrix of the view matrix V is a DCM and the first 3 rows of the 4th column represent eye space translation. Therefore, transposing V 's DCM inverts the upper 3x3 (orientation information) portion. Subsequently, multiplying V 's translation values, (x, y, z) , through the transposed 3x3 DCM and finally negating the resultant vector is equivalent to inverting

V. Furthermore, the perspective projection matrix P defines a viewing frustum as a function of the height and width of the near plane and its displacement from the camera/origin (and correspondingly the far plane's dimensions and displacement); in this case, an analytical solution to P^{-1} exists. The canonical perspective projection matrix is given by Eq. 2.6 and the inverse is given by Eq. 2.12, where r, l, t, b, n, f represent the right, left, top, and bottom eye space dimensions of the near plane, respectively. n and f represent the eye space distance from the origin of the near and far planes, respectively [28]; this inverse holds true as long as $r \neq l, t \neq b$, and $n \neq f$:

$$P^{-1} = \begin{pmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{pmatrix} \quad (2.12)$$

The result of transforming the 3 screen space vertices by Eq. 2.11 yields 3 world space vertices. The world space vertex originating from the screen space bounding box's center *is the world space position of q 's center*; note that this center point is different than the world space center of the oriented bounding box and achieves more precise results. This increased precision results from mapping the screen space center back into world space using the depth (z, w) values obtained from projecting $bboxCenter_{ws}$ into screen space; essentially, we slightly offset q 's world space position from obj 's world space position to compensate for the fact that projecting the screen space axis aligned bounding box back into world space will occupy an area greater than or equal to the area consumed by the oriented bounding box projected onto a plane parallel to the near plane at a distance of $objBndBox_{eye}.z$. In other words, the impostor will have an area greater than or equal to obj ; in the case, where the area is greater, q must be slightly offset in order to achieve the illusion that it is positioned exactly where obj was positioned. This is because q is parallel

to the view plane, and *obj* is not necessarily parallel to the view plane as it is composed of 3D geometry and not a single quad.

The remaining 2 world space vertices are subtracted to form a vector $lxlyWS = lowerright - upperleft$. This vector stores the optimal dimensions of q ; however, it is expressed in world space concatenated with the camera's DCM as per Eq. 2.11. This implies that $lxlyWS$ does not lie in the current view plane which means the x, y, z components may all have non-zero values. We wish to counter rotate $lxlyWS$ such that it lies in the current view plane causing $lxlyWS.x$ to drop to zero; in this right handed model, Z is up, X is forward, and Y is left, so the YZ plane is the current view plane. This can be achieved by transforming $lxlyWS$ through the inverse of the camera's current DCM; since the DCM is an orthonormal basis set of rank 3, its transpose is equivalent to its inverse and the 4th row and column may be set to identity. The transformed $lxlyWS$ now has an x component of 0, a y component whose absolute value specifies q 's width in world space, and a z component whose absolute value specifies q 's height in world space.

We have successfully generated an impostor; we computed the optimal texture size of tex , the contents inside of tex , q 's optimal position, and q 's optimal width and height without having to perform any costly matrix inversions. Furthermore, the computations described in Eqs. 2.11 and 2.10 may be performed in parallel; enhancing the concurrency of impostor generation. Unfortunately, although the computations are parallelizable, the essential task of rendering an object's geometry into its corresponding texture (resizing and generating the contents of its texture) using standard render-to-texture constructs such as Frame Buffer Objects (FBOs) must be performed sequentially due to the pipelined architecture of today's typical graphics cards. However, if one were to use modern programmable graphics kernels, made available by an API such as NVIDIA's CUDA [29]; each kernel could re-implement the graphics pipeline and output to a given impostor's texture. This would ensure independence between each impostor on both the CPU and

GPU and therefore imply available concurrency for the complete generation of each impostor. Of course, all impostor attributes (texture, position, size) must be completed prior to generating the next frame to avoid artifacts.

2.4 Light Detection and Ranging (LIDAR) and Billboarding Techniques

In the preceding section, a mechanism was presented that enables one to generate an image-based impostor from a geometric object. More generally, this mechanism can be applied to any data set that can be partitioned into discrete volumes. Each discrete volume contains a collection of vertices that define a local geometry called a mesh. For each mesh, a corresponding image-based impostor can be generated and rendered in lieu of the original data. This general mechanism offers highly interactive frame rates at the cost of some fidelity. To find an optimal balance between interactivity and accuracy, one may consider the strengths of the human visual system. In general, the human visual system excels at filtering out small errors, filling in gaps of missing data, and tracking smoothly moving objects. Fortunately, the properties intrinsic to the errors resulting from out of date impostors are small perspective errors and gaps of missing data; the benefit gained is smooth, interactive motion. In this way, sacrificing fidelity for increased interactivity plays to the strengths of the human visual system. In contrast, a highly accurate, yet non-interactive, jerky visualization is more difficult for a viewer to perceive and understand. One important consideration, however, is determining the impostor update frequency. If the update frequency is too high, each mesh's impostor is continually regenerated which increases computational complexity; this results in reduced interactivity, but yields a smaller average error per impostor. Conversely, if the frequency is too low, each mesh's impostor is regenerated infrequently; this results in greater interactivity, but yields a larger average error per impostor. The remainder of this chapter

applies this general image-based impostor mechanism to a large data set resulting in a real-time visualization of data that would not be possible using traditional methods.

LIDAR (Light Detection and Ranging) [30] is a remote sensing technology that is growing in popularity across disciplines. Modern LIDAR systems can produce substantial amounts of data in a very brief amount of time. One of the greatest challenges facing researchers is processing and visualizing all of this information in real time. Ideally, a scientific visualization of a set of LIDAR data should provide an *accurate* view of all the available information; however, sometimes it is beneficial to exchange a small portion of that accuracy for the increased usability and flexibility of a *real-time, interactive display*. This section presents:

- An image-based impostor technique to visualize data sets containing tens of millions of exemplars on commodity hardware in real-time.
- An update policy to dynamically balance this data set’s visualization *accuracy* against *interactivity*, quantitatively.
- An image-based error metric to quantify the visual error between the impostors and the reference data set.

LIDAR has become useful across many domains: ecology, to measure river corridor topography [31]; renewable energy, to measure wind speeds for optimal wind turbine placement [32]; athletics, to measure yacht racing conditions at the 2008 Olympic Games [33]; agriculture, to calculate the sun-facing tilt of land [34]; robotics, for environmental perception and object recognition [35], and most directly related to this research, avionics [30]. This increasing use of LIDAR introduces substantial research challenges, in particular, processing the vast amount of data produced by LIDAR and effectively visualizing such large amounts of data in real time. A 2008 LIDAR scan of approximately 3 miles² around Ohio University’s Snyder Field consisted of more than 12.7GB of 3D

point cloud data; the corresponding high resolution aerial photography of the same area was only 891MB, roughly 14.6 times smaller than the LIDAR, as an example.

2.4.1 Previous Work

Impostors, as discussed in section 2.1, have been used for many years as a computationally inexpensive replacement for high-fidelity models [8, 14, 36]. Recall that impostors are textured quadrilaterals that provide the user with a similar visual representation of the polygonal models, but at a reduced rendering time. While impostors have been in common usage for over a decade, active research is still being performed, particularly on aspects of parallelization [37] and augmenting impostors with displacement maps [38–40] (see Chapter 3).

Because this section deals primarily with ways to quickly approximate a scene of a large LIDAR point cloud while attempting to maintain an accurate yet interactive frame rate, metrics are needed that can quantify the error between the impostors and the original LIDAR point cloud. To quantify these differences, two image-based metrics are utilized: a metric based on luminosity [41] and a metric based on the Haar wavelet [42]. The Haar wavelet has previously been used for many image compression and processing techniques [43]; in depth descriptions of these metrics are found in section 2.4.3.

An alternative method for visualizing large scale point cloud data is to randomly subsample the cloud until its size is reduced to a point where it becomes interactive on the given hardware. The largest drawback with this approach is the accidental removal of salient and/or extreme points; without these points important features can be lost decreasing the usefulness of the LIDAR visualization. In contrast to this approach, the aforementioned impostor-based approach does not remove points, thereby avoiding the problems of lost salient features.

2.4.2 Visualization of LIDAR

LIDAR data, in its raw form, is a set of 3D points. Only a few square miles may contain billions of points using a moderate resolution LIDAR scanner. With higher resolution scanners and larger areas, the data collected is likely to continue to increase in size. Fortunately, a camera within a virtual world can only observe the volume defined by the view frustum at any time, see Fig.2.4. Using this constraint, one may interactively visualize a subset of the entire point cloud at any given time. Using a virtual camera with a 65° vertical field of view, a far clipping plane of $10km$, and an aspect ratio of 4:3 (width to height), it was determined that interactive visualization of only tens of millions of LIDAR points was sufficient to render the LIDAR volume contained within the camera's viewing volume. To visualize tens of millions of LIDAR points at highly interactive frame rates ($\sim 20 - 50$ fps), the presented algorithm utilizes and combines several well known approaches in a new way, namely, frustum culling using octree acceleration, point set subdivision into rectilinear grids, dynamic impostor generation, and an update policy controlling the fidelity / frequency of impostor generation.

Prior to rendering, the algorithm splits the LIDAR point cloud into an arbitrary number of rectilinear meshes of roughly equal size. The mesh size is user defined and is an experimental value discussed in Sec. 2.4.3.1. This subdivision enables the algorithm to create an impostor for each mesh versus trying to create one large impostor for the entire point cloud. Furthermore, each mesh is inserted into an octree which is used to accelerate frustum culling on a per frame basis. At each frame, the set of currently visible meshes is generated and given as input into the impostor *update policy*. The impostor update policy is responsible for minimizing potential visual errors given the time constraint necessary to yield a highly interactive frame rate. Evaluated impostor update policies are discussed in section 2.4.3.2.

2.4.3 Results

Using the impostor generation techniques presented in section 2.2 an *approximate scene* can be quickly generated. The proposed algorithm accomplishes this by mixing actual point cloud meshes with mesh impostors stored from previous frames. An impostor has little to no error if the camera has not changed position since the impostor's generation. The primary cause of differences between the approximate scene and the fully rendered scene is camera movement.

The LIDAR scene is composed of many discrete impostors geometrically represented as quadrilaterals which generate the appearance of a continuous surface when rendered to the screen. However, when the camera changes its position, the boundaries at each edge of an impostor may shift relative to one another breaking the illusion of a continuous surface. These shifts result in gaps in between the impostors causing a phenomenon known as *cracking*. The left side of Fig. 2.5 shows an impostor as viewed from the original camera position; on the right, the camera has moved from its original position, resulting in cracking. In general, cracking only occurs when rendering a continuous surface as a set of impostors; if each impostor represented a single object, such as the Stratotanker in Fig. 1.4, cracking will be avoided. Additional minor errors can also accumulate from the render-to-texture process, as well as round off (partial pixel) errors from the equations described in section 2.2.

Ultimately, an impostor for each mesh is created but only updated when determined by the *update policy*. The update policy balances the requirement for an interactive frame rate against the growing error in each impostor. Impostors with the highest error are updated first; impostors with low errors are reused in subsequent frames. Each impostor's generation time is amortized over its life time - the longer an impostor is reused, the greater the cumulative computational savings. It is the accumulation of these savings that

enables a real-time, interactive display of tens of millions of LIDAR points where it would otherwise not be possible.

Figure 2.5: Left) An approximate scene with no camera motion; Right) a scene with cracking caused by camera motion.

2.4.3.1 Experimental Design

The following experiments were performed using an NVIDIA GeForce 280M graphics card with 1GB of RAM. A LIDAR data set of 10,000,000 data points was used for these experiments; the entire data set was only able to be rendered at 4 frames per second on the testing machine. To achieve an interactive framerate, this update rate must be greater than or equal to 10 frames per second.

This data set was split into 64 subsets of roughly equal volume called *meshes*. 64 subsets were chosen experimentally and represented a good balance between the number of points in each mesh and the total number of meshes needing image-based impostors. This essentially strikes a balance between the manageability of the 64 impostors and the computational savings gained from the 10 million LIDAR points painted across them. To arrive at this number, consider choosing a number too small and a number too large. If only one mesh, encompassing all the points was chosen, the image-based impostor would simply be a picture of all 10 million data points. The cost of generating/updating this impostor is similar to rendering the entire data set using traditional vertex-based rendering. Furthermore, the large size of the impostor will quickly accumulate error as the

camera moves. In many cases, the impostor update time may exceed its useful lifespan. Such an approach limits the flexibility and usefulness of image-based impostors. Similarly, if the largest possible number of subdivisions was chosen: one mesh for each data point, then 10 million image-based impostors would have to be generated. This approach offers significant flexibility but requires managing all 10 million impostors. Furthermore, generating each impostor requires more time than simply rendering each vertex. Finally, each image-based impostor requires six vertices (two triangles) to render; creating one impostor for each LIDAR point essentially multiplies the per frame vertex count of the original data by six. A conservative upper limit to avoid increasing the final vertex count is the total data set size divided by six. At this point, the number of vertices transformed through the pipeline is the same for both the traditional and image-based impostor approach. However, additional computational overhead is required to manage and update those impostors via the update policy. The policy must determine which impostor(s) to update at each frame; the time consumed by the policy is an important consideration: as the number of impostors increases, the time required to compare each impostor's error against all others increases linearly. However, if interactivity is to be ensured, the time consumed to compute each frame must encompass both the policy update time as well as the actual rendering time. Through experimentation, choosing to decompose the large data set into 64 subsets enabled the policy manager to quickly select an update candidate. It also provided enough on-screen granularity to provide sufficient accuracy. In other words, the policy manager was able to efficiently manage all 64 impostors, select candidate(s) to update, and then update those candidate(s) without surpassing its per frame time budget. The scene was also sufficiently decomposed to minimize error caused by out-of-date impostors.

Using the impostor generation technique presented in section 2.3, combined with suitable *error metrics* and *update policies*, the objective was to find an optimal balance between accuracy and interactivity for the LIDAR's visualization.

To quantify the accuracy of the approximate scene, a preplanned flight path containing 827 frames was created. Each frame's width, w , and height, h was 800 by 600, respectively. Each fully rendered frame was stored in a *baseline* image set, B . All image sets were captured using the same flightpath but with a varied combination of update policy, error metric, and number updates per frame, U . Image sets were created for the *least recently used* (LRU) update policy [9] and a *weighted* update policy. Each test was conducted at $U = 1, 2, 4$, and 64 updates per frame.

The LRU policy updates the U meshes that have been updated least recently every frame; whereas, the weighted update policy scores each mesh based on a weighted criteria that values the last update time at twice the contribution as screen space size, and updates the U meshes with the highest scores. This 2:1 weighting ratio was chosen through minor experimentation - a 1:1 weighting ratio did not produce a good metric since some regions were never scored high enough to receive an update. A mesh never scoring high enough to receive an update is a real possibility with any metric that is not *fair*, including the *weighted* update policy. This metric is not considered fair because it cannot *guarantee* that each mesh will receive an update in a deterministic time frame; this is contrary to the aforementioned LRU policy.

Two image based metrics, commonly used for human perception, were used to compare the approximate image sets against the reference image set to generate a quantified error [41]. The first was a luminosity difference [41]. The second was a more sophisticated Haar wavelet based metric [42, 43] that accounted for similarity within subregions instead of focusing on the image as a whole.

The luminosity based metric was based on the International Telecommunication Union - Radiocommunication (ITU-R) Recommendation BT.601 [41] luma coefficients. This ITU-R standard defines the studio encoding parameters of digital television for 4:3 and 16:9 aspect ratios; it is also a standard implemented in the in Motion Pictures Expert Group (MPEG) video encoding standard. For an RGB pixel the corresponding luminosity value, Y can be determined using Eq. 2.13. The examples used in this experiment were monochromatic in nature, so the luminosity metric was not entirely necessary, but this approach does enable the algorithm to generalize to other scenarios.

The Haar wavelet based metric is an alternative approach to the aforementioned luminosity metric. Comparing two images via the luminosity metric involves per-pixel differences throughout the image as a whole. However, human perception is more sensitive to the correlation between sets of local pixels than across the entire image. The Haar wavelet metric is able to compare differences within subregions by recursively subdividing an image; as a result, its quantitative metrics may align more closely to human perception [42, 43].

$$Y(R, G, B) = 0.299R + 0.587G + 0.114B \quad (2.13)$$

A Haar Wavelet [42] based image metric, H , operating on two image sets, B , the baseline, and A , the approximation, was implemented as described in Eq. 2.16, where $\lambda = \log_2 \min(w, h)$ and $X_i(m, n)$ was the RGB triplet representing the pixel in the m^{th} row and n^{th} column in the i^{th} image of image set X :

$$p(B, A, i, m, n) = |Y(B_i(m, n)) - Y(A_i(m, n))| \quad (2.14)$$

$$P(B, A, i, j, k, l) = \sum_{m=w \cdot k/2^j}^{w \cdot (k+1)/2^j} \sum_{n=h \cdot l/2^j}^{h \cdot (l+1)/2^j} p(B, A, i, m, n) * \left(\frac{1}{2}\right)^{j+1} \quad (2.15)$$

$$H(B, A) = \sum_{i=1}^{\text{number of frames}} \sum_{j=0}^{\lambda} \left(\left(\sum_{k=0}^{2^j} \sum_{l=0}^{2^j} P(B, A, i, j, k, l) \right) / (w \cdot h) \right) \quad (2.16)$$

Eq. 2.14 represents the per-pixel contribution to the overall metric. There are λ inspection levels, and at each level the kernel size decreases (increasing the number of kernels). The larger the kernel size, the greater the weight given to that level; such that: the average of the entire image is given 50% of the weight, the averages of the quadrants are given 25% of the weight, the averages of the next level are given 12.5% of the weight, etc. The per-kernel contribution to the overall metric is described in 2.15.

2.4.3.2 Analysis

With 64 meshes, the results for updating 64 meshes per frame are the best possible result the algorithm may produce - each mesh is regenerated each frame. As expected, experiments with fewer updates per frame had higher mean error and standard deviation for both error metrics as seen in Tables 2.1 and 2.2. Fig. 2.6 shows the error for the Haar and Luminosity error metrics using the LRU update policy. Fig. 2.7 shows both error metrics again, but this time for the weighted update policy.

Based on the values in Tables 2.1 and 2.2, the Haar Wavelet based metric exhibits the following qualities that make it a better error metric overall than the luminosity based error metric. For the luminosity metric the range of the means was

$\delta_L = 5.65687 - 5.31108 = .34579$, whereas the range of values of the means of the Haar-based metric were $\delta_H = 4.04175 - 3.61884 = .42291$. This indicates that changes in similarity are more easily perceived by the Haar-based metric. Normalizing these values, over their associated extremes, this separation becomes clearer.

$$\hat{\delta}_L = \delta_L / 5.65687 = 6.213 \times 10^{-2} \text{ and } \hat{\delta}_H = \delta_H / 4.04175 = 1.046 \times 10^{-1}.$$

Figure 2.6: The Haar Wavelet based metric (top 4 lines) and Luminosity metric (bottom 4 lines) plots for each frame for 1, 2, 4, and 64 mesh updates per frame using the LRU policy.

Table 2.1: Results for LRU Metric

LRU Metric Results				
Mesh Updates per Frame	1	2	4	64
Haar Mean	4.04175	3.80513	3.71144	3.61884
Haar σ	2.57608	2.4126	2.33173	2.2758
Luminosity Mean	5.65687	5.46759	5.36976	5.31108
Luminosity σ	3.55703	3.44508	3.37185	3.33513

2.4.3.3 Interactivity

This section presents a method for balancing accuracy and interactivity for large point-cloud datasets. The accuracy of the impostors was quantified in Sec. 2.4.3 using a

Figure 2.7: The Haar Wavelet based metric (top 4 lines) and Luminosity metric (bottom 4 lines) plots for each frame for 1, 2, 4, and 64 mesh updates per frame using the Weighted policy.

Table 2.2: Results for Weighted Metric

Weighted Metric Results				
Mesh Updates per Frame	1	2	4	64
Haar Mean	3.97095	3.77042	3.67629	3.62033
Haar σ	2.5266	2.39203	2.31943	2.27583
Luminosity Mean	5.60678	5.4415	5.36018	5.31565
Luminosity σ	3.534	3.42804	3.36565	3.33098

Luminosity based metric and a Haar wavelet based metric. The average framerate over the 827 frames were captured, see Sec. 2.4.3.1. Both update policies averaged over 104 fps

Table 2.3: Average Framerates for Each Test

Average Framerate for LRU and Weighted Policy				
Mesh Updates per Frame	1	2	4	64
LRU Policy	104.163	54.446	27.000	2.000
Weighted Policy	107.587	55.76087	28.15217	2.000

with 1 update per frame, over 54 fps with 2 updates per frame, and over 27 fps with 4 updates per frame. For specific framerates, refer to Table 2.3. The error metrics were used for evaluation after the fact and were not part of the update policy itself. However, an area of future work involves incorporating the error metrics into the impostor update decision.

The fully rendered scene with 10,000,000 LIDAR points achieved only 4 frames per second. The presented algorithm showed that image-based impostors offered a substantial increase in interactivity and usability without too much loss in accuracy. Furthermore, accuracy can be traded for interactivity (and vice-versa) in fine-grained increments based on the selected number of updated meshes per frame (U).

2.4.3.4 Graphics Hardware Scalability

The presented image-based impostor technique must be rendered via graphics hardware to obtain interactive frame rates. When operating on data sets too large to fit inside the dedicated graphics memory, image-based impostors offer further advantages. If traditional vertex-based rendering is used, the graphics card will be unable to store all the vertices in memory at any time. This forces the GPU to shuffle data from host to graphics memory continuously resulting in a performance hit. However, if using image-based impostors, each mesh is replaced with a single impostor requiring only six vertices and the corresponding texture memory. Since an impostor requires less space than the actual vertices it replaces, more information may be stored within the same amount of graphics

memory. Additionally, when the update policy determines that an impostor needs to be regenerated, only the vertices of that impostor are required to be transferred to graphics memory. In the aforementioned experiment which used 64 meshes, roughly $\frac{1}{64}$ of the complete set is required to be transferred to update the impostor. This is in contrast to the traditional vertex-based technique which requires transferring all of the vertex information from the CPU to the graphics card each frame. This fractional advantage gained by the image-based impostor technique remains constant and scales with newer graphics cards as their memory and bandwidth increase. Given reasonable mesh sizes, as discussed earlier in Sec. 2.4.3.1, available memory bandwidth between the CPU and graphics card is more likely to successfully transfer a small fraction of the data set than the entire data set, giving an advantage to this image-based impostor technique.

3 INTERACTIVE MESOSTRUCTURES

This chapter presents a technique for interactively deforming and colliding with mesostructures at a per-texel level. It is compatible with a broad range of existing mesostructure rendering techniques including both safe and unsafe ray-height field intersection algorithms. This technique integrates well with existing physics engines and is able to replace traditional 3D geometrical deformations (vertex-based) with 2D image space operations (pixel-based) that are parallelized on a GPU without CPU-GPU data shuffling. Additionally, surface and material properties may be specified at a per-texel level enabling a mesostructure to possess varying attributes intrinsic to its surface and collision behavior; furthermore, this offers an image-based alternative to traditional decals. This technique provides a simple way to make almost every surface in a virtual world responsive to user actions and events. It requires no preprocessing time and storage requirements of one additional texture or less. The algorithm uses existing inverse displacement map algorithms as well as existing physics engines and can be easily incorporated into new or existing game pipelines.

3.1 Introduction

Texture mapping [44] is a foundational tool requisite for today's interactive rendering; it increases realism, fidelity, and immersion while preserving interactivity. Since computer graphics' early days, texture data has been continually extended to encapsulate more than a simple image interpolated across a polygon. Blinn [45] used texture data to store perturbed surface normals to enhance a surface's appearance when lit. Cook [46] extended this idea to store a height field within a texture and reposition the corresponding fragments accordingly, i.e., displacement mapping. The evolution of modern GPUs' parallel compute capability combined with a programmable pipeline has changed the implementation of these early algorithms into many different, but generally

ray-traced, approaches yielding realistic mesostructures at interactive frame rates [47]. Policarpo's relief mapping [48] and Tatarchuk's parallax occlusion mapping [39] are excellent examples of inverse displacement mapping algorithms that easily fit into existing game pipelines, with minimal storage, yet add significant surface details to a virtual world.

Current inverse displacement mapping algorithms focus on efficient and/or correct height map rendering, but they *have not explored the potential interactivity aspects associated with these surfaces*. This chapter presents a novel extension to inverse displacement mapping techniques that enables per-textel collision detection, per-textel material properties, and the corresponding dynamic deformations of the impacted surface. This technique integrates with existing physics engines, such as NVIDIA PhysX, is compatible with many popular mesostructure rendering techniques, such as Relief mapping (planar and curved) [3, 48], parallax occlusion mapping [39], and even some safe methods such as Maximal Mipmaps [49], and performs all deformation computations on the GPU without shuffling texture data between the CPU and GPU. Furthermore, this technique requires no preprocessing time. It requires only the existing displacement and normal data consumed by traditional mesostructure rendering algorithms as well as any per-textel material properties desired.

The contributions of this chapter include the following:

- A technique for interactively deforming and colliding with mesostructures. This technique leverages existing physics engines, such as NVIDIA PhysX, for initial collision information; it subsequently performs per-textel tangent space collisions and deforms the affected mesostructure in parallel on the GPU without any server-client texture transfers and minimal CPU involvement.
- Each deformation processed by the GPU is able to account for collider-specific physical attributes as well as per-textel surface properties describing the behavior

and/or constraints of the impacted mesostructure; therefore, a mesostructure's visual and behavioral dynamics can be tailored to other objects, animations, or time.

- These deformable mesostructure operations are compatible with many existing inverse displacement mapping rendering techniques including relief mapping, parallax occlusion mapping, and even some safe methods lending themselves to easy integration into existing game pipelines and applications that use GPU-based inverse displacement mapping techniques.
- Replaces 3D vertex-based deformations with 2D image-based operations performed solely on the GPU, in parallel, and free from CPU-GPU data shuffling.
- Provides a per-textel alternative to traditional decals. This alternative offers enhanced visual cues, requires no additional geometry, and persists for the life-time of the application.

3.2 Related Work

Since Catmull's introduction of texture mapping [44], the applications of *texture data* have continued to evolve. Blinn [45] stored perturbed surface normals in textures creating the appearance of wrinkled surfaces. Cook [46] used displacement maps and perturbed normals to offer correct self-occlusion, shadows, interpenetration, and silhouettes at the expense of a highly micro-polygonized scene. Since then, ray-tracing techniques have been developed that offer the benefits of displacement mapping without the need to render large numbers of micro-polygons.

Patterson [50] proposed inverse displacement mapping, which serves as the basis for modern ray-traced displacement mapping. Instead of perturbing the actual geometry, a texture space ray intersection is performed against a height field and the result is used to position the fragment. In the early 1990s, this CPU based approach was too slow for

interactive computer graphics; what was needed was a vector processor capable of computing thousands of these ray-height field intersections per frame. An alternative approach to ray-traced displacement mapping was Oliveira's relief texture mapping [51]. This technique used textures augmented with depth information and offered 3D surface details and view motion parallax based on 3D image warping equations proposed by McMillan and Bishop [52]. This work helped connect image-based modeling and rendering (IBMR) with traditional polygonal rendering.

By 2005, programmable pipelines had reached consumer graphics cards, and Policarpo and Oliveira [48] proposed mapping relief textures onto polygonal surfaces. The intersection is performed in tangent space [53] and parallelizes efficient ray-height field intersections in the fragment shader offering correct self-occlusions, interpenetrations, motion parallax, shadows, and per-pixel lighting in a compact form. This work was later extended enabling relief mapping to non-height field surface details [54] as well as quadric-based curved surfaces [3], see Sec.3.3.5.

Tatarchuk [39] improved upon Policarpo's technique in three ways. She approximated the 8-bit relief map data using a piecewise linear curve enabling 32-bit ray-height field intersections; this minimized 'slicing' artifacts at grazing angles. Her technique offered soft shadows with penumbra size approximation whereas Policarpo generated hard shadows. Lastly, she presented an adaptive level of detail (LOD) control system such that displacement mapping was only applied in the near field and traditional bump mapping in the far field.

The method by which the ray-height field intersection is computed generally falls into two categories: safe and unsafe. Policarpo and Tatarchuk both emphasize real-time rendering for dynamic height fields (where the displacement map may be altered on the fly), and therefore prefer a fast, approximate intersection algorithm. In such unsafe approaches, one typically marches uniformly along the view ray until penetrating the

height field [39] and then refines the intersection with a binary search [3, 48, 54]. Risser replaced the binary search using the secant method to improve upon convergence time [55] while others have used a Newton-Raphson method for convergence [56, 57].

Depending on the smoothness of the height field and the step size, however, potential intersections may be skipped and the correct result cannot be guaranteed. In applications requiring a correct solution, a safe approach must be used. To avoid inspecting each texel the ray traverses, these techniques require precomputed acceleration structures encoding areas that may be safely skipped. The majority of these structures require off-line preprocessing time and are not conducive to dynamically changing height fields [58–62]. Fortunately, Tev’s work on Maximal Mipmaps showed how it was possible to create an acceleration structure with negligible overhead [49], thereby enabling a safe approach to be used in combination with dynamically changing height fields.

3.3 Interactive Mesostructures

Mesostructures have been traditionally treated as graphical constructs. The small perturbations, bumps, shadows, and occlusions give rise to heightened visual cues, yet interaction with the object, such as collision, and the corresponding deformation still rely upon the underlying macrostructure geometry. *We introduce a technique for interactively deforming and colliding with mesostructures at a per-texel level.* This technique integrates well with existing physics engines, is compatible with a broad range of mesostructure rendering techniques, both safe (ex: Tevs’ Maximal Mipmaps [49]) and unsafe (ex: Policarpo [48], Oliveira [3], Tatarchuk [39]), and is able to replace 3D geometrical deformations (vertex-based) with 2D image space operations (pixel-based) that are parallelized on the GPU without CPU-GPU data shuffling. Furthermore, surface / material properties may be specified at a per-texel level allowing a mesostructure to possess varying attributes such as friction, hardness, restitution coefficients, or even attributes not

Figure 3.1: An interactive mesostructure deformed by spherical impacts. Such an approach provides per-voxel interactivity and higher fidelity than traditional decals. The right column of the image shows the mesostructure's color map, normal map, and displacement map from top to bottom, respectively.

commonly related to computer graphics, such as acoustical properties, magnetism, electrical conductance, temperature, specific heat, density, smell, or any other domain that is able to be captured within a scalar or vector field and sampled at a discrete, per-voxel level. The collisions and deformations with these mesostructures do not necessarily match

physical reality, but offer a believable, efficient method for providing virtual environments and games with interactive, malleable surfaces.

3.3.1 Per-Texel Collision Detection

Figure 3.2: The collision kernel performs the ray-height field intersection using $\vec{uv}$ (non-displaced initial texture coordinate), $\vec{Tc_{ts}}$ (collider's trajectory in tangent space), and any desired mesostructure rendering technique. The kernel returns $uv\vec{disp}$ (texture displacement offset), $depth_{ws}$ (world space depth into the mesostructure), and $\vec{Nc_{ms}}$ (model space contact normal) associated with the intersection point between $\vec{Tc_{ts}}$ and the height field. The displaced UV coordinate, $\vec{uv} + uv\vec{disp}$, is denoted by the pink dot; the $depth$ scalar is the distance from the pink dot to the intersection point along the inverse of macrogeometry's normal vector. The overlapping bounding volumes of the two objects are shown in green. Care must be taken to ensure that $uv\vec{disp}$ and $depth$ are mapped into world space from tangent/texture space prior to world space use; this scalar multiplication can be performed at the end of the GPU Collision Kernel or any other convenient point prior to calculation of $\vec{C_{ws}}$ (world space collision point), as described in item 6 of the Collision Kernel algorithm in Sec.3.3.1.

This technique makes use of existing physics engines that support volume intersection; NVIDIA PhysX, Bullet Physics, Havoc Physics, Open Dynamics Engine, among others support this common feature. When the bounding volume of a mesostructured surface is penetrated by a colliding object, the collision information provided by the physics engine is passed as input to a set of GPU kernels for processing, see Figs.3.2,3.4. The GPU kernels compute the contact point lying on the mesostructure, the deformation characteristics to be imparted upon the mesostructure, and the corrective forces to be applied upon the collider. Within these kernels, the GPU may also account for surface / material properties intrinsic to the mesostructure. These properties may vary per texel and may be described by either a scalar or vector field since the material data is encoded as a texture accessible to the kernel.

In general, when the bounding volumes of two objects begin to intersect, state regarding the collision's progress must be maintained. The aforementioned physics engines generally integrate time across fixed, discrete intervals; therefore, two bounding volumes may intersect for more than one time step. During each step, the physics engine applies corrective forces on each body, eventually repelling them from one another to simulate the behavior of a natural collision. The state of the collision persists from the initial intersection until the corresponding bounding volumes no longer overlap.

This technique extends this concept by using canonical mesostructure rendering techniques to determine precisely when a collider is in contact with the displaced mesostructure surface and only applies corrective forces at such times, see Fig.3.2. By doing so, the surface described by the displacement map effectively becomes the surface upon which collisions occur (as opposed to the macrogeometry). Furthermore, since the GPU computes the texture space collision point ($u\vec{v} + uv_{disp}\vec{v}$ in Fig.3.2) and has access to all the server-side textures, the kernel can efficiently perform additional operations such as mesostructure deformation. By deforming the displacement map texture as a function of

the collider's impact, one effectively reduces a traditional three dimensional, vertex-based deformation to a two dimensional, image-based operation. Such image-based operations are efficiently parallelizable on modern GPUs and require no client-server texture shuffling when compared to vertex-based deformations. Additionally, the resolution of this deformation is at the per-textel level, which for a mesostructure is by definition, higher than the fidelity of the macrogeometry. As shown in Fig.3.1, a mesostructure has been impacted by many collisions. The macrogeometry is composed of exactly two triangles, yet the collision and deformation behavior is similar to a micro-polygonized mesh where the number of faces equals the number of texels in the displacement/normal maps.

This technique can be decomposed into two major GPU kernels: the first kernel, see Fig.3.2, determines the initial collision information between the collider and mesostructure. The second kernel, see Fig.3.4 and Sec.3.3.2, at the instance(s) of collision with the mesostructure's surface, deforms the mesostructure according to the collision dynamics, surface properties, and the collider's physical attributes. The initial collision GPU kernel returns: 1) the offset, $uv_{disp}^{\vec{}}$, from $u\vec{v}$ to the texture space contact point (pink dot in Fig.3.2); 2) the world space depth, $depth_{ws}$, into the mesostructure from $u\vec{v} + uv_{disp}^{\vec{}}$ to the depth of height field intersection; and 3) the associated contact normal, $N_{c_{mc}}^{\vec{}}$, of the height-field contact point (for consumption by the physics engine and calculation of the bounce angle and/or restitution forces). The deformation GPU kernel parallelizes texture updates approximating 3D vertex-based deformations as 2D image-based operations; the kernel may also return computed corrective forces for consumption by the physics engine as well as other updates, see Sec.3.3.2. This section's subsequent text describes how this technique is achieved, Fig.3.3 illustrates the entire state machine.

Upon an initial collision notification between the physics engine's bounding volumes of a mesostructure and a collider, the following occurs, see Fig.3.2:

1. Create a new state machine linking the collider with the mesostructure; append this state to the mesostructure's list of current collisions.
2. Let $T\vec{c}_{ws}$ be the collider's world-space trajectory vector. Intersect $T\vec{c}_{ws}$ against the mesostructure's macrogeometry to retrieve the 3D intersection point, $\vec{C}_{uv}$, and the intersected face, f . Optionally, to accelerate this intersection, one may transform $T\vec{c}_{ws}$ to the mesostructure's local object space and use a local hierarchial bounding volume, such as an octree for logarithmic acceleration. Performing intersection tests on candidate faces in object space requires only one transformation of $T\vec{c}_{ws}$ into object space instead of transforming each candidate face into world space.
3. Using f and $\vec{C}_{uv}$, compute the mesostructure's non-displaced texture coordinate, $u\vec{v}$. $u\vec{v}$ may be computed using a Barycentric coordinate system on f and linearly interpolating for $\vec{C}_{uv}$ [63]. At this point, it may also be convenient to compute a scale factor, $s\vec{f}$, which maps the lengths of f in texture space to the lengths of f in object space. This scaling can then be used inside the GPU kernel to map uv texture displacements to object-space offsets along the plane defined by f . With respect to traditional inverse displacement mapping, $u\vec{v}$ represents the starting position for the ray-height field intersection; it is analogous to the view vector's initial intersection with the macrostructure's surface in Policarpo's planar relief mapping [48].
4. Transform the collider's trajectory vector, $T\vec{c}_{ws}$, from world space to tangent space, $T\vec{c}_{ts}$ using f 's corresponding Tangent, Binormal, Normal (TBN) matrix.
5. Dispatch the GPU Collision Kernel with input $T\vec{c}_{ts}$ and $u\vec{v}$, see Fig.3.2. The GPU kernel returns 1) the displaced texture coordinate offset $uv\vec{disp}$; 2) the world-space depth into the mesostructure, $depth_{ws}$; and 3) the model space normal vector, $N\vec{c}_{ms}$, associated with $u\vec{v} + uv\vec{disp}$. The scale vector, $s\vec{f}$ (see item 3 above), may be

consumed by the kernel to map $uv_{disp}^{\vec{}}$ and $depth$ from texture space into world space.

6. The world space 3D collision point between the collider and mesostructure surface

$$\text{is } \vec{C}_{ws} = \vec{C}_{uv} + \frac{T\vec{c}_{ws}}{\|T\vec{c}_{ws}\|} \cdot \sqrt{depth_{ws}^2 + \|uv_{disp_{ws}}^{\vec{}}\|^2}.$$

7. Store all aforementioned computed values inside the state machine specific to this collision.

Figure 3.3: The right side shows a mesostructure with several current colliders. Each collider is paired with state information managed by the mesostructure. Each collider-state pair contains the information shown in the left box. This state is updated via callbacks within the physics engine when colliders enter or leave the mesostructure's bounding volume. The red box denotes information computed within the GPU kernels. The green box denotes information required prior to the GPU kernel dispatch.

Within the kernel, the displacement map and any supporting acceleration structures may be utilized such as Maximal Mipmaps. Beginning at texture coordinate $u\vec{v}$, in the

direction of $T\vec{c}_{ts}$, the kernel may implement any desired displacement map rendering technique. Note, however, the notion of many view rays cast from the eye to each fragment is replaced by a single trajectory vector, $T\vec{c}_{ts}$, originating from the collider. Thus, only a single thread needs to be dispatched to compute $uv_{disp}^{\vec{}}$, $depth_{ws}$, and $N\vec{c}_{ms}$ requiring less computation than a typical displacement mapped set of fragments. Unsafe methods, such as Policarpo’s relief mapping [48], Tatarchuk’s parallax occlusion mapping, or Risser’s secant-based displacement mapping [55], are excellent candidates; safe methods, such as Kolb and Rezk-Salama’s dilation and erosion maps [64], Oh’s pyramidal displacement maps [62], Donnely’s sphere tracing [60], Dummer’s cone stepping [65], Oliveira and Policarpo’s relaxed cone stepping [61], or Tev’s Maximal Mipmaps [49], are also potential candidates. It is worth noting, however, that the majority of these safe techniques require acceleration structures that cannot be precomputed on a per-frame basis (with the exception of Tev’s Maximal Mipmaps). As a result, they are not compatible with dynamic deformation as discussed in 3.3.2. If dynamic deformation is not needed, a safe technique may be preferable as it guarantees the correct intersection point; otherwise, Tev’s Maximal Mipmaps is a good choice for a safe method that is dynamically deformable. This kernel can also utilize inverse displacement algorithms that operate on curved surfaces such as Oliveira’s quadric-based relief mapping [3], see Sec.3.3.5 for details.

Although the above describes using a single trajectory vector, one may also use a set of trajectory vectors to accommodate the geometry of more complex colliders. Each vector may correspond to a different face or salient point on the collider enabling one to model the edges, corners, and geometry of a more complex collider. When a salient point(s) collides with the mesostructure, the corresponding restitution force(s) can be summed and applied to the collider at the collision point(s) causing a realistic rotational response. This approach, for example, would enable a rectangular prism to collide with a

mesostructure at a corner or an edge, and begin to tumble in a believable manner. Furthermore, the GPU deformation kernel could be passed the corner(s) or edge(s) attributes and deform the displacement map as though it was imprinted with the prism's corner(s) or edge(s). An extension to increase the deformation accuracy involves rendering the impacting corner(s) or edge(s) into a depth map and using that depth map as part of the image operator in the GPU deformation kernel. Although this approach requires more computational effort, it offers highly detailed deformations. Since collisions only occur at salient points, it is assumed that, in general, the mesostructure is larger than the collider. If the collider was significantly larger than the mesostructure, the two volumes could overlap, yet it may be unlikely that a salient point of the collider would ever touch the mesostructure's surface; in this case, the two objects could pass through one another resulting in a non-realistic faux collision. If mesostructures are used to add surface details to floors, walls, and/or other larger objects in a virtual world, the smaller, dynamic objects will not be able to pass through the mesostructure's surface without a salient point intersecting the mesostructure.

(a) The Deformation kernel is passed in $\vec{uv} + uv_{disp}$ and collider-specific attributes; these two values are then used in conjunction with the server-side displacement map and per-texel surface property map to generate and/or select an image operation that will be applied to the displacement map at location $\vec{uv} + uv_{disp}$. Subsequently, the deformed displacement map is used to generate a corresponding normal map; a linearly separable Sobel kernel will efficiently generate a new normal map based on the deformed displacement map's gradient. The region deformed about $\vec{uv} + uv_{disp}$ lies within the red rounded-rectangles.

(b) The final output of the GPU Deformation kernel is shown on the right side - the deformed region lies within the red rounded-rectangle. The original input prior to deformation is shown on the left side - the region to be deformed lies within the red rounded-rectangle.

Figure 3.4: The GPU Deformation Kernel applies an image operator about the collision point, $uv + uv_{disp}$, while accounting for collider-specific attributes and per-texel mesostructure surface properties. The image operator deforms the displacement map accordingly; subsequently, the normal map is updated using the newly deformed displacement map, see Fig.3.4a. The before and after images of the deformation kernel are shown in Fig.3.4b.

Up to this point, we have discussed how to compute the initial intersection point between a collider and mesostructure. However, all subsequent time steps where the two bounding volumes overlap require housekeeping. The following list describes how to deal with changes in either object and ensure the computed contact points remain up to date. Upon a subsequent notification of a continuing collision between bounding volumes of a mesostructure and a collider, the following occurs:

1. Retrieve the state machine corresponding to this collision from the mesostructure's list of current collisions.
2. If the collider's position is within a user defined epsilon of $\vec{C}_{ws}$, a true collision with the mesostructure's surface has occurred. Therefore, dispatch a deformation kernel on the GPU, see Fig.3.4 and Sec.3.3.2. This kernel uses the existing collision state information, along with any optional surface property information (see Sec.3.3.4), to approximate the 3D deformation by performing a 2D image-based operation on the displacement map texture (Fig.3.4a); this kernel may also compute and return corrective forces to be applied onto the collider via the physics engine's API. Subsequently, the mesostructure's normal map is updated along with any additional maps such as specular maps, ambient occlusion maps, or other maps/data. Each update is parallelized across the GPU decreasing processing time and avoiding CPU-GPU data shuffling. Finally, the corrective forces computed by the deformation kernel may be applied upon the collider via the physics engine's API.
3. If $T\vec{C}_{ws}$ has changed since the previous notification, update the collision state machine using the new $T\vec{C}_{ws}$ and the steps defined in the previous list; this searches for any possible collision(s) along the collider's new trajectory.
4. When the physics engine notifies the state machine that the bounding volumes are no longer intersecting, the collision has fully completed and the corresponding collision state is removed from the mesostructure's list of current collisions.

Each time step, the physics engine updates its collision state for each pair of colliding objects. The extended collision state checks this information to determine if the bounding volumes are still intersecting. If so, the collider's distance from $\vec{C}_{ws}$ (or the set of all $\{\vec{C}_{ws1} \dots \vec{C}_{wsN}\}$) is compared, within some epsilon, to determine if any true contact with the mesostructure's displaced surface has occurred. If collision(s) exist, the deformation

kernel is dispatched. This kernel takes as input, $\vec{u}\vec{v} + uv\vec{disp}$, $N\vec{c}_{ms}$ (or $\{u\vec{v}_1 + uv\vec{disp}_1 \dots u\vec{v}_n + uv\vec{disp}_n\}$), and any desired collider attributes; for example, if the collider is a sphere, the radius may be passed to the kernel defining the radius of the ‘crater’ to be applied upon the mesostructure. This concept can also be used to enumerate different classes of image operations the kernel applies to the displacement map texture. In this manner, physical characteristics specific to a collider will result in correspondingly specific deformations on the mesostructure, see Sec.3.3.2.

If the mesostructure is manipulated (change in pose or animation) or the collider’s trajectory is changed midway through a collision, the collision state information may be recomputed to ensure accurate collision points. This is not necessarily required, however, since one may assume the exterior of the mesostructure’s bounding volume is ‘close enough’ to the mesostructure’s surface that any offset incurred from a change in trajectory or mesostructure state will not be large enough to be of noticeable concern nor justify the expense of an additional kernel dispatch. This choice, nevertheless, is application specific - updating the collision state each frame ensures accurate collisions. If the trajectory changes such that the collider exits the volume without ever intersecting the mesostructure, the collision state is simply removed from the mesostructure’s list of current collisions and the deformation kernel is never dispatched. Similarly, if a collision occurs and a deformation kernel is dispatched, the collision state is only removed when notified by the physics engine that the bounding volumes cease to intersect.

3.3.2 Dynamic Deformation, Animation, and Decals

Since mesostructures are internally represented as a 2D height field stored as a displacement map and only acquire a third dimension via the rendering pipeline, image-based operations are a natural means of manipulation. Fortunately, such manipulation benefits from the massively parallel processing power of modern GPUs;

Figure 3.5: Normal and displacement maps before and after a collision, respectively. The displacement map texture is manipulated first and the normal map is regenerated from the updated displacement texture. The deformation process only requires a 2D image-based manipulation to approximate 3D deformations. The contact normal on the left side of this image is consumed by the physics engine to compute corrective force vectors.

furthermore, interoperability between graphical languages such, as OpenGL / DirectX, and GPGPU languages, such as NVIDIA's CUDA, enables a GPU kernel to directly operate on OpenGL / DirectX textures bypassing the traditional need of CPU-GPU texture

shuffling. OpenGL 4.3's Compute Shaders further empower such behavior. This two-fold advantage makes per-frame texture manipulation a viable approach for real-time graphics.

This technique uses image-based operations on the displacement map to approximate deformations incurred by a collider as shown in Figs.3.4 and 3.6. This operation may be tailored to each specific collision since, at the time of impact, the GPU kernel knows the precise coordinate(s) of contact, $\vec{uv} + uv\vec{disp}$, the collider's trajectory(s), the mesostructure's surface properties at and surrounding $\vec{uv} + uv\vec{disp}$, and the collider's enumerated attributes. Together these inputs support many permutations that can be parallelized efficiently, in contrast to traditional 3D vertex-based deformations. Furthermore, many 2D image-based manipulation effects, such as those commonly used in image editors, become applicable to this domain and may be used to enhance the collider's effect on the mesostructure.

On a similar note, deformation need not occur only when in contact with a collider. This technique allows for mesostructures to change as a function of time or any other desired variable; for example, one could simulate water [66], periodic motion, or other animated behavior. The image operations result in a smooth 3D deformation eliminating 'vertex popping' that results from vertices being repositioned, such as during a geometric-based level of detail transition.

This technique also provides an alternative to traditional decals. Surfaces affected by impacts, such as bullets, exploding debris, or other environment manipulations are commonly updated by placing a decal over the impact area giving the illusion of interactivity. Most modern games limit the number of decals visible at any one time leading to 'disappearing bullet holes' or 'self-cleaning blast zones'. Such a limit is reasonable as each decal inserts more geometry into the world and needs to be depth-positioned carefully to avoid z-hitting with the underlying surface and any other underlying decals. By accepting collider-specific attributes, such as 'bullet', 'laser beam', 'paint ball', etcetera, the deformation kernel can be correspondingly specific in how each

Figure 3.6: Identically color textured mesostructures, each deformed differently. The top surface was impacted by several spheres, some regions were struck multiple times accumulating deeper cavities - a useful advantage over traditional decals. The middle surface was deformed via radial sinusoidal waves emanating from differing points causing interference with one another; the corresponding color, normal, displacement, and specular maps are shown on the right side of the image, top to bottom, respectively. The displacement map shows the origins of each wave. The lower surface has not yet been deformed.

deformation occurs. For example, given a ‘bullet’ and its trajectory, a precise skid mark can be generated; given a ‘laser beam’ and its angle of incidence, a burn imprint affecting

Figure 3.7: A dynamically deformed box punctured by collisions. Notice colliders are able to enter inside the box after breaking through the depth described by the mesostructure's height map.

both the color map and height map can be generated; given a 'paint ball' and trajectory, a realistic splat can be generated. At no point is any additional geometry or texture data inserted. Textures are merely manipulated and accumulated onto themselves; as a result, continual deformation is limited only by texture resolution. At this point, we can now

incorporate the per-textel surface properties into these decal deformations. For example, if the ‘paint ball’ is colliding with a mesostructure with a per-textel ‘hardness’ field, the ball may reflect off leaving only a dent if the contact point is below a certain ‘hardness’ value; otherwise, the ‘paint ball’ may burst and cause the deformation kernel to invoke a splat image operation. Equation 3.1 shows a piecewise function defining which image operator to apply at the contact point $\vec{u}\vec{v} + uv\vec{disp}$, given the collider-specific attributes and per-textel surface properties of the mesostructure. The permutations grow large quickly offering great diversity per collider per surface.

$$\begin{aligned}
 c &= \text{ColliderSpecificAttribute} \\
 y &= \text{HardnessSurfacePropertyAt}(\vec{u}\vec{v} + uv\vec{disp}) \\
 I(c, y) &= \begin{cases} \text{splat}(y, T\vec{c}_{ts}) & : c = pball, S \leq y \\ \text{dent}(y, T\vec{c}_{ts}) & : c = pball, D \leq y < S \\ \text{smudge}(y, T\vec{c}_{ts}) & : c = pball, 0 \leq y < D \end{cases} \quad (3.1)
 \end{aligned}$$

Given the corresponding surface properties, collider trajectory, and enumerated collider attributes, the deformation kernel selects an appropriate image-operation to apply about the contact point $\vec{u}\vec{v} + uv\vec{disp}$. Upon completion of the image operation, the displacement map has been updated; subsequently, the corresponding normal map is updated based on the new displacement map. This can be efficiently performed using a linearly separable Sobel kernel [67] to generate a normal map from the displacement map’s gradient. As shown in Fig.3.5, a simple circular image operator decreased the scalar values of the displacement map (according to the radius of the collider) about the collision point. Afterwards, a linearly separable Sobel kernel passed over this new displacement

map to generate the updated normal map, as shown on the right side. Selecting the ‘best’ image operator is application specific and enables one to trade fidelity for performance or vice-versa.

3.3.2.1 Texture Repetition

When manipulating textures using this technique, the common practice of texture repeats needs to be considered. If a texture is repeated across many surfaces, any corresponding deformation on that texture will be mirrored across all of those surfaces. In some cases, this may be desired, see Fig.3.12; in other cases, this may lead to unexpected changes on a surface. This may be mitigated by loading multiple copies of a texture as the need arises; however, texture duplication is not a general solution for scenes that rely on repeated textures. Application of this technique should consider these potential limitations.

3.3.2.2 Simultaneous Mesostructure Collisions

The presented technique assumes each mesostructure maintains its own state about which colliders currently reside in its own volume as shown in Fig. 3.3. If the bounding volumes of two mesostructures overlap, a collider could simultaneously reside in two volumes. In the situation where a collider strikes both mesostructures simultaneously, both mesostructures will register a collision and dispatch a deformation kernel. Normally, this produces a plausible effect - each mesostructure deforms as expected. However, if the surface properties of each mesostructure depend on the force of the collision, the presented technique may not divide the collision forces properly to each mesostructure unless the two mesostructures are extended to communicate with one another. While this drawback can be overcome, the intent of an interactive mesostructure is to create a plausible interaction between a collider and a mesostructure resulting in a plausible deformation, instead of a perfectly modeled physical collision.

3.3.3 Dynamic Hole Creation and Constant Genus

As shown in Fig.3.7, colliders are able to deform, puncture, and penetrate through a mesostructure. Since the GPU Collision Kernel is responsible for reporting the location of the collider's impact point within the mesostructure, one may allow the kernel to report that no actual collision has occurred. This will enable a collider to pass through the mesostructure's surface. Since the kernel returns a collision point in the form of $uv\vec{disp}$ and $depth_{ws}$, see Fig.3.2, one may simply define a maximal depth d_{max} . Any returned $depth_{ws}$ greater than d_{max} indicates a 'hole' has been encountered and the collider may then be allowed to pass through the surface. In this way, the displacement map is able to encode areas representing 'holes'. When rendering the mesostructure, the fragment shader can discard any fragment with a corresponding depth greater than d_{max} . This way, physical and visual coherence remains at a per-texel level across the mesostructure.

Since a deformation may impact and decrease the displacement map's height over time, areas that were once elevated may slowly erode into a hole through which a viewer can see and a collider may traverse. In this manner, the underlying geometry's genus is preserved; the physically interactive hole(s) formed within the mesostructure surface are image-space constructs. Figure 3.8 shows a gradual deformation of a mesostructure resulting in a hole through which a collider is able to traverse.

3.3.4 Surface Properties

As shown in Fig.3.4a, the deformation kernel may choose to use per-texel surface properties. These surface properties are encoded within a texture as either a scalar or vector field. If one overlays this texture upon the displacement map, any given texture coordinate, uv , corresponds to the same texel in both maps, thereby describing the surface of each texel on the displacement map.

Figure 3.8: A stone wall eroded by collisions. The rendering fragment shader discards fragments where $d_{max} > depth_{ws}$. Holes formed by colliders, as well as areas with small height values in the displacement map, appear invisible and interact as empty space with the physics engine.

The per-textel surface properties may encode a myriad of attributes; some common ones may include ‘bounciness’, ‘hardness’, ‘brittleness’, etcetera. However, more exotic choices may be useful for certain applications. For example, imagine a game where a player must locate a secret passage below a stone floor and has only one charge capable of creating an opening. The stone floor could encode acoustic properties such that a hollow stone echoes differently when walked upon. Furthermore, a second scalar in the surface map may encode a lower ‘hardness’ level, enabling the charge to break through the floor only at specific texels. Similarly, one could encode temperature on a per texel basis, if a texel’s temperature rises above a threshold, due to some event, that texel’s malleability increases enabling deformation of a previously non-deformable object. Many other material properties can be encoded in a similar fashion offering a general mechanism with per-textel descriptions.

Since the collider's contact point(s) and trajectory(s) are known, a more intricate deformation may be approximated; for example, a collider striking a 'brittle' area of a mesostructure may result in a 'chipping'/'fracturing' image-operation about the collision point whereas a collider striking a 'clay-like' area of the mesostructure may result in a 'crater-like' image-operation about the contact point. These properties may also affect the corrective forces applied upon the collider; for example, contact with a 'bouncy' area may result in a more energetic corrective force than contact with a 'soft' area.

These surface properties may be encoded in their own textures or may optionally be placed in existing color channels that are not used; for example, displacement maps are commonly stored in the alpha channel of a normal map, and similarly, surface properties may be stored in a specular map or ambient occlusion map's unused color channels. Depending on the application and unused texture channels, per-textel surface properties may require no additional space.

3.3.5 Applying to Curved Surfaces

All of the aforementioned applications of interactive mesostructures described in Section 3.3 can be applied to curved surfaces as well. One simply needs to extend the aforementioned GPU Collision Kernel, see Fig.3.2, to account for the underlying curvature representation. Figures 3.9, 3.10, 3.11, and 3.12 show mesostructures applied onto curved surfaces. These surfaces still support per-textel collision detection, dynamic deformation, animations, decals, collider puncturing, and surface properties. The following discusses the drawbacks of traditional planar inverse displacement mapping on non-planar triangular meshes and shows how these drawbacks can be overcome using existing techniques [3].

When applying planar mesostructures on a non-planar triangular mesh, one may observe a lack of surface details at the silhouette edge, as shown on the left side of

Figure 3.9: Colliders have impacted, deformed, and punctured this curved surface.

Fig.3.11. This occurs because planar inverse displacement mapping algorithms assume the underlying macrogeometry is locally flat, see Fig.3.2. Without modeling the macrogeometry as a curved surface, the rendering algorithm fails to place each fragment correctly. The GPU kernel will similarly fail to compute the collision points correctly. Fortunately, Oliveira and Polcarpo proposed an efficient method to naturally extend inverse displacement mapping to curved surfaces, as discussed in [3] and elaborated in

Figure 3.10: A dynamically deformable curved mesostructure. Using Oliveira and Policarpo's rendering technique, one may avoid the drawbacks of planar inverse displacement mapping.

Policarpo's book [68]. This technique applies curvature onto existing triangular meshes by approximating the local curvature at each vertex as a second order quadric; the quadric's two coefficients are computed in a preprocessing phase and stored per vertex. These preprocessed coefficients are extracted from a triangular mesh using a technique proposed by Petitjean [69]. During rendering, the fragment shader curves the view ray according to the interpolated quadric while performing the usual ray-height field intersection. Using this technique, silhouette edge details are properly generated. As shown in Fig.3.11, notice how the left silhouette edge of the sphere is flat and the macrogeometry is easily

discernable. Conversely, notice how the right silhouette edge of the sphere avoids excessive flattening while appearing more natural.

Figure 3.11: A partial sphere rendered using planar inverse displacement mapping on the left side. The same partial sphere rendered (and horizontally mirrored) using curved quadric-based inverse displacement mapping on the right side. Notice the improved silhouette edges on the right half and the flattened edges on the left half.

There are two pieces of information that these curved surfaces require beyond the planar inverse displacement mapping counterpart. The first is the aforementioned per-vertex quadric coefficients. The second is the UV texture dimensions, defined in tangent space, across the set of faces neighboring each vertex. This UV scalar is required to map the tangent space view vector's length into texture space when intersecting against the curved height field as discussed in [3]. Correspondingly, the two quadric coefficients as well as the two UV scalar coefficients need to be stored per-vertex and interpolated across each face inside the fragment shader. Once these values are computed and stored, the curved intersection is similar in complexity to the planar version, however, it still requires a preprocessing phase. To avoid this caveat, an area of future work involves using the geometry shader to compute and emit the quadric and UV mapping scalars.

3.3.5.1 Approximating Local Curvature via Quadrics

As originally described by Oliveira in [3], local curvature is approximated at each vertex during preprocessing. Let T be the set of triangles sharing a vertex v_k and let $V = \{v_1, v_2, \dots, v_n\}$ be the set of vertices in T . The coordinates of v_k are (x_k, y_k, z_k) , specified in the local tangent space of V_k . The relative differences between v_k and each of its neighbors can then be computed, $V' = \{v'_1, v'_2, \dots, v'_n\}$, where $v'_i = (x'_i, y'_i, z'_i) = (x_i - x_k, y_i - y_k, z_i - z_k)$. Using the 3D triplets of each vertex in V' , the a and b quadric coefficients for v_k are then computed using Petitjean's [69] method for extracting quadrics from a triangular mesh. Furthermore, using the texture coordinates corresponding to each vertex in V , the UV texture dimensions of T , defined in tangent space, can be computed; these dimensions are needed to map the tangent space view vector's length into texture space when intersecting against the curved height field.

To compute a compact representation of curvature, a second order quadric is fit to the vertices as shown in Eq.3.2. The a and b coefficients are computed by solving the linear

Figure 3.12: Top left: Time-animated waveform of displacement map on a sphere. Top Right: Curvature of a relief mapped saddle point on a Gordian Knot. Bottom Left: Parabolic tops of relief mapped Gordian Knot. Bottom Right: Time-animated waveform of displacement map on a sphere.

system $\mathbf{Ax} = \mathbf{b}$ as shown in Eq.3.3. Oliveira notes that a least-squares solution,

$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$, is adequate for this approach and requires an inversion of a 2×2 matrix.

$$z = ax^2 + by^2 \quad (3.2)$$

$$\begin{pmatrix} x_1'^2 & y_1'^2 \\ x_2'^2 & y_2'^2 \\ \dots & \dots \\ x_n'^2 & y_n'^2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} z_1' \\ z_2' \\ \dots \\ z_n' \end{pmatrix} \quad (3.3)$$

At this point, the per-vertex curvature coefficients, a and b , as well as the UV texture dimensions (measured in tangent space) of T , (u, v) , have been computed. These four scalars may be stored per-vertex and accessed within the GPU's shaders during rendering.

3.3.5.2 Approximating Curved Ray-Height field Intersection

The local curvature of the mesostructure's displacement map is now augmented by the second order quadric defined with coefficients a and b and centered about its local vertex, see Eq.3.2. Figure 3.13, taken from Oliveira [3], shows how the quadric's curvature affects the interpretation of the displacement map. The ray-height field algorithm must account for this curvature in order to produce natural imagery and interaction.

In both the planar (see Sec.3.3) and curved ray-height field intersections, the algorithm steps along the view vector some amount and then checks if the height of its current location is *below* the height of the displacement map at that same location. However, for the curved case, the height field has been effectively curved by the quadric, see Fig.3.13, and the ray-height field intersection must account for this. Mathematically, however, instead of curving the height field itself, the view ray, V , is instead curved along its trajectory as it steps into the mesostructure. The amount of curvature is given by V 's parametric distance, t , into the mesostructure. Curving a single ray by offsetting its z position is less expensive than recomputing the entire displacement map using the quadric. Let point $P = (V_x t, V_y t, V_z t) = (P_x, P_y, P_z)$ be a location that is evaluated along the view vector, V , at a parametric distance, t . To account for the quadric's curvature, one first computes the quadric's height, Q_z , at (P_x, P_y) . Subsequently, the curved position of P is

Figure 3.13: Taken from Oliveira [3]. The left figure shows a normal displacement map being traversed by a view ray; this is the exact same procedure as illustrated in Fig.3.2. The right figure shows the same displacement map after being curved by the quadric. Mathematically, however, instead of transforming the height field itself, the view ray is instead curved along its trajectory into the mesostructure.

then computed as $P' = (P_x, P_y, P_z - t^2 Q_z)$. It is useful to note that although the view vector has been curved, the scalar height values within the displacement map's texture have not been altered; hence, the subtraction of the $t^2 Q_z$ term from P_z is necessary to curve the view ray as a function of its distance into the mesostructure. Subsequently, P'_z may be used in the height field comparison to determine if P' has passed below the surface of the curved mesostructure.

3.4 Results

The presented algorithm and corresponding GPU Kernels were implemented twice: once using NVIDIA CUDA 4.1 and the second using OpenGL 4.3 Compute Shaders. NVIDIA CUDA relied upon the CUDA / GL interoperability whereas the GL Compute Shaders supported this natively. The following two subsections describe each

implementation. Appendix A.1 lists the OpenGL 4.3 Compute Shader implementation for planar surfaces and Appendix A.2 lists the OpenGL 4.3 Compute Shader implementation for curved surfaces.

3.4.1 Implementation using CUDA

Table 3.1: Performance matrix listing execution time of the collision and deformation kernels on a NVIDIA GTX 580 GPU and NVIDIA GTX 560M GPU using CUDA 4.1. The corresponding host machines included a Core i7-2600K CPU @ 3.4Ghz with 32 GB of RAM and a Core i7-2670QM @ 2.2 Ghz with 16 GB of RAM, respectively.

Performance Matrix (10,000 Iterations) (msec)		
<i>GPU Dispatch</i>	<i>GPU Kernel</i>	<i>Avg Time</i>
Collision Dispatch NVIDIA 580 GTX	Intersection Kernel	0.415422
Deformation Dispatch NVIDIA 580 GTX	Disp Map Deformation	0.176633
	Normal Map Generation	0.107206
Collision Dispatch NVIDIA 560M GTX	Intersection Kernel	1.412999
Deformation Dispatch NVIDIA 560M GTX	Disp Map Deformation	1.253339
	Normal Map Generation	0.310065

The first implementation used OpenGL for rendering, NVIDIA CUDA 4.1 for GPU kernel execution, and NVIDIA PhysX 3.1.1 as the physics engine. CUDA provided the necessary interoperability between OpenGL and the kernel implementations. OpenGL textures were able to be mapped into the memory space of a CUDA kernel, manipulated, and returned to GL without requiring any CPU-GPU data shuffling.

In terms of performance, there were two separate kernels consuming resources. The GPU Collision Kernel required essentially one mesostructure render of J fragments to compute the collision point(s), where J is the cardinality of $\{T\vec{c}_{ts1} \dots T\vec{c}_{tsN}\}$ (number of trajectory vectors cast from the collider upon the mesostructure). Therefore, the GPU Collision Kernel required significantly less computational resources than a typical mesostructure render; however, the presented CUDA implementation was not optimal and required more time than a pipelined render of displacement mapped fragments. Subsequently, no further computation was required unless the collider's trajectory changed or the collider intersected the mesostructure. Upon intersection, the GPU Deformation Kernel was dispatched. The GPU Deformation Kernel 1) deformed the displacement map according to an image-operator, 2) updated any per-textel surface properties, if necessary (see Sec.3.3.4), and 3) updated the normal map using the deformed displacement map algorithm described in 3.3.1.

Table 3.1 lists the average execution time of the collision dispatch, deformation dispatch, and the constituent kernels. Each row in the table was dispatched 10,000 times and its average execution time recorded. The kernels denote only GPU execution time. The intersection kernel required the most execution time on both cards. It was responsible for performing the ray-height field intersection and was implemented using Policarpo's Real-Time Relief Mapping algorithm with 30 linear steps and 5 binary steps [48]. Execution time for a single ray, $T\vec{c}_{ts}$, or a set of rays, $\{T\vec{c}_{ts1} \dots T\vec{c}_{tsN}\}$, required times on the same order since the GPU performed each cast in parallel and accessed the displacement map in a read-only manner. The deformation kernel required each thread to read its corresponding texel, transform the texel's value based on the image operator (accounting for surface properties), see Fig.3.4, and output the new value. The normal map generation kernel required each thread to read the updated displacement map and

output the normal vector at its corresponding position; this can be efficiently implemented with a linearly separable Sobel kernel.

The GTX 580 GPU out performed the GTX 560M GPU for each of the kernels by a factor of 3.4, 7.1, and 2.9, respectively. Throughout the tests, the GTX 580 GPU did not become fill-rate limited whereas the GTX 560M GPU did when rendering 1920x1080 displacement mapped fragments. These times did not reflect the dispatch and scheduling overhead or the CUDA/GL interoperability overhead. Fortunately, two factors mitigated this GPU overhead: 1) no other threads, CPU-side or GPU-side, were blocked during these dispatches; therefore, the CPU was free to perform other work, such as physics computation for the next frame. 2) Future improvements to the CUDA/GL interoperability will hopefully reduce the need for an intermediate GPU-side buffer thereby reducing time and space requirements.

In terms of framerates, Fig.3.14 shows the CUDA-based implementation tested on 3 different graphics cards. In each experiment, a ring composed of N spherical colliders approached a single mesostructure, see Fig.3.15. The plane formed by the ring was parallel to, and positioned in front of, the mesostructure. As the ring approached, all colliders simultaneously entered the mesostructure's bounding volume. At each frame, the CUDA GPU Collision Kernel, see Fig.3.2, was dispatched for each collider as it approached its collision point within the mesostructure. This continued until each of the colliders impacted, deformed, and bounced out of the mesostructure's bounding volume. The resulting framerate, averaged over 100*msec* intervals, is shown in Fig.3.14.

3.4.2 Implementation with OpenGL 4.3 Compute Shaders

The second implementation used the same code base (OpenGL for rendering, NVIDIA PhysX 3.1.1) as above with one change: all CUDA GPU Kernel invocations were replaced with their corresponding OpenGL 4.3 Compute Shader counterparts. In

Figure 3.14: CUDA performance and Compute Shader performance on 3 different graphics cards. Each collider inside the mesostructure's volume dispatches its CUDA / Compute Shader GPU Kernel each frame to account for potential changes in collider trajectory or mesostructure motion.

terms of implementation, using compute shaders simplified the initialization, dispatch, and data I/O over CUDA since OpenGL internally managed this on behalf of the user. Furthermore, since compute shaders are written in GLSL and therefore share common utility functions and syntax as well as the same compiler, the desired compute shaders were almost a copy/paste from the existing fragment shader with only minor changes. The

Figure 3.15: Test configuration showing $N = 25$ colliders simultaneously enter a mesostructure's volume, collide, and bounce back out of the intersection volume.

CUDA GPU Kernel, on the other hand, required use of the CUDA API and NVIDIA CUDA Compiler (NVCC).

In terms of performance, Fig.3.14 shows the compute shader implementation tested on 3 different graphics cards via the same methodology, as discussed at the end of section 3.4.1, with the exception that compute shader GPU kernels were dispatched in place of the

CUDA GPU kernels. All three graphics cards were able to maintain interactive rates for all plotted N . Depending on the application, it may not even be necessary to dispatch the GPU Collision Kernel for each collider at each step thereby further saving computation.

3.4.3 Scalable Link Interface (SLI)

NVIDIA graphics hardware supports using multiple GPUs in parallel to enhance the rendering capabilities of an application via a Scalable Link Interface (SLI). Placing two GPUs in SLI-mode enables the graphics driver to treat them as a single, logical device and divide the rendering work automatically. Once represented as a single, logical device, SLI supports any one of three rendering techniques: Alternate Frame Rendering (AFR), Split Frame Rendering (SFR), and SLI Antialiasing (SLIAA).

Alternate Frame Rendering theoretically allows one GPU to compute the even frames and the other to compute the odd frames. Ideally, this would double the frame rate; however, since all texture information must be replicated to each card, the mesostructure advantage that eliminated CPU-GPU data shuffling transforms to GPU-GPU data shuffling. After one video card efficiently updates a displacement map or other corresponding texture(s), that entire texture(s) must be copied to the other video card so that the next frame computed on that card will reflect the changes computed by the first card. Since these operations are handled internally by the video driver, the user has little control over how and when this occurs. NVIDIA recommends the AFR approach be used for geometry limited applications. Since interactive mesostructures are an image-based rendering technique designed to minimize geometry, they are more commonly pixel-rate limited than geometry limited. Furthermore, using AFR requires that up to three frames be buffered ahead of time, this introduces higher latency via input devices since a key press must wait for three frames to be rendered before the output is affected. With AFR enabled, there was no noticeable performance change in the performance tests shown in Fig. 3.14

[70]. However, future driver updates offer the potential to improve upon this performance and offer higher framerates for interactive mesostructures.

Split Frame Rendering divides a scene into subregions and assigns each subregion to be handled by a different GPU. The scene may be divided vertically or horizontally, and the driver can dynamically resize the subregions when it determines one GPU is doing more work than another GPU. NVIDIA states this approach benefits pixel-rate limited applications but does not divide the work load as well as AFR. This approach, depending on the specific application, may duplicate work and result in higher GPU-GPU overhead than AFR [70]. Unfortunately, the SFR approach has been deprecated and is unsupported with the latest generation of graphics APIs, Direct X 10+ and OpenGL 4.3+. This has occurred since modern APIs require support for geometric tessellation and more complex fragment shader functionality, such as rendering to multiple targets. As a result of these complexities, AFR has gained favor over SFR and SFR's support has been removed; as a result, SFR could not be tested within the interactive mesostructure's performance analysis.

Finally, SLI Antialiasing (SLIAA), is not a technique to accelerate rendering performance, but instead a hardware accelerated antialiasing technique. In this technique, each GPU generates an output frame, but performs antialias sampling in a direction perpendicular to that of the other GPU. These two resulting frames are then combined to offer improved antialiased output. SLIAA offers up to 32x antialiasing [70]. As this is not a performance enhancement, it had no impact on the interactive mesostructure performance analysis shown in Fig. 3.14.

3.5 Conclusion

We have presented a technique for interactively deforming mesostructures through dynamic collisions, animation, user interaction, etcetera. This technique also offers an

alternative per-texel approach to traditional decal insertion, placement, and manipulation. This technique may be used with existing displacement map rendering techniques, both safe and unsafe. It integrates with existing physics engines, requires no preprocessing time, minimal storage, and approximates 3D deformations (vertex-based) as 2D image operations (image-based). These image operations remain server-side, are efficiently parallelizable on modern GPUs, and offer a powerful means to dynamically manipulate almost every aspect of a mesostructure's 3D characteristics. With this technique, we hope to bring physical per-texel interactivity to mesostructures and promote worlds where almost every surface can be interactively deformed and manipulated through 2D texture operations. These operations execute concurrently, remain server-side (on the GPU), offer per-texel resolution, accumulate naturally, and persist throughout the application's life time.

4 CONCLUSION

This work presents contributions in the areas of computer graphics, visualization, and simulation, see Fig.1.1. Chapter 2 presents a general image-based method for visualizing large data sets while Chapter 3 introduces a novel image-based rendering technique called Interactive Mesostructures. Both billboards and interactive mesostructures may be used to render a complex virtual world at an interactive framerate such that a human viewer is able to observe a smooth scene that responds quickly and naturally. These techniques do not necessarily render an entire scene in a perfectly accurate manner. Instead, they trade a small amount of fidelity for an increase in performance. The goal of this trade off is to ensure the rendered output is interactive and looks ‘good enough’ to form a plausible scene when observed by a human. Billboards achieve this by replacing a subset of a scene’s complex geometry with a picture of that complex geometry. This façade has several important limitations, as discussed in Chapter 2. These limitations lead to a new form of image-based rendering called Interactive Mesostructures, which are built on the same underlying image-based approach, yet are able to address many of the billboard’s shortcomings and make image-based rendering a practical solution to modeling almost all surfaces in an interactive virtual world, see Chapter 3. These chapters present a set of novel algorithmic rendering approaches designed for massive concurrent execution on parallel hardware, exploiting modern GPU design efficiently. They offer image-based approaches for rendering as much interactive detail as possible while maintaining real-time framerates.

4.1 Image-Based Impostor Contributions

Chapter 2 uses image-based rendering techniques to replace areas of computationally expensive scenes with *image-based impostors*, or *pictures* of those areas. The generated *impostor* is then used in lieu of the original geometry. The impostor is simply a

quadrilateral (two triangles) with the picture *textured* across itself offering the illusion of the original geometry. If the user changes their position with respect to the impostor, the façade becomes apparent and the impostor must be regenerated. The computational cost of generating the impostor is amortized over the number of frames that impostor remains valid. In many circumstances the number of vertices that need to be transformed per frame can be substantially reduced using image-based impostors, see Chapter 2.

This technique is then applied to the interactive visualization of 3D Light Detection and Ranging (LIDAR) point cloud data. Using commodity laptop hardware circa 2010, tens of millions of LIDAR points were able to be visualized and updated in real-time, see Section 2.4.3. An update policy determining how frequently to update each impostor was presented. A quantitative analysis of visual, per-frame differences between the real-time approximation and the original, non-interactive data was presented. In summary, Chapter 2 contributes the following:

- An image-based impostor technique to visualize data sets containing tens of millions of exemplars on commodity hardware in real-time.
- An update policy to dynamically balance this data set’s visualization *accuracy* against *interactivity*, quantitatively.
- An image-based error metric to quantify the visual error between the impostors and the reference data set.

4.2 Interactive Mesostructure Contributions

Chapter 3 presents a novel image-based rendering technique called Interactive Mesostructures. This technique adds interactivity, animation, and deformation to image-based surfaces; it enables one to replace highly detailed geometry with significantly fewer vertices while still offering interactive, physically plausible surfaces. Interactive

Mesostructures are made possible through an image-based rendering technique known as *inverse displacement mapping*. This technique can be used to render a 2D height field, encoded as a texture, on top of planar geometry providing surface resolution at a per-texel level. These constructs are known as *mesostructures* and are similar to the image-based impostors discussed in Chapter 2 but mitigate many issues including a lack of parallax motion and a lack of self occlusion. This work extends the feature set of mesostructures by making them *animate*, *interact*, *collide*, and *deform* with other objects in a natural way. By doing so, mesostructures are able to become *interactive* with the rest of the virtual world. This technique helps to bring physical per-texel interactivity to mesostructures and promote worlds where almost every surface can be interactively deformed and manipulated through efficiently parallelizable 2D texture operations. Specifically, Chapter 3 contributes the following:

- A technique for interactively deforming and colliding with mesostructures. This technique leverages existing physics engines, such as NVIDIA PhysX, for initial collision information; it subsequently performs per-texel tangent space collisions, and deforms the affected mesostructure in parallel on the GPU without any server-client texture transfers and minimal CPU involvement.
- Each deformation processed by the GPU is able to account for collider-specific physical attributes as well as per-texel surface properties describing the behavior and/or constraints of the impacted mesostructure; therefore, a mesostructure's visual and behavioral dynamics can be tailored to other objects, animations, or time.
- These deformable mesostructure operations are compatible with many existing inverse displacement mapping rendering techniques including relief mapping, parallax occlusion mapping, and even some safe methods lending themselves to

easy integration into existing game pipelines and applications that use GPU-based inverse displacement mapping techniques.

- Replaces 3D vertex-based deformations with 2D image-based operations performed solely on the GPU, in parallel, and free from CPU-GPU data shuffling.
- Provides a per-textel alternative to traditional decals. This alternative offers enhanced visual cues, requires no additional geometry, and persists for the life-time of the application.

REFERENCES

- [1] D. P. Luebke, "A developer's survey of polygonal simplification algorithms," *Computer Graphics and Applications, IEEE*, vol. 21, no. 3, pp. 24–35, 2001.
- [2] S. H. Ahn. OpenGL projection matrix: Perspective projection. [Online]. Available: http://www.songho.ca/opengl/gl_projectionmatrix.html
- [3] M. M. Oliveira and F. Policarpo, "An efficient representation for surface details," *Instituto de Informatica UFRGS*, 2005.
- [4] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," *Commun. ACM*, vol. 19, pp. 547–554, October 1976. [Online]. Available: <http://doi.acm.org/10.1145/360349.360354>
- [5] D. Luebke and C. Erikson, "View-dependent simplification of arbitrary polygonal environments," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 199–208. [Online]. Available: <http://dx.doi.org/10.1145/258734.258847>
- [6] D. Luebke and B. Hallen, "Perceptually driven simplification for interactive rendering," in *Rendering Techniques 2001*. Springer, 2001, pp. 223–234.
- [7] D. Luebke, *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.
- [8] G. Schaufler, "Dynamically generated impostors," *Proc. GI Workshop on Modeling, Virtual Worlds, and Distributed Graphics*, pp. 129–136, 1995.
- [9] T. Funkhouser and C. Séquin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 247–254.
- [10] P. W. C. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, ser. I3D '95. New York, NY, USA: ACM, 1995, pp. 95–ff. [Online]. Available: <http://doi.acm.org/10.1145/199404.199420>
- [11] D. G. Aliaga, "Visualization of complex models using dynamic texture-based simplification," in *Proceedings of the 7th conference on Visualization '96*, ser. VIS '96. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996, pp. 101–ff. [Online]. Available: <http://portal.acm.org/citation.cfm?id=244979.245020>
- [12] D. Aliaga and A. Lastra, "Smooth transitions in texture-based simplification," *Computers + Graphics*, vol. 22, no. 1, pp. 71 – 81, 1998. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0097849397000848>

- [13] D. G. Aliaga and A. Lastra, "Automatic image placement to provide a guaranteed frame rate," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 307–316. [Online]. Available: <http://dx.doi.org/10.1145/311535.311574>
- [14] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder, "Hierarchical image caching for accelerated walkthroughs of complex environments," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 75–82. [Online]. Available: <http://doi.acm.org/10.1145/237170.237209>
- [15] A. Meyer and F. Neyret, "Interactive volumetric textures," in *Rendering techniques '98: proceedings of the Eurographics Workshop in Vienna, Austria, June 29-July 1, 1998*. Springer Verlag Wien, 1998, p. 157.
- [16] G. Schaufler, "Image-based object representation by layered impostors," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '98. New York, NY, USA: ACM, 1998, pp. 99–104. [Online]. Available: <http://doi.acm.org/10.1145/293701.293714>
- [17] G. Schaufler, "Per-object image warping with layered impostors," in *Rendering Techniques 98*. Springer, 1998, pp. 145–156.
- [18] A. Jakulin, "Interactive vegetation rendering with slicing and blending," in *Proc. Eurographics*, vol. 2000. Citeseer, 2000.
- [19] S. Jeschke, M. Wimmer, and H. Schumann, "Layered environment-map impostors for arbitrary scenes," in *Graphics Interface*. Citeseer, 2002, pp. 1–8.
- [20] L. Darsa, B. Costa Silva, and A. Varshney, "Navigating static environments using image-space simplification and morphing," in *Proceedings of the 1997 symposium on Interactive 3D graphics*, ser. I3D '97. New York, NY, USA: ACM, 1997, pp. 25–ff. [Online]. Available: <http://doi.acm.org/10.1145/253284.253298>
- [21] F. Sillion, G. Drettakis, and B. Bodelet, "Efficient impostor manipulation for real-time visualization of urban scenery," in *Computer Graphics Forum*, vol. 16. Wiley Online Library, 1997, pp. C207–C218.
- [22] L. Darsa, B. Costa, and A. Varshney, "Walkthroughs of complex environments using image-based simplification," *Computers + Graphics*, vol. 22, no. 1, pp. 55 – 69, 1998. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0097849397000836>
- [23] L. Darsa and B. Costa, "Multiresolution representation and reconstruction of adaptively sampled images," in *IN SIBGRAPI*, 1996, pp. 321–328.

- [24] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, R. Bastos, M. Whitton, F. Brooks, and D. Manocha, "Mmr: an interactive massive model rendering system using geometric and image-based acceleration," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, ser. I3D '99. New York, NY, USA: ACM, 1999, pp. 199–206. [Online]. Available: <http://doi.acm.org/10.1145/300523.300554>
- [25] S. Jeschke and M. Wimmer, "Textured depth meshes for real-time rendering of arbitrary scenes," in *Proceedings of the 13th Eurographics workshop on Rendering*. Citeseer, 2002, pp. 181–190.
- [26] A. Wilson and D. Manocha, "Simplifying complex environments using incremental textured depth meshes," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 678–688. [Online]. Available: <http://doi.acm.org/10.1145/1201775.882325>
- [27] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey, "Multi-layered impostors for accelerated rendering," in *Computer Graphics Forum*, vol. 18. Wiley Online Library, 1999, pp. 61–73.
- [28] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1*, 6th ed. Boston, MA, USA: Addison-Wesley Publishing Co., Inc., 2008.
- [29] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf
- [30] J. Campbell, M. de Haag, F. van Graas, and S. Young, "Light detection and ranging-based terrain navigation-a concept exploration," in *Proceedings from the*. Citeseer, 2003.
- [31] Z. Bowen and R. Waltermire, "EVALUATION OF LIGHT DETECTION AND RANGING (LIDAR) FOR MEASURING RIVER CORRIDOR TOPOGRAPHY1," *JAWRA Journal of the American Water Resources Association*, vol. 38, no. 1, pp. 33–41, 2002.
- [32] C. Hasager, A. Peña, T. Mikkelsen, M. Courtney, I. Antoniou, S. Gryning, P. Hansen, and P. Sørensen, "12MW Horns Rev experiment," Risø National Laboratory, Tech. Rep., 2007.
- [33] J. Hewett, "Optics.org," <http://optics.org/article/34878>, July 2008.

- [34] J. McKinion, J. Willers, and J. Jenkins, "Spatial analyses to evaluate multi-crop yield stability for a field," *Computers and Electronics in Agriculture*, vol. 70, no. 1, pp. 187–198, 2010.
- [35] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann *et al.*, "Stanley: The robot that won the DARPA Grand Challenge," *The 2005 DARPA Grand Challenge*, pp. 1–43, 2007.
- [36] P. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM, 1995, pp. 95–ff.
- [37] O. LAWLOR, "IMPOSTORS FOR PARALLEL INTERACTIVE COMPUTER GRAPHICS," Ph.D. dissertation, University of Illinois, 2005.
- [38] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi, "Detailed shape representation with parallax mapping," in *Proceedings of ICAT*, vol. 2001. Citeseer, 2001, pp. 205–208.
- [39] N. Tatarchuk, "Dynamic parallax occlusion mapping with approximate soft shadows," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 63–69.
- [40] S. Nykl, C. Mourning, and D. Chelberg, "Interactive mesostructures," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2013, pp. 37–44.
- [41] I. T. Union, "Recommendation ITU-R BT.601-7," International Telecommunication Union, Tech. Rep., 2011.
- [42] A. Haar, "Zur theorie der orthogonalen funktionensysteme," *Mathematische Annalen*, vol. 69, no. 3, pp. 331–371, 1910.
- [43] Y. Lai and C. Kuo, "A haar wavelet approach to compressed image quality measurement," *Journal of Visual Communication and Image Representation*, vol. 11, no. 1, pp. 17–40, 2000.
- [44] E. Catmull, "A subdivision algorithm for computer display of curved surfaces." DTIC Document, Tech. Rep., 1974.
- [45] J. F. Blinn, "Simulation of wrinkled surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, pp. 286–292, August 1978. [Online]. Available: <http://doi.acm.org/10.1145/965139.507101>
- [46] R. L. Cook, "Shade trees," in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '84. New York, NY, USA: ACM, 1984, pp. 223–231. [Online]. Available: <http://doi.acm.org/10.1145/800031.808602>

- [47] L. Szirmay-Kalos and T. Umenhoffer, "Displacement mapping on the gpu state of the art," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1567–1592, 2008. [Online]. Available: <http://dx.doi.org/10.1111/j.1467-8659.2007.01108.x>
- [48] F. Policarpo, M. Oliveira, and J. Comba, "Real-time relief mapping on arbitrary polygonal surfaces," in *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. ACM, 2005, pp. 155–162.
- [49] A. Tevs, I. Ihrke, and H. Seidel, "Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. ACM, 2008, pp. 183–190.
- [50] J. Patterson, S. Hoggar, and J. Logie, "Inverse displacement mapping," *Computer Graphics Forum*, vol. 10, no. 2, pp. 129–139, 1991. [Online]. Available: <http://dx.doi.org/10.1111/1467-8659.1020129>
- [51] M. M. Oliveira, G. Bishop, and D. McAllister, "Relief texture mapping," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 359–368. [Online]. Available: <http://dx.doi.org/10.1145/344779.344947>
- [52] L. McMillan Jr, "An image-based approach to three-dimensional computer graphics," Ph.D. dissertation, University of North Carolina, 1997.
- [53] M. Peercy, J. Airey, and B. Cabral, "Efficient bump mapping hardware," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 303–306. [Online]. Available: <http://dx.doi.org/10.1145/258734.258873>
- [54] F. Policarpo and M. Oliveira, "Relief mapping of non-height-field surface details," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 55–62.
- [55] E. Risser, "True imposters," in *ACM SIGGRAPH 2006 Research posters*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006. [Online]. Available: <http://doi.acm.org/10.1145/1179622.1179688>
- [56] W. Hu and K. Qin, "Interactive approximate rendering of reflections, refractions, and caustics," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 1, pp. 46–57, 2007.
- [57] C. Wyman, "Interactive image-space refraction of nearby geometry," in *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM, 2005, pp. 205–211.

- [58] L. Baboud and X. Décoret, “Rendering geometry with relief textures,” in *Proceedings of Graphics Interface 2006*, ser. GI ’06. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2006, pp. 195–201. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1143079.1143112>
- [59] S. Jeschke, S. Mantler, and M. Wimmer, “Interactive smooth and curved shell mapping,” *Proceedings of EGSR07*, pp. 351–360, 2007.
- [60] W. Donnelly, “Per-pixel displacement mapping with distance functions,” *GPU gems*, vol. 2, no. 22, p. 3, 2005.
- [61] F. Policarpo and M. Oliveira, “Relaxed cone stepping for relief mapping,” *GPU gems*, vol. 3, pp. 409–428, 2007.
- [62] K. Oh, H. Ki, and C. Lee, “Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid,” in *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM, 2006, pp. 75–82.
- [63] C. Ericson, *Real-time collision detection*. Morgan Kaufmann, 2004.
- [64] A. Kolb and C. Rezk-Salama, “Efficient empty space skipping for per-pixel displacement mapping,” in *Proc. Vision, Modeling and Visualization*, 2005, pp. 407–414.
- [65] J. Dummer, “Cone step mapping,” <http://www.lonesock.net/files/ConeStepMapping.pdf>, 2006.
- [66] J. Tessendorf *et al.*, “Simulating ocean water,” *Simulating Nature: Realistic and Interactive Techniques. SIGGRAPH*, 2001.
- [67] L. Shapiro and G. Stockman, “Computer vision. 2001,” 2001.
- [68] A. H. Watt and F. Policarpo, *Advanced game development with programmable graphics hardware*. AK Peters, 2005.
- [69] S. Petitjean, “A survey of methods for recovering quadrics in triangle meshes,” *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 211–262, 2002.
- [70] *SLI Best Practices*, http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf, 2011. [Online]. Available: http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf

APPENDIX: INTERACTIVE MESOSTRUCTURE IMPLEMENTATIONS

A.1 A Compute Shader for Planar Interactive Mesostructures

```
#version 430 compatibility

//An example OpenGL 4.3 Compute Shader used with Interactive Mesostructures
//Based off of Policarpo's "Real-Time Relief Mapping on Arbitrary Polygonal
//Surfaces". This shader is for planar surfaces and does not include the
//curved surface implementation.

layout( rgba8 ) coherent uniform image2D nrmMap;
layout( local_size_x = 16, local_size_y = 16 ) in;

struct CSOutputBuffer
{
    //output values for a ray intersection against the mesostructure
    vec2 uvOffset; ///< texture space offset for parallax motion
    float hypotLen; ///< Length of the hypotenuse scaled to fit the geometry
    float depth; ///< normalized depth in meso, computed in rayIntersectDepthMap
    vec4 normal; ///< model space collision normal at point of collision
    vec4 xyzDispTS; ///< Tangent space displacement vector from original
};                                     ///< polygonal intersection to offset

struct CSInputBuffer
{
    //ray cast variables
    vec4 startUV; ///< tex coord of the (UV) where ray cast begins
    vec4 tanDir; ///< tangent space vector in direction of collider's trajectory
    vec2 curvXY; ///< per vertex interpolated curvature corresponding to startUV

    //spherical / linear sine wave deformations
    vec2 center; ///< 0,0 is bottom left, negative is invalid for deformation
}
```

```

float radius; ///< radius of deformation
float deformScalar; //amount to extrude / deform by
float phase; ///< used in linear sine wave, determines speed of waves

//control parameter determines which kernel is executed upon entering main()
unsigned int flag; //0=spherical deform,1=normal map generation,
                  //2=radial sine wave,3=rayCastReturnCollisionInfo
};                  //4=setHeightFieldToUniformHeight

layout( std430, binding=1 ) buffer OutData
{
    CSOutputBuffer buf;
} outDat;

layout( std430, binding=5 ) buffer InData
{
    CSInputBuffer buf;
} inDat;

void deformSphereExtrusion();
void deformRadialSineWave();
void generateNormalMapFromDispMap();
void rayCastReturnCollisionInfo();
void deformSphereIntrusion();
float rayIntersectDepthMap( vec2 p, vec2 ds, uvec2 imageSize );
void setHeightFieldToUniformHeight();

void main()
{
    if( inDat.buf.flag == 0 ) deformSphereExtrusion();
    else if( inDat.buf.flag == 1 ) generateNormalMapFromDispMap();

```

```

else if( inDat.buf.flag == 2 ) deformRadialSineWave();
else if( inDat.buf.flag == 3 ) rayCastReturnCollisionInfo();
else if( inDat.buf.flag == 4 ) setHeightFieldToUniformHeight();
}

void setHeightFieldToUniformHeight()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );
    if( pos.x < size.x && pos.y < size.y )
    {
        vec4 texel = imageLoad( nrmMap, pos );
        float h = inDat.buf.deformScalar;
        texel.a = h;
        imageStore(nrmMap, pos, texel );
    }
}

void deformSphereExtrusion()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    vec2 p = vec2( float(pos.x) / float(size.x), float(pos.y) / float(size.y) );
    vec2 c = inDat.buf.center; //floating point origin of sphere extrusion

    float r = inDat.buf.radius;
    float rsq = r * r;
    float dsX = (p.x - c.x) * (p.x - c.x);
    float dsY = (p.y - c.y) * (p.y - c.y);

```

```

if( dsX + dsY <= rsq )
{
    vec4 texel = imageLoad( nrmMap, pos );
    float h = texel.a;
    h += sqrt( rsq - dsX - dsY ) * inDat.buf.deformScalar;
    texel.a = h;
    imageStore(nrmMap, pos, texel );
}
}

void deformRadialSineWave()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    vec2 p = vec2( float(pos.x) / float(size.x), float(pos.y) / float(size.y) );
    vec2 c = inDat.buf.center; //floating point origin of sphere extrusion

    float distX = p.x - inDat.buf.center.x;
    float distY = p.y - inDat.buf.center.y;

    if( pos.x >= size.x || pos.y >= size.y )
        return;

    float dX = distX * distX;
    float dY = distY * distY;
    float dist = sqrt( dX + dY );

    const float waveLength = 0.25; // 1/4 of image width is 2PI rads
    const float TWOPI = 2.0 * 3.14159265359;
    float input = ((dist + inDat.buf.phase) / waveLength) * TWOPI;

```

```

float disp = sin( input );
disp *= 0.03; //decrease amplitude by half

vec4 texel = imageLoad( nrmMap, pos );
float height = texel.a;
height += disp;
height = clamp( height, 0.05, 1.0 ); //eliminate holes from forming
imageStore( nrmMap, pos, vec4( texel.xyz, height ) );
}

void generateNormalMapFromDispMap()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );
    if( pos.x >= size.x || pos.y >= size.y )
        return;
    vec4 texel = imageLoad( nrmMap, pos );
    int x = pos.x; int y = pos.y; int dv = 1; int du = 1; float scale = 0.25;
    int sp = 2; //secondary pixel offset to weight with primary pixel weight
    //sample each neighbor and extended neighbor's height to compute gradient
    int hLt = x; int hRt = x; int hLt2 = x; int hRt2 = x;
    int hUp = y; int hDn = y; int hUp2 = y; int hDn2 = y;
    //clamp to avoid going beyond the edges in V direction
    if( (hUp + dv) < size.y ) hUp += dv;
    if( (hDn - dv) >= 0 ) hDn -= dv;
    if( (hUp2 + dv + sp) < size.y ) hUp2 += dv + sp;
    if( (hDn2 - dv - sp) >= 0 ) hDn2 -= dv - sp;
    //clamp to avoid going beyond the edges in U direction
    if( (hLt - du) >= 0 ) hLt -= du;
    if( (hRt + du) < size.x ) hRt += du;
    if( (hLt2 - du - sp) >= 0 ) hLt2 -= du - sp;

```

```

    if( (hRt2 + du + sp) < size.x ) hRt2 += du + sp;

    //now read the displacement map values from the image2D
    //imageLoad( nrmMap, ivec2( 174, 156 ) );

    float me      = imageLoad(nrmMap, ivec2( x, y ) ).a * 255.0;
    float left     = imageLoad(nrmMap, ivec2( hLt, y ) ).a * 255.0;
    float right    = imageLoad(nrmMap, ivec2( hRt, y ) ).a * 255.0;
    float left2    = imageLoad(nrmMap, ivec2( hLt2, y ) ).a * 255.0;
    float right2   = imageLoad(nrmMap, ivec2( hRt2, y ) ).a * 255.0;
    float top      = imageLoad(nrmMap, ivec2( x, hUp ) ).a * 255.0;
    float bot      = imageLoad(nrmMap, ivec2( x, hDn ) ).a * 255.0;
    float top2     = imageLoad(nrmMap, ivec2( x, hUp2 ) ).a * 255.0;
    float bot2     = imageLoad(nrmMap, ivec2( x, hDn2 ) ).a * 255.0;
    float dhdv = (( top - (bot) ) * 0.6f ) + ( (top2) - (bot2) ) * 0.4f;
    float dhdu = (( right - (left) ) * 0.6f ) + ( (right2) - (left2) ) * 0.4f;
    dhdv *= -1.0; dhdv = scale / ( 2.0 * float(dv) ) * dhdv;
    dhdu *= -1.0; dhdu = scale / ( 2.0 * float(du) ) * dhdu;
    vec3 norm = vec3( dhdu, dhdv, 1.0 );
    norm = normalize( norm );
    vec3 intVer = vec3( int(127.5*(norm.x+1.0)),
                       int(127.5*(norm.y+1.0)),
                       int(127.5*(norm.z+1.0)) );

    intVer /= 255;
    texel = vec4( intVer, texel.a );
    imageStore(nrmMap, pos, texel );
}

void rayCastReturnCollisionInfo()
{
    const float DEPTH = 0.1;
    if( gl_GlobalInvocationID.x == 0 && gl_GlobalInvocationID.y == 0 )
    {

```



```

//this is the main thread; we just compute the answer in one thread
uvec2 imageSize = imageSize( nrmMap );
vec3 stepDir = vec3( inDat.buf.tanDir );
stepDir.z *= -1.0;
vec2 ds = vec2( stepDir.x * DEPTH / stepDir.z,
                stepDir.y * DEPTH / stepDir.z );
float depth = rayIntersectDepthMap(inDat.buf.startUV.xy, ds, imageSize);
vec2 uvPrime = inDat.buf.startUV.xy + ds * depth; //shifted tex coord
//compute the magnitude to scale 'stepDir' by to compute precise location
//'10.0' below is the scale factor mapping the face's object space to
//texture space this may optionally be passed into the shader
float aa = sqrt( ( dot( ds*depth*10.0, ds*depth*10.0 ) ) );
float bb = depth * DEPTH * 10.0;
float cc = sqrt( aa*aa + bb*bb );
outDat.buf.uvOffset = uvPrime;
outDat.buf.hypotLen = cc;
outDat.buf.depth = depth;
//get contact normal at uvPrime
vec4 texel = imageLoad( nrmMap,
                        ivec2( int( uvPrime.x * imageSize.x + 0.5 ),
                              int( uvPrime.y * imageSize.y + 0.5 ) ) );
//unpack vec3 normal
float nx = 2.0 * texel.x - 1; //unpack from (unsigned)RGB to (signed)XYZ
float ny = 2.0 * texel.y - 1;
float nz = 2.0 * texel.z - 1;
outDat.buf.normal = vec4(nx,ny,nz,0);
}
}

// Assumes depth map is in alpha channel of image2D in the range [0,1].
// Assumes depth map is stored as a "height map" (whiter areas are 'higher')

```

```

float rayIntersectDepthMap( vec2 p, vec2 ds, uvec2 imageSize )
{
    const int linear_search_steps = 30;
    const int binary_search_steps = 5;
    float depth_step = 1.0 / linear_search_steps;
    float size = depth_step; // current size of search window
    float depth = 0.0; // current depth position
    // best match found thus far (starts with last position 1.0)
    float best_depth = 1.0; //first point below the surface

    vec2 uv;
    for( int i = 0; i < linear_search_steps - 1; i++ )
    {
        depth += size;
        uv = p + ds * depth;
        vec4 charW = imageLoad( nrmMap,
                                ivec2( int( uv.x * imageSize.x + 0.5 ),
                                         int( uv.y * imageSize.y + 0.5 ) ) );
        float w = charW.a; //compute shader already returns value normalized [0,1]

        w = 1.0 - w; //If using a "height map", then we must do a 1 - w to get Z
        if( best_depth > 0.996 ) // if no depth found yet
            if( depth >= w )
                best_depth = depth; // store best depth
    }
    depth = best_depth;
    //// search around first point (depth) for closest match
    for( int i = 0; i < binary_search_steps; i++ )
    {
        size *= 0.5;
        uv = p + ds * depth;
    }
}

```

```

vec4 charW = imageLoad( nrmMap, ivec2( int( uv.x * imageSize.x + 0.5 ),
                                         int( uv.y * imageSize.y + 0.5 )));

float w = charW.a; //compute shader already returns value normalized [0,1]
w = 1 - w;
if (depth >= w)
{
    best_depth = depth;
    depth -= 2 * size;
}
depth += size;
}
return best_depth;
}

```

A.2 A Compute Shader for Curved Interactive Mesostructures

```
//An example OpenGL 4.3 Compute Shader used with Curved Interactive
//Mesostructures.
//Based off Oliveira and Policarpo's "An Efficient Representation for Surface Details".
//This shader has been extended to compute and return contact points within the
//mesostructure. It may be used with curved surfaces.
```

```
#version 430 compatibility
```

```
layout( rgba8 ) coherent uniform image2D nrmMap;
layout( local_size_x = 16, local_size_y = 16 ) in;
```

```
struct CSOutputBuffer
```

```
{
    //output values for a ray intersection against the mesostructure
    vec2 uvOffset; ///< texture space offset for parallax motion
    float hypotLen; ///< Length of the hypotenuse scaled to fit the
                    ///< macrostructure's geometry
    float depth; ///< normalized parametric depth into mesostructure
                ///< as computed in rayIntersectDepthMap
    vec4 normal; ///< model space collision normal at point of collision
    vec4 xyzDispTS; ///< Tangent space displacement vector from original
                    ///< polygonal intersection to offset
};
```

```
struct CSInputBuffer
```

```
{
    //ray cast variables
    vec4 startUV; ///< texture coordinate of the initial tex coord (UV)
                ///< where relief map begins
};
```

```

vec4 tanDir; ///< tangent space vector in direction of collider's trajectory
vec2 curvXY; ///< per vertex interpolated curvature corresponding to startUV
vec2 uvScale; ///< per vertex interpolated uvScale values corresponding to startUV

//spherical / linear sine wave deformations
vec2 center; ///< 0,0 is bottom left, negative is invalid for deformation
float radius; ///< radius of deformation
float deformScalar;
float phase; ///< used in linear sine wave, determines speed of waves

//control parameter determines which kernel is executed upon entering main()
unsigned int flag; ///< 0=spherical deform,1=normal map generation,
                    ///< 2=radial sine wave,3=rayCastReturnCollisionInfo
};

layout( std430, binding=1 ) buffer OutData
{
    CSOutputBuffer buf;
} outDat;

layout( std430, binding=5 ) buffer InData
{
    CSInputBuffer buf;
} inDat;

//layout( std430, binding=5 ) buffer MyData
//{
//    MyDataStruct vals[];
//} myDat;

//these 3 methods do an "apples to apples" cuda to compute shader

```

```

//comparison
void deformSphereExtrusion();
void deformRadialSineWave();
void generateNormalMapFromDispMap();
void generateNormalMapFromDispMapSobel();
void rayCastReturnCollisionInfo();
void deformSphereIntrusion();
float rayIntersectDepthMap( vec2 p, vec2 ds, uvec2 imageSize );
void rayCastReturnCollisionCurvedInfo();
float rayIntersectDepthMapCurved( vec2 tx, vec3 v, float tmax,
                                   float dataz, uvec2 imageSize );
vec3 rayPositionCurved( float t, vec2 tx, vec3 v, float dataz );
vec3 rayDisplacementCurved( float t, vec3 v, float dataz );

void setHeightFieldToUniformHeight();

void main()
{
    if(      inDat.buf.flag == 0 )
        deformSphereExtrusion();
    else if( inDat.buf.flag == 1 )
        generateNormalMapFromDispMap();
    else if( inDat.buf.flag == 2 )
        deformRadialSineWave();
    else if( inDat.buf.flag == 3 )
        rayCastReturnCollisionInfo();
    else if( inDat.buf.flag == 4 )
        rayCastReturnCollisionCurvedInfo();
    else if( inDat.buf.flag == 5 )
        setHeightFieldToUniformHeight();
}

```

```

    return;
}

void setHeightFieldToUniformHeight()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    if( pos.x < size.x && pos.y < size.y )
    {
        vec4 texel = imageLoad( nrmMap, pos );
        float h = inDat.buf.deformScalar;
        texel.a = h;
        imageStore(nrmMap, pos, texel );
    }
}

void deformSphereExtrusion()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    //floating point pos
    vec2 p = vec2( float(pos.x) / float(size.x), float(pos.y) / float(size.y) );
    vec2 c = inDat.buf.center; //floating point origin of sphere extrusion

    float r = inDat.buf.radius;
    float rsq = r * r;
    float dsX = (p.x - c.x) * (p.x - c.x);
    float dsY = (p.y - c.y) * (p.y - c.y);

```

```

if( dsX + dsY <= rsq )
{
    vec4 texel = imageLoad( nrmMap, pos );
    float h = texel.a;

    h += sqrt( rsq - dsX - dsY ) * inDat.buf.deformScalar;

    texel.a = h;

    imageStore(nrmMap, pos, texel );
    //generateNormalMapFromDispMap( pos, size, texel );
    //generate new normal map only for affected pixels
}
}

void deformRadialSineWave()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    //floating point pos
    vec2 p = vec2( float(pos.x) / float(size.x), float(pos.y) / float(size.y) );
    vec2 c = inDat.buf.center; //floating point origin of sphere extrusion

    float distX = p.x - inDat.buf.center.x;
    float distY = p.y - inDat.buf.center.y;

    if( pos.x >= size.x || pos.y >= size.y )
        return;

    float dX = distX * distX;

```



```

float dY = distY * distY;
//float rsq =radius * (float)radius;

float dist = sqrt( dX + dY );

const float waveLength = 0.25; // 1/4 of image width (on axis) is
                                //one full wavelength (2PI rads)
const float TWOPI = 2.0 * 3.14159265359;
float input = ((dist + inDat.buf.phase) / waveLength) * TWOPI;

float disp = sin( input );
disp *= 0.03; //decrease amplitude by half

vec4 texel = imageLoad( nrmMap, pos );
float height = texel.a;
height += disp;

height = clamp( height, 0.05, 1.0 ); //eliminate holes from forming

imageStore( nrmMap, pos, vec4( texel.xyz, height ) );
}

void generateNormalMapFromDispMap() // ivec2 pos, uvec2 size, vec4 texel )
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    if( pos.x >= size.x || pos.y >= size.y )
        return;

    vec4 texel = imageLoad( nrmMap, pos );

```

```

int x = pos.x;
int y = pos.y;
int dv = 1;
int du = 1;
float scale = 0.25;

int sp = 2; //secondary pixel offset to weight averagely with

int hLt = x; //height on left of this pixel
int hRt = x; //height on right of this pixel

int hLt2 = x; //height on left of this pixel
int hRt2 = x; //height on right of this pixel

int hUp = y; //height on top of this pixel
int hDn = y; //height on bottom of this pixel

int hUp2 = y; //height on top of this pixel
int hDn2 = y; //height on bottom of this pixel

//clamp to avoid going beyond the edges in V direction
if( (hUp + dv) < size.y )
    hUp += dv;
if( (hDn - dv) >= 0 )
    hDn -= dv;

if( (hUp2 + dv + sp) < size.y )
    hUp2 += dv + sp;
if( (hDn2 - dv - sp) >= 0 )
    hDn2 -= dv - sp;

```

```

//clamp to avoid going beyond the edges in U direction
if( (hLt - du) >= 0 )
    hLt -= du;
if( (hRt + du) < size.x )
    hRt += du;

if( (hLt2 - du - sp) >= 0 )
    hLt2 -= du - sp;
if( (hRt2 + du + sp) < size.x )
    hRt2 += du + sp;

//now read the displacement map values from the image2D
//imageLoad( nrmMap, ivec2( 174, 156 ) );
float me      = imageLoad(nrmMap, ivec2( x, y ) ).a * 255.0;
float left    = imageLoad(nrmMap, ivec2( hLt, y ) ).a * 255.0;
float right   = imageLoad(nrmMap, ivec2( hRt, y ) ).a * 255.0;
float left2   = imageLoad(nrmMap, ivec2( hLt2, y ) ).a * 255.0;
float right2  = imageLoad(nrmMap, ivec2( hRt2, y ) ).a * 255.0;
float top     = imageLoad(nrmMap, ivec2( x, hUp ) ).a * 255.0;
float bot     = imageLoad(nrmMap, ivec2( x, hDn ) ).a * 255.0;
float top2    = imageLoad(nrmMap, ivec2( x, hUp2 ) ).a * 255.0;
float bot2    = imageLoad(nrmMap, ivec2( x, hDn2 ) ).a * 255.0;

float dhdv = (( top - (bot) ) * 0.6f ) + ( (top2) - (bot2) ) * 0.4f;
float dhdu = (( right - (left) ) * 0.6f ) + ( (right2) - (left2) ) * 0.4f;

dhdv *= -1.0;
dhdu *= -1.0;

```

```

dhdv = scale / ( 2.0 * float(dv) ) * dhdv;
dhdu = scale / ( 2.0 * float(du) ) * dhdu;

//float dZ = 0.25 * sqrt( 1.0 - dhdu * dhdu - dhdv * dhdv );

vec3 norm = vec3( dhdu, dhdv, 1.0 );
norm = normalize( norm );
vec3 intVer = vec3( int(127.5*(norm.x+1.0)),
                    int(127.5*(norm.y+1.0)),
                    int(127.5*(norm.z+1.0)));
intVer /= 255;

texel = vec4( intVer, texel.a );

imageStore(nrmMap, pos, texel );
}

void generateNormalMapFromDispMapSobel()
{
    uvec2 size = imageSize( nrmMap );
    ivec2 pos = ivec2( gl_GlobalInvocationID.xy );

    int x = pos.x;
    int y = pos.y;

    int du = 1;
    int dv = 1;
    float scale = 0.25;

    if( x >= size.x || y >= size.y/2 )
        return;

```

```

vec4 texel = imageLoad( nrmMap, pos );

//left
float tl = imageLoad( nrmMap, ivec2(x - du, y + dv)).a;
float l  = imageLoad( nrmMap, ivec2(x - du,      y)).a;
float bl = imageLoad( nrmMap, ivec2(x - du, y - dv)).a;

//middle
float t = imageLoad( nrmMap, ivec2(x,      y + dv)).a;
float me = imageLoad( nrmMap, ivec2(x,      y)).a;
float b  = imageLoad( nrmMap, ivec2(x,      y - dv)).a;

//right
float tr = imageLoad( nrmMap, ivec2(x + du, y + dv)).a;
float r  = imageLoad( nrmMap, ivec2(x + du,      y)).a;
float br = imageLoad( nrmMap, ivec2(x + du, y - dv)).a;

// Compute dx using Sobel:
//          1 0 -1
//          2 0 -2
//          1 0 -1
//float dX = tr + 2.0 * r + br - tl - 2.0 * l - bl;
float dX = 1.0 * tl + -1.0 * tr +
          2.0 * l + -2.0 * r +
          1.0 * bl + -1.0 * br;

// Compute dy using Sobel:
//          -1 -2 -1
//          0  0  0
//          1  2  1

```

```

//float dY = bl + 2.0 * b + br - tl - 2.0 * t - tr;
float dY = -1.0 * tl + -2.0 * t + -1.0 * tr +
           1.0 * bl + 2.0 * b + 1.0 * br;

// Generate the missing Z component - tangent
// space normals are +Z which makes things easier
// The 0.5f leading coefficient can be used to control
// how pronounced the bumps are - less than 1.0 enhances
// and greater than 1.0 smooths.
float dZ = 0.25 * sqrt( 1.0 - dX * dX - dY * dY );

// Build the normalized normal
vec3 norm = vec3( 2.0 * dX, 2.0 * dY, dZ );
norm = normalize( norm );

vec3 intVer = vec3( int(127.5*(norm.x+1.0)),
                   int(127.5*(norm.y+1.0)),
                   int(127.5*(norm.z+1.0)) );

intVer /= 255;

texel = vec4( intVer, me );//packSnorm4x8

memoryBarrierImage();
imageStore(nrmMap, pos, texel );
}

void rayCastReturnCollisionInfo()
{
    const float DEPTH = 0.1;

    if( gl_GlobalInvocationID.x == 0 && gl_GlobalInvocationID.y == 0 )

```

```

{
    //this is the main thread; we just compute the answer in one thread
    uvec2 imageSize = imageSize( nrmMap );

    vec3 stepDir = vec3( inDat.buf.tanDir );
    stepDir.z *= -1.0;
    vec2 ds = vec2( stepDir.x * DEPTH / stepDir.z,
                    stepDir.y * DEPTH / stepDir.z );

    float depth = rayIntersectDepthMap( inDat.buf.startUV.xy, ds, imageSize );

    vec2 uvPrime = inDat.buf.startUV.xy + ds * depth; //gets populated with
                                                    //the parallax shifted texture coordinate

    //compute the magnitude to scale 'stepDir' by to compute precise location
    //(when displaced3DPos = 'stepDir' * scale + surfaceContactPt)
    float aa = sqrt( ( dot( ds*depth*10.0, ds*depth*10.0 ) ) );
    float bb = depth * DEPTH * 10.0;
    float cc = sqrt( aa*aa + bb*bb );

    //populate the output buffer with the results
    //vec2 uvOffset; ///< texture space offset for parallax motion
    //float hypotLen; ///< Length of the hypotenuse scaled to fit the
                    ///< macrostructure's geometry
    //float depth; ///< normalized parametric depth into mesostructure as
                    ///< computed in rayIntersectDepthMap
    //vec3 normal; ///< model space collision normal at point of collision
    outDat.buf.uvOffset = uvPrime;
    outDat.buf.hypotLen = cc;
    outDat.buf.depth = depth;

```

```

//get contact normal at uvPrime
vec4 texel = imageLoad( nrmMap, ivec2( int( uvPrime.x * imageSize.x + 0.5 ),
                                         int( uvPrime.y * imageSize.y + 0.5 )));

//unpack vec3 normal
float nx = 2.0 * texel.x - 1; //unpack from (unsigned)RGB to (signed)XYZ
float ny = 2.0 * texel.y - 1;
float nz = 2.0 * texel.z - 1;

outDat.buf.normal = vec4(nx,ny,nz,0);
}
}

// Assumes the depth map is stored in the alpha channel of the
//image2D in the range [0,1].
float rayIntersectDepthMap( vec2 p, vec2 ds, uvec2 imageSize )
{
    const int linear_search_steps = 30;
    const int binary_search_steps = 5;
    float depth_step = 1.0 / linear_search_steps;
    float size = depth_step; // current size of search window
    float depth = 0.0; // current depth position (more like the current parametric
                        //value along ds, 0 is point A, 1 is point B)

    // best match found thus far (starts with last position 1.0)
    float best_depth = 1.0; //first point below the surface
    //search from front to back for first point inside the object
    //  this for loop performs the linear march along vector A->B which
    //  finds the first linearly spaced point under the height field this
    //  result depth is used as the anchor in the binary search

```



```

vec2 uv;
for( int i = 0; i < linear_search_steps - 1; i++ )
{
    depth += size;

    uv = p + ds * depth;
    vec4 charW = imageLoad( nrmMap, ivec2( int( uv.x * imageSize.x + 0.5 ),
                                             int( uv.y * imageSize.y + 0.5 ) ) );
    float w = charW.a; //compute shader already returns value normalized [0,1]

    w = 1.0 - w; //If using a "height map", then we must do a 1 - w to treat
                 //value as a "depth map"
    if( best_depth > 0.996 ) // if no depth found yet
        if( depth >= w )
            best_depth = depth; // store best depth
}
depth = best_depth;

///// search around first point (depth) for closest match
for( int i = 0; i < binary_search_steps; i++ )
{
    size *= 0.5;

    uv = p + ds * depth;
    vec4 charW = imageLoad( nrmMap, ivec2( int( uv.x * imageSize.x + 0.5 ),
                                             int( uv.y * imageSize.y + 0.5 ) ) );
    float w = charW.a; //compute shader already returns value normalized [0,1]

    w = 1 - w;
    if (depth >= w)

```

```

    {
        best_depth = depth;
        depth -= 2 * size;
    }
    depth += size;
}
return best_depth;
}

void rayCastReturnCollisionCurvedInfo()
{
    if( gl_GlobalInvocationID.x == 0 && gl_GlobalInvocationID.y == 0 )
    {
        vec2 VSCurvature = vec2( inDat.buf.curvXY.x, inDat.buf.curvXY.y );
        vec2 VSuv = vec2( inDat.buf.startUV.xy );
        vec3 VSview = vec3( inDat.buf.tanDir.xyz );
        vec3 VSscale = vec3( inDat.buf.uvScale.x, inDat.buf.uvScale.y, 0.5 );
        ////
        uvec2 imageSize = imageSize( nrmMap );

        //Ensure that VSuv is normalized to floating point values in the range [-1,1]
        VSuv.x = float(VSuv.x) - float(int(VSuv.x)); //same as fract(VSuv.x)
        VSuv.y = float(VSuv.y) - float(int(VSuv.y));

        vec3 v = normalize( VSview );
        v.z *= -1;

        // mapping scale from object to texture space
        vec3 mapping = 1.0 / (VSscale);
    }
}

```

```

// evaluate quadric z value at x,y using curvature coefficients
float dataz = VSCurvature.x * v.x * v.x + VSCurvature.y * v.y * v.y;

dataz = sign( dataz ) * max( abs( dataz ), 0.001 );

// compute max distance for search min(t(z=0),t(z=1))
float d = v.z * v.z - 4 * dataz * VSscale.z;
float tmax = 50;
if( d > 0 ) //t when z = 1
    tmax = min( tmax, ( -v.z + sqrt(d) ) / ( -2.0 * dataz ) );
d = v.z / dataz; // t when z=0
if( d > 0 )
    tmax = min( tmax,d );

// transform view and quadric data to texture space
v *= mapping;
dataz *= mapping.z;

float t = rayIntersectDepthMapCurved( VSuv, v, tmax, dataz, imageSize );
outDat.buf.depth = t;
outDat.buf.hypotLen = t; //t is both depth an hypot because it has been scaled
                        //by the UV coord in texture space
if( t > tmax )
{
    outDat.buf.depth = -1.0; //Notify the CPU that no intersection has occurred
    outDat.buf.hypotLen = -1.0;
}

// compute intersected position
vec3 p = rayPositionCurved( t, VSuv, v, dataz );
vec2 uv = p.xy;

```

```
vec3 rayDispCurved = rayDisplacementCurved( t, v, dataz );

ivec2 pixel = ivec2( int(p.x * imageSize.x + 0.5),
                    int(p.y * imageSize.y + 0.5) );

vec3 normal = imageLoad( nrmMap, pixel ).xyz;
//expand normal from normal map in local polygon space
normal.xyz = normalize( normal.xyz * 2.0 - 1.0 );

//compute the 3D collision point in tangent space since that's the only
//vector we have access to
vec3 ddview = normalize( VSview.xyz );
vec3 tanSpaceDisp = ddview * t;

outDat.buf.uvOffset[0] = p.x;//inDat.buf.uvScale.x;
outDat.buf.uvOffset[1] = p.y;//inDat.buf.uvScale.y;

outDat.buf.xyzDispTS[0] = tanSpaceDisp.x;
outDat.buf.xyzDispTS[1] = tanSpaceDisp.y;
outDat.buf.xyzDispTS[2] = tanSpaceDisp.z;
outDat.buf.xyzDispTS[3] = float(pixel.x) / float(imageSize.x);

outDat.buf.normal[0] = normal.x;
outDat.buf.normal[1] = normal.y;
outDat.buf.normal[2] = normal.z;
outDat.buf.normal[3] = float(pixel.y) / float(imageSize.y);
}
}
```

```
float rayIntersectDepthMapCurved( vec2 tx, vec3 v, float tmax,
```

`float dataz, uvec2 imageSize )`

```

{
    const int linear_search_steps = 64;

    float t = 0.0;
    float size = ( tmax + 0.001 ) / linear_search_steps;

    // search front to back for first point inside object
    for( int i = 0; i < linear_search_steps; i++ )
    {
        vec3 p = rayPositionCurved( t, tx, v, dataz );
        ivec2 pixel = ivec2( int(p.x * imageSize.x + 0.5) % imageSize.x,
                             int(p.y * imageSize.y + 0.5) % imageSize.y );
        float tex = 1.0 - imageLoad( nrmMap, pixel ).w;
        if( p.z < tex )
            t += size;
    }

    const int binary_search_steps = 5;

    // recurse around first point for closest match
    for( int i = 0; i < binary_search_steps; i++ )
    {
        size *= 0.5;
        vec3 p = rayPositionCurved( t, tx, v, dataz );
        ivec2 pixel = ivec2( int(p.x * imageSize.x + 0.5) % imageSize.x,
                             int(p.y * imageSize.y + 0.5) % imageSize.y );
        float tex = 1.0 - imageLoad( nrmMap, pixel ).w;
        if( p.z < tex )
            t += 2 * size;
        t -= size;
    }
}

```

```

    return t;
}

vec3 rayPositionCurved( float t,      // search parameters
                        vec2 tx,      // original pixel texcoord
                        vec3 v,       // view vector in texture space
                        float dataz ) // data constants
{
    vec3 r = v * t;
    r.z -= t * t * dataz;
    r.xy += tx;
    return r;
}

vec3 rayDisplacementCurved( float t,      // search parameters
                             vec3 v,      // view vector in texture space
                             float dataz ) // data constants
{
    vec3 r = v * t;
    r.z -= t * t * dataz;
    return r;
}

```

OHIO
UNIVERSITY

Thesis and Dissertation Services