

Efficient Representation of Layered Depth Images for Real-time Volumetric Tests

Matthias Trapp[†] & Jürgen Döllner[‡]

Hasso-Plattner-Institute, University of Potsdam, Germany

Abstract

Representing Layered Depth Images (LDI) as 3D texture can be used to approximate complex, arbitrarily shaped volumes on graphics hardware. Based on this concept, a number of real-time applications such as collision detection or 3D clipping against multiple volumes can be implemented efficiently using programmable hardware. One major drawback of this image-based representation is the high video memory consumption. To compensate that disadvantage, this paper presents a concept and associated algorithms that enable a lossless, efficient LDI representation which is especially designed for the usage within shader programs. The concept comprises the application of a viewpoint selection, a cropping, and a compression algorithm. We evaluated our algorithm with different test cases and show possible use cases.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors; I.3.5 [Computer Graphics]: Boundary representations; I.3.6 [Computer Graphics]: Graphics data structures and data types;

1 Introduction

Real-time volumetric tests introduced in [TD08] enable a multiple binary partition of a given arbitrary scene on vertex, primitive, and fragment level. They have a number of applications in real-time rendering and interactive visualization, such as pixel-precise clipping, collision detection, and rendering with hybrid styles [J03]. This volumetric parity test (VPT) relies on an image-based representation of solid, arbitrarily shaped polygonal meshes (volumes). This representation is an extension of the concept of Layered Depth Images (LDI) [SGwHS98].

On GPU, an LDI can be stored as 3D texture or 2D texture array [NVI06] of depth maps. This form of representation enables a full hardware accelerated creation using render-to-texture (RTT) but introduces also a number of disadvantages. The main drawback of this approach is the high space complexity of the 3D texture that results in a high amount of graphics memory which is usually a limiting factor. The texture size depends on two conditions: the depth complexity

and the utilization of the depth maps. The depth complexity of a shape determines the number of necessary texture layers in an LDI and has therefore an effect on the creation time. For non-convex shapes, most of the depth maps are usually sparsely utilized. Further, the implementation of the VPT has to consider all texture layers to ensure correct results. This can become costly in terms of runtime because most of the texel fetches from a sparse utilized texture are redundant.

The aim of this work is to accomplish an efficient GPU representation in terms of minimizing the texture size and the necessary texture fetches. Thus, this paper makes the following contributions. It presents three algorithms that facilitate the efficient creation and storage of an LDI:

1. A method to find an optimal viewpoint for the creation of an LDI for which the depth complexity is minimal.
2. A fast algorithm to determine the axis-aligned bounding box (AABB) of an LDI. It is used to crop unused coherent texture areas.
3. A lossless compression algorithm that encodes the depth values of a 3D texture into a 2D texture and thereby achieves maximal texture utilization. The decompression can be performed using programmable hardware.

[†] matthias.trapp@hpi.uni-potsdam.de

[‡] doellner@hpi.uni-potsdam.de

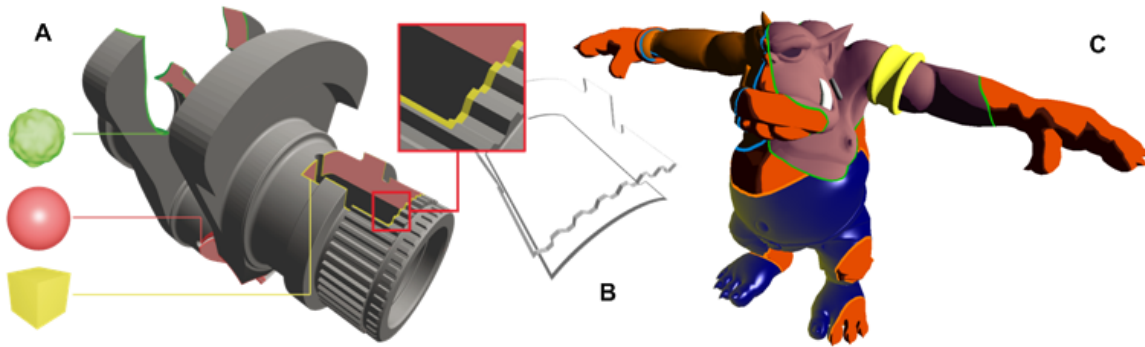


Figure 1: Application examples for Volumetric Depth Sprites (VDS) in combination with a Volumetric Parity Test (VPT). Figure A shows pixel-precise clipping against three different VDS. The same approach can be used to extract cut-edges (Figure B). Figure C shows rendering with hybrid styles. All images are rendered within a single pass.

The remainder of this paper is structured as follows. Section 2 reviews approaches and related techniques of this work. Section 3 presents an overview to the concept of volumetric depth sprites, reviews the idea of volumetric tests, and shows application examples. Section 4 introduces algorithms for viewpoint selection, fast 2D bounding box calculation, and lossless compression which facilitates an efficient representation of an LDI. Section 5 presents our results and discusses limitations of our approach while Section 6 concludes this paper and gives ideas for future work.

2 Related Work

The basic concept of LDIs is presented in [SGwHS98]. An LDI is a view of the scene from a single input camera view but with multiple pixels along each line of sight. The size of the representation grows only linearly with the observed depth complexity in the scene. In [BH03], a hardware-accelerated method for volumetric collision detection and intersection volume approximation is presented. The intersection test is CPU based, and thus not applicable in shader programs.

In [DL01] a compression algorithm for LDI is introduced that records the number of LDI layers at each pixel location, and compresses LDI color and depth components separately. For LDI layer with sparse pixels, the data is aggregated and then encoded. This representation cannot be decoded efficiently using shader programs. The same holds true for the representation of compressed depth maps presented in [CSSH04]. It uses a mesh representation for the compression of the depth maps.

Perfect spatial hashing [LH06] is a general approach for compressing sparse 3D textures within a dense offset table. This approach is designed for the usage in shader programs but introduces two new textures, a hash and an offset table instead of a single original texture. Applying this method would increase the implementation complexity, for example, of the VPT.

The depth peeling algorithm for order-independent transparency was introduced by [Cas01]. It uses a second depth test to extract layers of unique depth complexity from an arbitrary scene. It is possible to re-use these layers by performing a render-to-volume technique [Dro07]. In [Lef03] various memory layout options and optimizations are discussed. In this context, ray marching is a well known algorithm for interactive volume rendering [ZRL*07].

3 Concept of Volumetric Depth Sprites

A Volumetric Depth Sprite (VDS) is an image-based representation of the shapes volume that stores its depth values along a viewing ray that is aligned towards the negative z -axis. A VDS extends the concept of LDIs [SGwHS98] that contain layers of unique depth complexity. Figure 2 shows an example of a VDS derived from a complex 3D shape. A VDS representation consists of the following components:

$$VDS = (P, LDI, d, w_i, h_i) \quad (1)$$

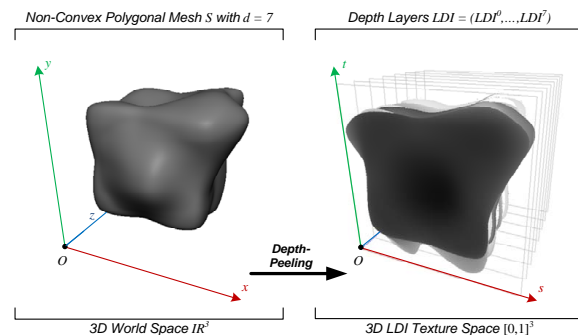


Figure 2: Example for an layered depth image representation of a non-convex polygonal mesh. S is depth-peeled into a number of slices, each containing depth maps of unique depth complexity.

Where $P \in \mathbb{R}^3$ denotes the position of the VDS in world space coordinates. The depth complexity of S is denoted as $d \in \mathbb{N}/\{0,1\}$. The layered depth image consist of d depth maps $LDI = (LDI^0, \dots, LDI^{d-1})$. The initial texture resolution of width and height is given by $w_i, h_i \in \mathbb{N}$. To obtain a depth value $d_i \in [0, 1] \subset \mathbb{R}, 0 \leq i \leq d-1$ in the i^{th} -depth layer for a 2D point $(s, t) \in [0, w_i] \times [0, h_i]$, we sample the 3D texture in LDI texture space with the coordinate $LDI^i_{(s,t)} = (s, t, i)$.

3.1 Hardware Accelerated Creation Process

The creation of a VDS is performed within a pre-processing step using multi-pass RTT. Given a solid polygonal mesh S , the associated LDI is generated by performing the following steps:

1. Uniformly scale the shape to fit into the unit volume $[0, 1]^3$. A camera orientation O_{DP} and on orthogonal projection is set that covers this unit volume. The near and far clipping planes are adjusted accordingly.
2. Determine depth complexity d and create a 3D texture or 2D texture array with an initial resolution of w_i , height h_i , and depth d . Our implementation uses a luminance texture format with a single 32bit floating point channel. The texture is initialized with a depth of 1, that we refer to as invalid depth value.
3. Perform depth-peeling [Cas01] in combination with RTT. The solid S is peeled using linearized depth values using a W -buffer [LJ99]. Figure 3 shows an OpenGL shading language (GLSL) implementation of the second depth test necessary for depth peeling.

The implementation is based on OpenGL [NVI06] in combination with GLSL [Kes06]. We use framebuffer objects, high precision 32bit float textures, and floating point depth buffer precision for RTT.

```
uniform sampler3D LDI;
uniform int pass;
varying float linearDepth;

void main(void){
    if((pass > 0) &&
        (linearDepth <= texelFetch3D(LDI,
            ivec3(gl_FragCoord.xy, pass-1),0).x)){
        discard; }
    gl_FragDepth = linearDepth;}
```

Figure 3: GLSL implementation of depth peeling with a 3D texture that contains layers of unique depth layers.

3.2 Volumetric Parity Test

Given a VDS, the Volumetric Parity Test (VPT) classifies a point $V \in \mathbb{R}^3$ with respect to its position in relation to the shape's volume. It can either be inside or outside the volume. For reasons of precision, we do not consider the case that the point can be on the border of the shape. To model such test, we introduce a Boolean *coordinate parity* $p_T \in \{0, 1\}$.

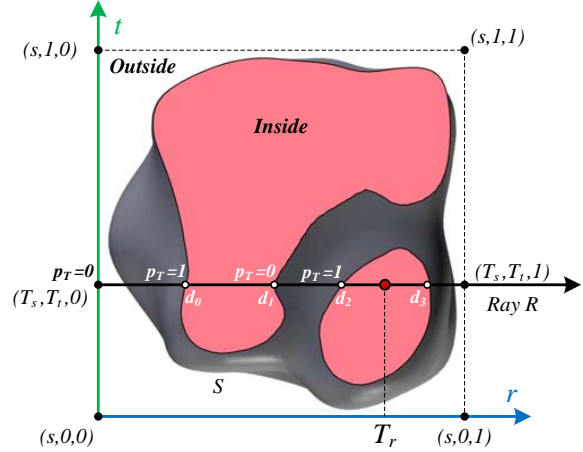


Figure 4: Ray marching through an LDI representation of the complex shape shown in Figure 2. The ray R intersects the depth layers LDI^i at four points and adjusts the rays parity p_T accordingly.

Before testing V , it must be transformed into the specific 3D LDI texture. For example, if V is a point in world space coordinates, the transformed coordinate T can be obtained by:

$$T = (T_s, T_t, T_r) = \mathbf{M} \cdot V \quad (2)$$

The matrix \mathbf{M} represents the mapping of world space coordinates into LDI texture coordinates. It is defined by $\mathbf{M} := \mathbf{T}(C) \cdot \mathbf{S} \cdot \mathbf{B} \cdot \mathbf{T}(-P)$. Where \mathbf{B} is a rotated orthonormal base of the VDS. V is transformed into the LDI texture coordinate space $(\mathbf{B} \cdot \mathbf{T}(-P))$, scaled by \mathbf{S} , and then translated $(\mathbf{T}(C))$ into the LDI origin $C = (0.5, 0.5, 0.5)$.

Now, we construct a ray $R = [(T_s, T_t, 0)(T_s, T_t, 1)]$ that marches through the depth layers LDI^i and compares T_r with the stored depth values d_i . Starting with an initial parity, p_T is swapped every time R crosses a layer of unique depth complexity (see Figure 4). This test can be formulated as $p_T = VPT(T, LDI)$ so that:

$$VPT(T, LDI) = \begin{cases} 1, & \exists d_i \wedge \exists d_{i+1} : d_i \leq T_r \leq d_{i+1} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

$$d_i \in LDI^i_{(T_s, T_t)} \quad d_{i+1} \in LDI^{i+1}_{(T_s, T_t)}$$

3.3 Application Examples for Volumetric Depth Tests

Despite clipping and collision detection, the concept of VDS and the associated VPT has a number of applications. For example, it enables rendering with hybrid styles [J103] per pixel and facilitates the generic usage of volumetric 3D lenses [VCWP96] without limitations concerning the volumes shape or the intersection of lenses. The rendering of volumetric depth sprites is similar to those of nailboards (depth or z-

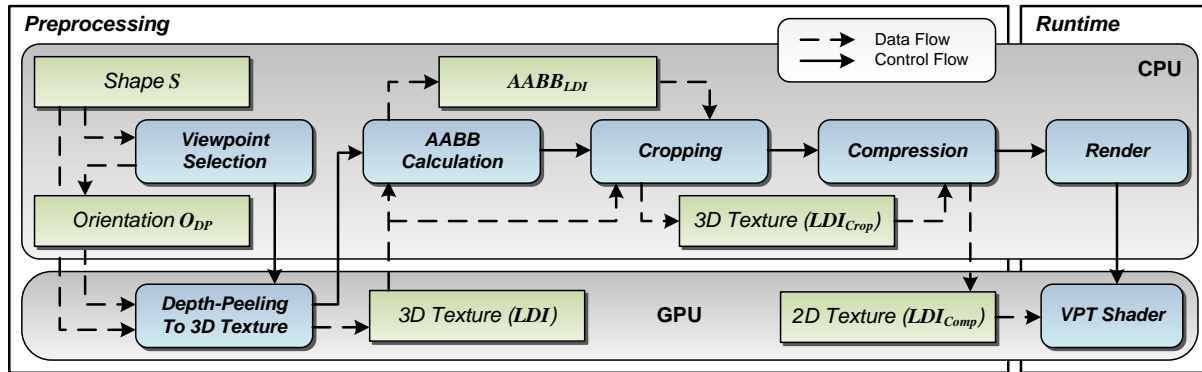


Figure 7: Conceptual overview and data flow between algorithms participated in the preprocessing of an input shape S into its LDI representation. After proper viewpoint selection, S is depth-peeled, cropped, and then compressed.

```

bool volumetricParityTestSM4(
    in vec3      T,          // Point in LDI texture-space
    in sampler3D LDI,       // layered depth image
    in ivec3     dimensions, // LDI dimensions
    in bool     initParity) // initial parity
{ // initial parity; true = outside
    bool parity = initParity;
    // for each texture layer do
    for(int i = 0; i < dimensions.r; i++){
        // perform depth test
        if(T.r <= texelFetch3D(LDI, ivec3(T.st, i), 0).x){
            parity = !parity; // swap parity
        }
    }
    return parity; }
    
```

Figure 5: Efficient GLSL implementation of the VPT.

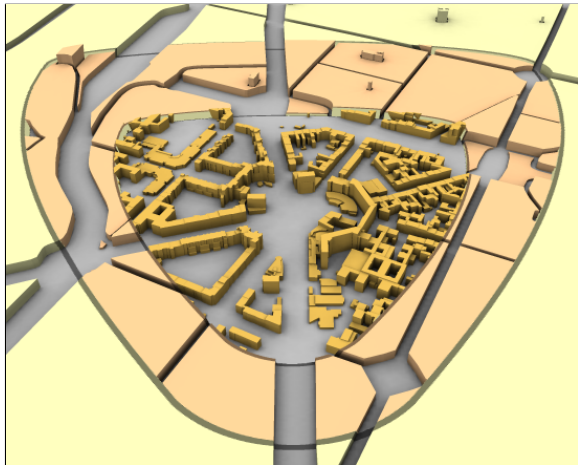


Figure 6: Combination of different geometries in a geovirtual environment using 3D lenses. It is generated using pixel-precise clipping and multi pass rendering.

sprites) introduced in [Sch97]. It is implemented using a ray-marching and z-replacement shader.

Clipping: One application is clipping a polygonal shape against multiple VDS. This can be done pixel-precise, by applying the VPT in the fragment shader for each fragment.

The fragments coordinate is interpolated in eye-space and then transformed into the LDI texture space using Equation 2. If the fragment is inside a VDS it is discarded. Figure 1.A shows a result for performing clipping against three volumes within a single rendering pass.

An extension of the same method can be applied for visualizing the shapes borders or to extract cut-edges. For a correct application in 3D space a 3D point V is transferred into a cube and then each corner vertex of this cube is tested using the VPT. So, given a border size we offset V in each of the eight directions. Figure 1.B shows this method by clipping all non-border fragments.

Rendering with Hybrid Styles: Instead of per-object hybrid rendering [J103], we are able to perform rendering with different styles on per-vertex, per-primitive, and per-pixel basis within a single rendering pass (Figure 1.C). Therefore, we map a style to each VDS and exploit static branching to apply the particular style.

Multiple 3D Lenses: Another interesting application for volumetric depth sprites are 3D lenses [VCWP96] for focus + context visualization of large scale scenes. Figure 6 shows an application that integrates different geometries within a single image using clipping and multi-pass rendering. Our approach delivers an alternative to multi-pass image-based rendering algorithms [RH04, Rop04].

4 Algorithms for Efficient LDI Representation

One main drawback is the high memory consumption of $M = w \cdot h \cdot d$ when representing an LDI as 3D texture. This is especially true for shapes with a high depth complexity d . Consequently, lowering the texture resolution w or h can result in a lack of precision when performing volumetric tests. Our goal is to determine optimal width w_i , height w_i and depth d to providing a high texture resolution simultaneously. A reduction of d implies a reduced number of depth-peeling passes which would speed up the dynamic creation of an LDI. To achieve this, we propose three algorithms. The flowchart

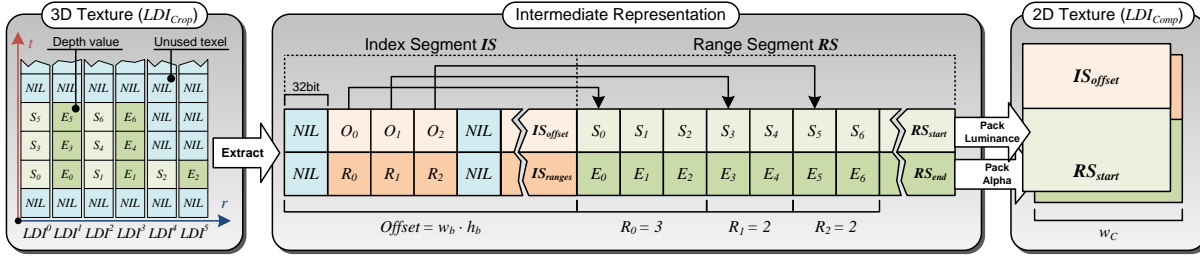


Figure 10: Concept of compressing the 3D texture LDI_{Crop} into a 2D texture LDI_{Comp} . All depth values are extracted from the depth maps LDI^i and then stored successively into a range segment RS . During this process an index segment IS is constructed. It stores an offset (O_V) for a particular 2D coordinate (s, t) that points to the start of the respective depth ranges within RS as well as the number (R_V) of successive depth ranges DR_j . This intermediate representation is packed into a 2D texture LDI_{Comp} .

in Figure 7 describes the complete preprocessing including the following optimization algorithms.

4.1 Viewpoint Selection for LDI Creation

This step determines a camera orientation O_{DP} with a minimal depth complexity $d_{min} \leq d_{max}$. The approach is only effective for non-convex shapes with $d_{max} > 2$. We use a simple setup for viewpoint selection. The camera is placed onto a unit sphere that is constructed around the center of the shape S . We modify the camera position with respect to its horizontal and vertical position on the sphere. The pseudo code to determine the camera orientation O_{DP} for a shape S , an initial viewpoint O_C , and the number of horizontal s_H and vertical segments s_V is presented in Figure 8. For each segment on the sphere, we calculate the orientation O_C by

```

procedure orientation( $S, O_C, s_H, s_V$ ) {
    // for each segment on sphere
    for ( $h \leftarrow 0; h < s_H; h \leftarrow h+1$ )
        for ( $v \leftarrow 0; v < s_V; v \leftarrow v+1$ )
             $O_C \leftarrow \text{adjustOrientation}(O_C, s_H, s_V)$ 
            // calculate metric
             $d \leftarrow \text{depthComplexity}(S, O_C)$ 
             $o \leftarrow \text{coverage}(S, O_C)$ 
            // store result
            append(list, ( $O_C, d, o$ ))
        }
    // sort results
    sortDepthComplexityAscending(list)
    sortCoverageDecending(list)
     $O_{DP} \leftarrow \text{getOrientation}(list, 0)$ 
}
    
```

Figure 8: Pseudo code for choosing the optimal orientation for LDI creation.

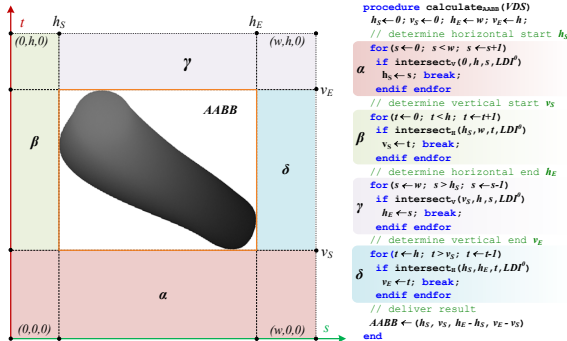


Figure 9: Algorithm for determining the AABB of an LDI efficiently. The algorithm uses a scan line approach and tests each texel of the LDI just once.

rotation the camera position around the x and y-axis with $\theta = 360/s_H$ and $\phi = 360/s_V$. Following to that, we determine the depth complexity d and the coverage ratio c of the occupied and invalid texels. The results of all segments are stored in a list that is sorted ascending by depth complexity afterwards. Under the preservation of this order, we sort the coverage values c in a descending order to obtain the orientation with the minimal depth complexity d and the maximal coverage c . After O_{DP} is retrieved we create the VDS according to Section 3.1 and then proceed to crop and compress the results.

4.2 Bounding Box Calculation & Cropping

Since hardware is not bounded to power-of-two texture dimensions anymore [NVI06], it is possible to optimize the texture storage on video memory by cropping the 3D texture to its 2D axis-aligned bounding box $AABB_{LDI} = (x, y, w_b, h_b)$ that includes all occupied (valid) texel in the LDI. This is particularly efficient if the shape has a main spatial extend along one of the two axis s and t , e.g., such as a torus. The AABB calculation is performed on CPU using a scan line approach. At first, we read the LDI texture back into main memory and then apply the algorithm described in Figure 9. The algorithm needs to test every texel just once. It uses two functions, $\text{intersect}_H(h_S, h_E, v, LDI^i)$ and $\text{intersect}_V(v_S, v_E, LDI^i)$ that test if a horizontal or vertical scan line, defined by two rays $R_H = [(h_S, v)(h_E, v)]$ and $R_V = [(h, v_S)(h, v_E)]$, contain a valid LDI texel. After the $AABB_{LDI}$ is calculated, we crop the 3D texture against it. The cropped LDI is denoted as LDI_{Crop} .

4.3 Compression Algorithm

Compressing the 3D texture representation of an LDI can decrease the amount of memory that sparsely occupied depth layers require on hardware. Compression can also increase the application performance by reducing the texture upload time and the number of texture samples: Due to the design of the VPT, the ray-marching algorithm (Figure 5) has to take all depth layers into account to decide if a 3D point lies inside the volume or not. So, texture samples are retrieved for layers that may not contain any depth information.

Concept: For sparse 3D textures, perfect spatial hashing [LH06] can be applied to loss less pack sparse data into a dense table. Since LDIs contain only depth values that describe a solid volume, we can propose a simpler alternative for compression. Unlike existing compression algorithms for LDIs [DL01] our approach has not to deal with color information and exploits this specific property by storing only the structure of the LDI into a 2D texture. Consider a ray R as depicted in Figure 4. The depth values $d_i, i = 0, \dots, d-1$ at the intersection points of R and the LDI^i depth maps can be grouped into a number of depth ranges $DR_j = (S_j, E_j)$, with $j = 0, \dots, d/2$. The interval $[S_j, E_j]$, with $S_j = d_{j \cdot 2}$ and $E_j = d_{j \cdot 2 + 1}$ specifies the inside of the volume along R . The proposed compression algorithm consists of two phases: *extract* and *pack*. Figure 10 illustrates the process and involved entities.

Extract: The first phase extracts all available depth ranges and stores the values of the even depth layer into $RS_{start} = (S_0, \dots, S_m)$ and the values of the odd layer into $RS_{end} = (E_0, \dots, E_m)$ respectively. Simultaneously, an index segment IS , consisting of the vector $IS_{offset} = (O_0, \dots, O_n)$ and $IS_{ranges} = (R_0, \dots, R_n)$, with $n = w_b \cdot h_b$ is constructed. IS_{offset} , initialized with a zero offset, stores an offset into the RS for every 2D texel (s, t) in the first depth layer LDI^0 . IS_{ranges} stores the number of depth ranges for the coordinate (s, t) . Thus, we denote an *index* as a tuple $I = (O_V, R_V)$, where R_V represents the number of depth ranges $DR_i, i = 0, \dots, R_V$ for the ray coordinates (R_s, R_t) . The pseudo code displayed in Figure 11 provides details the first phase of the compression algorithm that calculates the content of the specific segments.

Pack: The second phase packs the IS and RS into a 2D luminance-alpha texture which is denoted as LDI_{Comp} . Therefore, IS_{offset} and RS_{start} are stored successively in the luminance channel and IS_{ranges} and RS_{end} in the alpha channel respectively. The texture resolution is given by $w_c = h_c = \lceil \sqrt{|IS| + |RS|} \rceil$. Due to the constraints of texture resolution, it is not possible to use 1D textures or texture arrays for our data structure. The current hardware generation [NVI06] limits the 1D texture resolution to 8192 pixels which can easily be exceeded. Therefore, we have to evade to 2D textures and need to introduce an additional un-mapping step for decompression.

```

procedure extractVDS(VDS)
vector  $IS_{offset}, IS_{ranges}, RS_{start}, RS_{end};$ 
for ( $s \leftarrow 0; s < w; s \leftarrow s + 1$ )
for ( $t \leftarrow 0; t < h; t \leftarrow t + 1$ )
    // check if texel is set
    if ( $LDI[s, t, 0] \neq NIL$ )
         $layers \leftarrow 0$ 
        // iterate over depth
        for ( $r \leftarrow 0; r < d; r \leftarrow r + 1$ )
             $depthValue \leftarrow LDI[s, t, r]$ 
            if ( $depthValue \neq NIL$ )
                if ( $r \& 2 = 1$ )
                     $append(RS_{start}, depthValue)$ 
                else
                     $append(RS_{end}, depthValue)$ 
                endif
            else break
        endfor
         $IS_{offset}[t \cdot w + s] \leftarrow \|RS_{start}\|$ 
         $IS_{ranges}[t \cdot w + s] \leftarrow layers / 2$ 
    endif endfor endfor
end
    
```

Figure 11: Pseudo code for extracting depth ranges.

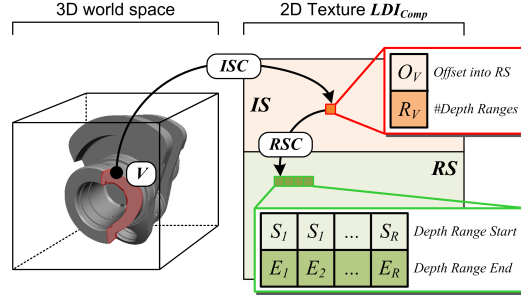


Figure 12: Coordinate transformation to access the depth ranges of a 3D point V .

4.4 Decompression on GPU

The decompression of an LDI_{Comp} can be performed in all programmable hardware stages. Figure 13 shows a GLSL implementation of the VPT for compressed and cropped LDI. We use the NVIDIA `gpu_program4` extension [NVI06] for hardware support of unsigned integer data type and unfiltered texel fetching. Figure 12 illustrates the process of fetching texels from a compressed LDI by unmapping texture coordinates.

To obtain the coordinates ISC into the IS for a given point V , we first transform V using Equation 2. After scaling to compensate cropping, we linearize (T_s, T_t) with respect to the texture resolution of the original (w_i, h_i) and cropped LDI (w_b, h_b) . We then re-interpret the linearized coordinate with respect to w_c . After ISC is calculated, we retrieve the range segment offset O_V and the number of depth ranges R_V . Now, the coordinate RSC into the range segment RS is determined and the depth ranges DR_j are successively sampled. The VPT







```

bool testAABB2D(in vec2 v, in vec2 b1, in vec2 b2){
    return (v.x>=b1.x)&&(v.y>=b1.y)&&(v.x<=b2.x)&&(v.y<=b2.y);
}

bool volumetricParityTestCompressedSM4(
in vec4 T, // point in LDI texture space
in sampler2D compressedLDI, // compressed LDI
in vec4 bounds, // crop bounds
in uvec2 size, // resolution of uncompressed LDI
in bool initParity) // initial parity
{ // initial parity; true = outside
bool parity = initParity;
// compensate cropping
if(testAABB2D(T.xy, bounds.xy, bounds.xy + bounds.zw)){
//map cropped coordinates to [0;1]
T.xy = (T.xy - bounds.xy) / bounds.zw;
// retrieve compressed texture size (CTS)
uvec2 CTS = uvec2(textureSize2D(compressedLDI, 0));
// uncompressed vds coordinates (UVC)
uvec2 UVC = uvec2(T.st * vec2(size));
// convert to linearized UVC (LUVC)
unsigned int LUVC = UVC.y * size.x + UVC.x;
// coordinate of the index segment (ISC)
ivec2 ISC = ivec2(LUVC % CTS.x, LUVC / CTS.x);
// sample from index segment IS
uvec2 IS = uvec2(texelFetch2D(compressedLDI, ISC, 0).ra);
if(IS.x != 0u) { // depth ranges available ?
for(unsigned int i = 0u; i < IS.y; i++) {
// calculate range sample coordinate (RSC)
ivec2 RSC = ivec2((IS.x + i) % CTS.x, (IS.x + i) / CTS.x);
// sample range depth range DR
vec2 DR = texelFetch2D(compressedLDI, RSC, 0).ra;
if(T.z <= DR.x && T.z >= DR.y){ // perform parity test
parity = !parity;
break;
} } }
return parity;}
    
```

Figure 13: GLSL shader source code for performing the VPT with cropped and compressed LDI.

Table 1: Performance results of our algorithms for input meshes of different depth complexity. The tests are performed with an initial texture resolution of $w_i = h_i = 1024$ and $s_H = s_V = 8$ segments for viewpoint selection. The time metric is seconds, the texture sizes are displayed in texel.

	Shape	#Vertex	d_{min}	d_{max}	t_{crop}	t_{comp}	t_{peel}	t_{view}	M_{crop}	M_{comp}	C_{ratio}
	Sphere	994	2	2	0.077	0.437	0.187	2.532	2,097,152	3,742,848	1.78
	Complex	768	2	6	0.078	0.297	0.187	3.085	2,097,152	3,011,058	1.44
	Cow	2,903	8	14	0.234	0.109	2.156	7.531	3,026,688	1,204,352	0.39
	Potato	6,146	6	12	0.203	0.219	25.766	7.172	5,683,200	3,317,888	0.58
	Knot	23,232	6	12	0.2	0.125	25.875	9.265	3,143,880	1,835,528	0.58
	Hebe	34,344	10	18	0.313	0.078	37.078	12.187	3,358,720	1,089,288	0.32

of Equation 3 is implemented by swapping the input parity p_T if $S_j < R_z < E_j$.

5 Results & Discussion

Our test platform is a NVIDIA GeForce 8800 GTS with 640 MB video memory and Athlon™62 X2 Dual Core 4200+ with 2.21 GHz and 2 GB of main memory at a viewport resolution of 1600x1200 pixel. The test application does not utilize the second CPU core. We have tested our algorithms with simple or complex, convex and non-convex input shapes of different geometric and depth complexity.

5.1 Performance

Preprocessing: Table 1 shows preprocessing results for different input shapes. The compression ratio is given by $C_{ratio} = M_{comp}/M_{crop}$. The proposed compression algorithm performs effectively for non-convex meshes with a high depth complexity. We are able to achieve compression ratios of 1:2-3 which is usual for lossless compression [DL01]. Compression should be avoided for symmetric convex meshes or meshes with $d_{min} = 2$ since the compressed texture size M_{comp} is always larger than the cropped size M_{crop} .

The runtime performance $t_{view}, t_{peel}, t_{crop}$ and t_{comp} depends on the geometrical complexity of the input mesh (#Vertex) and the initial resolution w_i, h_i of the LDI. The readback of the 3D texture from video memory to perform cropping and compression can become costly for large resolutions. To speed up the viewpoint selection, it can be performed with a lower resolution than the resolution needed for the actual bounding representation.

Volumetric Depth Test: We are able to render the depicted scenes at interactive frame rates (>15 FPS). The rendering performance depend on the number and depth complexity of the used LDIs, thus the number of samples the VPT has to perform, and the geometrical complexity of the rendered

scene. Performance tests point out that using compressed VDS is slower than using uncompressed ones. This can be explained by the calculation costs for the sampling coordinates of the depth ranges. Although, the computational complexity for the VPT reduces from $O(d)$ to $O(d/2)$ for compressed LDIs.

5.2 Limitations

Both, the concept of volumetric depth sprites and the proposed compression algorithm possess limitations that constrain their application.

Volumetric Depth Sprites: Figure 14 illustrates aliasing and undersampling artifacts that can occur during the VPT. They can be caused by low LDI resolutions and depend on the texture precision. Undersampling occurs for planes nearly parallel to the LDI direction. To compensate this drawback, one could apply multi-sampling but that increases the number of samples at the same time. Performing depth peeling with a high texture resolution for a number of complex shapes with $d > 7$ can hardly be done in real-time. Due to this, our approach is limited to static meshes because animating the shapes would require a re-computation of its VDS.

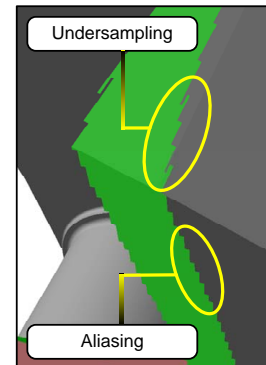


Figure 14: Occurring artifacts during VPT.

Compression Algorithm: The introduced depth range compression algorithm works only for LDIs that contain depth maps. If the user wants to incorporate additional per-text data,

such as normal or color, perfect spatial hashing [LH06] can be used.

Viewpoint Selection: Due to the regular step size for the alteration of the camera parameter, it cannot be guaranteed, that an orientation with a minimal depth complexity can be found for every shape.

6 Conclusions & Future Work

In this paper we have extended the concept of volumetric tests for real-time rendering purposes. We provided application examples as well as detailed sources for shader implementations of algorithms which based on this representation. Furthermore, we have presented three algorithms to facilitate the efficient storage of LDIs on GPU. We are able to minimize to space complexity for representing complex, non-convex polygonal meshes with a high depth complexity. So, we were able to compensate one of the key drawbacks of LDIs using raster data to represent complex volumes. The performance evaluation of these methods indicates a classical trade-off between space and time complexity.

We are heading to move the complete preprocessing phase of an LDI onto the GPU to avoid texture readback. This includes the AABB calculation, cropping, and the compression algorithm. Our main goal is to improve the preprocessing speed to be performed in real-time. This can enable the usage of animated meshes for the volume representation. Further, we research possibilities to perform an LDI-ray intersection test that allow line clipping and the calculation of intersection points. Furthermore, we will apply 3D noise to the LDI and border sizes to produce a sketchy impression [ND04] of the clipped areas.

Acknowledgments

This work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the InnoProfile research group "3D Geoinformation" (www.3dgi.de).

References

- [BH03] B. HEIDELBERGER M. TESCHNER M. G.: Real-Time Volumetric Intersections of Deforming Objects. In *VMV* (Munich, Germany, November 2003), pp. 47–54.
- [Cas01] CASS EVERITT: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation, 2001.
- [CSSH04] CHAI B.-B., SETHURAMAN S., SAWHNEY H. S., HATRACK P.: Depth Map Compression for Real-time View-based Rendering. *Pattern Recogn. Lett.* 25, 7 (2004), 755–766.
- [DL01] DUAN J., LI J.: Compression of the Layered Depth Image. In *DCC '01* (Washington, DC, USA, 2001), IEEE Computer Society, p. 331.
- [Dro07] DRONE S.: Real-Time Particle Systems On the GPU in Dynamic Environments. In *SIGGRAPH '07* (New York, NY, USA, 2007), ACM, pp. 80–96.
- [JI03] JESSE R., ISENBERG T.: Use of Hybrid Rendering Styles for Presentation. In *Poster Proceedings of WSCG 2003* (2003), pp. 57–60. Short Paper.
- [Kes06] KESSENICH J.: *The OpenGL Shading Language Language Version: 1.20 Document Revision: 8*, September 2006.
- [Lef03] LEFOHN A.: Interactive Visualization of Volumetric Data on Consumer PC Hardware. In *Tutorial, IEEE Visualization* (2003).
- [LH06] LEFEBVRE S., HOPPE H.: Perfect Spatial Hashing. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 579–588.
- [LJ99] LAPIDOUS E., JIAO G.: Optimal Depth Buffer for Low-Cost Graphics Hardware. In *HWWS '99* (New York, NY, USA, 1999), ACM, pp. 67–73.
- [ND04] NIENHAUS M., DÖLLNER J.: Blueprints: Illustrating Architecture and Technical Parts Using Hardware-Accelerated Non-Photorealistic Rendering. In *GI '2004* (2004), pp. 49–56.
- [NVI06] NVIDIA: *NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x)*, November 2006.
- [RH04] ROPINSKI T., HINRICHS K.: Real-Time Rendering of 3D Magic Lenses Having Arbitrary Convex Shapes. In *WSCG* (February 2004), vol. 12, pp. 379–386.
- [Rop04] ROPINSKI T.: Exploration of Geo-Virtual Environments using 3D Magic Lenses. In *EURESCO-ESF Conference on Geovisualization* (2004).
- [Sch97] SCHAUFLE G.: Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97* (London, UK, 1997), Springer-Verlag, pp. 151–162.
- [SGwHS98] SHADE J., GORTLER S., WEI HE L., SZELISKI R.: Layered Depth Images. In *SIGGRAPH '98* (New York, NY, USA, 1998), ACM, pp. 231–242.
- [TD08] TRAPP M., DÖLLNER J.: Real-Time Volumetric Tests Using Layered Depth Images. In *Proceedings of Eurographics 2008* (April 2008), K. Mania E. R., (Ed.), Eurographics, The Eurographics Association. to appear.
- [VCWP96] VIEGA J., CONWAY M. J., WILLIAMS G., PAUSCH R.: 3D Magic Lenses. In *UIST '96* (New York, NY, USA, 1996), ACM Press, pp. 51–58.
- [ZRL*07] ZHOU K., REN Z., LIN S., BAO H., GUO B., SHUM H.-Y.: *Real-Time Smoke Rendering Using Compensated Ray Marching*. Tech. Rep. MSR-TR-2007-142, Microsoft Research, September 2007.