Hardware-Accelerated Geometry Instancing for Surfel and Voxel Rendering of Scanned 4D Media

Marcos Slomp, Hiroshi Kawasaki Ismael Daribo Ryusuke Sagawa Ryo Furukawa, Shinsaku Hiura, Naoki Asada
Kagoshima University NII Japan AIST Japan Hiroshima City University

***Abstract*—Range scanners based on camera-projector setups permit capturing dynamic objects at high frame rates at affordable costs, producing a time-varying set of points. This paper establishes a 4D media framework for acquiring, compressing and displaying these point clouds, albeit with more emphasis on the rendering aspects. Displaying these point clouds requires techniques capable of alleviating irregularities that are inherent in these low cost high-speed scanners. One of the contributions of this paper is a simple, general surfel splatting method that mitigates these imperfections. Rendering must also react quickly to the volatility of the 4D media dataset and cope with reasonably large quantities of points. To that end, this paper exploits modern features of the graphics hardware, namely geometry amplification and instancing, for efficient streaming and displaying of point clouds as either surfel or voxel primitives. A comparative performance analysis of the investigated strategies is also available.**

I. Introduction

Inferring three-dimensional information from real objects is an important subject in computer vision. In recent years, hardware setups for 3D scanning have become largely available and affordable. End-users are now benefiting from breakthroughs in human-computer interaction, virtual reality and telepresence, on what was once exclusive technology for engineering, process automation, robotics and medical systems.

Range scanners built upon camera-projector configurations are capable of capturing fast moving entities at high frame rates. The amount of geometrical data produced by these scanners, however, imposes challenges on the storage, streaming and displaying of the time-varying three-dimensional information (4D media assets). This paper focuses on the rendering aspects of 4D media. Rendering must react quickly to these assets for smooth playback. Therefore, a robust rendering engine should produce attractive results given the time, volume and reconstruction constraints, with minimum lagging for preprocessing or stream preparation.

Scanners based on camera-projector setups produce a time series of 3D point samples, referred to as *point clouds*. Polygonal representations are also feasible, although likely to increase the overall processing time and storage requirements. In addition, maintaining polygonal consistency across multiple poses is difficult and prone to damaging compression efficiency and the smoothness of the rendering. Since point clouds comprise a more natural, robust and widely adopted representation for 3D scanning systems, the techniques described in this paper are geared towards rendering point-based input.

The graphics hardware has historically supported point primitives. Naïve point-cloud rendering engines for such hardware can therefore be implemented without much effort. Although fast to render, these straightforward point cloud visualizers bring many issues: holes in the surface, no back-face culling, limited per-point size control and harsh silhouettes due to screen-aligned square-shaped point splatting. Some hardware manufacturers provide palliative extensions to mitigate some of these problems, such as vertex culling, view-dependent variable point sizes, smooth points and point-sprites.

Surfels (short for “surface elements”) provide an alternative way for displaying point-based geometry. A surfel is a geometrical extension of a point that can be though of as an oriented, flat microfacet surface patch. The orientation of the surfels follow the curvature of the surface they approximate, thus allowing for smoother silhouettes, while their sizes serve the purpose of filling holes on the surface.

One disadvantage of surfels is the absence of general mechanisms for handling level of detail (LOD). Displaying detailed geometry in lower-resolution viewports is not only a waste of processing, bandwidth and memory resources, but also prone of introducing rendering artifacts due to high-frequency aliasing. Multiresolution surfel-based representations are possible at the cost of increased memory requirements and processing time. This ultimately escalates the complexity of managing, storing and displaying 4D media assets.

A perhaps less conventional approach for representing point clouds is through *voxel* hierarchies. Although commonly employed on dense volumetric data-sets, *sparse* voxel representations of hollowed surfaces have gained momentum recently in computer graphics. One of the many advantages of voxel hierarchies is the fact that they can elegantly address LOD and progressive rendering. Moreover, since geometry can be represented implicitly within a voxel hierarchy, storage requirements are reduced – or at least compete – with their polygonal, point, or surfel-based counterparts.

The main contributions of this paper are: the establishment of a complete framework for acquiring, compressing and rendering 4D media; the use of hardware-accelerated geometry instancing for fast rendering of surfels and voxels; a surfel splatting technique suitable for covering irregularities inherent in the recommended 4D scanning framework; a simple and effective stop criterion for octree-based hierarchical subdivision of point clouds; and a comparative analysis of the different hardware instancing rendering methodologies studied.

The remainder of the paper is structured as follows: Section II compiles the related work of the research; Section III provides an overview of the entire framework; Section IV discusses hardware instancing; Sections V and VI describes, respectively, surfel and voxel rendering techniques for point clouds; Section VII depicts additional results and a discussion of the methods; finally, Section VIII concludes the paper and points future research opportunities.

II. Related Work

Surfels as rendering primitives were introduced by Pfister et al. [1] to mitigate point-based rendering issues. Intuitively, these "surface elements" can be thought of as microfacet extensions of points, often rendered as oriented quadrilaterals. The method combines aspects from point, polygon and image-based rendering, being influenced by sprite-based techniques (billboards, or impostors) that until then were used for geometric level of detail [2]. In contrast to pure image-based rendering approaches, surfels allow for view-independent, object-centered representations of arbitrary surfaces.

Zwicker et al. later improved the rendering quality of surfels using screen-space elliptical-weighted average (EWA) filtering [3]. Ren et al. followed by proposing an object-space formulation for the EWA filtering that was more suitable for hardware acceleration [4]. Unfortunately, EWA filtering assumes hermetic surfaces which are difficult to reconstruct with the high-speed 4D moving object scanning framework that this paper builds upon. As a contribution, this paper offers a simplified surfel splatting mechanism to reduce surface holes due to irregularities during moving object scanning.

In any case, surfel rendering of point clouds requires promoting points to oriented quadrilaterals. This geometrical *amplification* is performed by the CPU prior to submitting the surfels for rendering to the GPU. An alternative is to preprocess the surfels, but this implies in increased memory and CPU-GPU bandwidth usage. The work of Guennebaud and Paulin is a notable exception [5]. They use regular point splatting and "skew" the screen-aligned square-shaped appearance of the splats programmatically in a fragment shader. The technique, however, relies on less trivial branching paths and discard instructions that are notorious for affecting fragment shading performance. As another contribution, this paper investigates and compares recent graphics hardware features applied to real-time surfel amplification and instancing.

Level-of-detail for surfels is generally achieved through hierarchical structures that encode multiresolution surfel information. Examples of techniques include the work of Yamazaki et al. [6] and the ones surveyed by Kobbelt and Botsch [7]. These schemes are difficult to apply for 4D media rendering due to the additional memory and processing time required.

Voxel hierarchies, on the other hand, are simpler than multiresolution surfel hierarchies since geometry can be represented implicitly. The literature on voxel rendering is extensive and much of it is out of the scope of this paper. More notably, Laine and Karras have recently investigated the use of *sparse voxel octrees* (SVO) as an alternative to polygons for real-time high-quality rendering in current graphics hardware technology [8]. As will be demonstrated in this paper, point clouds can be efficiently encoded as sparse voxel octrees.

Contrary to Laine and Karras – and to conventional voxel rendering paradigms in general – this paper focuses on rendering voxels through rasterization instead of ray-casting. The reason is the fact that despite recent advances towards more general-purpose graphics hardware, the underlying pipeline remains largely raster-oriented. The same hardware-accelerated geometry amplification and instancing techniques that this paper proposes for surfels can be likewise applied for efficient voxel rasterization.

III. A Framework for 4D Object Scanning

Data acquisition is performed through a number of cameras that register the patterns extruded by multiple projectors. The distribution of cameras and projectors around the target scene allows for entire shape scanning. The speed of the capturing process is only limited by the frame rate of the cameras. Multi-colored patterns are used to improve the accuracy and robustness of the final reconstruction. These image frames are later analyzed, wherein intersection points of the extruded patterns are identified. The correspondences between these points and the projection patterns are then established and surface reconstruction is obtained via triangulation. A complete description of the measurement system can be found in [9].

The raw output of the system is a time series of points. Surface normals and splat sizes are estimated by analyzing the vicinity around each point at each time frame. Textures, however, are not readily available since the multi-colored projected patterns interfere with the optical proprieties of the surfaces. This remains as an open problem for camera-projector setups that do not rely on monochromatic, white-colored patterns. For rendering purposes, in this paper a fake-color is assigned to each point based on their normal vectors.

Noise and irregularities are inherent in these time-varying point clouds, complicating their compression. The technique proposed by Daribo et al. [10] is capable of addressing these issues. Their encoder first promotes each raw point cloud into sets of 3D space curves. A competitive-based predictive unit removes the spatial and temporal correlation along and across the 3D space curves. Finally, a rate-distortion (RD) cost computation control guarantees the best trade-off between the utilized bitrate and end-to-end quality. Decoding takes the opposite direction by providing point clouds from the internally compressed curve-based representations.

At the very end of the framework is the rendering infrastructure, which attempts to synthesize images of the reconstructed point clouds. Rendering speed is severely constrained by the surface reconstruction and compression stages that precede it. In a more practical view, these former stages are favorably performed offline. Only the fully encoded 4D asset is allegedly streamed for rendering. Therefore, decoding performance is what actually restricts rendering speed.

A more aggressive and isolated rendering design approach will be taken in this paper. It is assumed that the decoder incurs zero overhead to the rendering stage, i.e., the point clouds are promptly available for displaying. The challenge is then to respond quickly and effectuate quality rendering only with the point attributes available. Given the amount of data contained in these 4D media assets, the rendering engine can spare little time for stream preparation or preprocessing.

IV. Hardware-Accelerated Geometry Instancing

Point clouds consist of sequences of point attributes such as position, normal vector, color and splat size. Surfels, however, are represented as aligned quadrilateral patches; similarly, octree voxels are best realized as cubes. The process of promoting points to higher-level primitives – like billboards or cubes – is known as *amplification*. In addition, the replication and transformation of large quantities of amplified geometry based on different attributes is known as *instancing*.

One of the contributions of this paper is the investigation of modern graphics hardware features suitable for real-time amplification and instancing of surfels and voxels. As will be seen, this can be accomplished either programmatically on the Geometry Shader unit or via a hardware-controlled geometry instancing pipeline, as shown in Figure 1. From the perspective of the CPU, its solely task, aside from setting the appropriate GPU state, is to pack each point attribute into a buffer and issue a *single* draw call on that buffer. The buffer itself is therefore nothing more than a mirrored copy of the point cloud data array. This not only guarantees the immutability of the point cloud data, but also yields optimal streaming efficiency, draw batching and CPU-GPU bandwidth usage.

A. Geometry Instancing Using the Geometry Shader

The Geometry Shader is a relatively new component of the graphics hardware pipeline, sitting in between the Vertex Shader and the Fragment Shader. This unit enables programmatic modifications on the geometry and topology of input rendering primitives as they flow through the pipeline. Modifications include both geometry amplification and simplification. Amplifications can be dynamic and arbitrary, but restricted to moderate levels, as the unit is ill suited for heavy tessellation of primitives. Fortunately, only constant-size amplifications, like quadrilaterals or cubes, are of interest in this paper.

Based on the above assumptions, the Geometry Shader unit is configured to accept point primitives and to output triangle strips. The actual description of the amplified geometry (the template) resides inside a user-defined program which is loaded into the Geometry Shader unit. Besides amplification, the geometry program is also responsible for instancing the augmented primitive. This means transforming the coordinates of the canonical template according to the respective point attributes. The only responsibility of the vertex program is to forward point attributes, untouched, to the Geometry Shader. Object, world, view and projection transformations take place within the geometry program running in the Geometry Shader. The process is depicted in Figure 1a.

B. Geometry Instancing Using the Instancing Pipeline

Hardware instancing allows for the rendering of identical objects in large quantities. The general idea is to keep a single geometric description of an object (the template) in video memory and render it many times using only a single draw call. In contrast with the Geometry Shader, hardware instancing does not require a programmatic descriptions of the template object, and has no restrictions regarding the complexity of the template geometry. In fact, the Geometry Shader unit is not involved at all here. Although primarily intended for rendering multiple instances of relatively dense polygonal meshes with different attributes, the instancing pipeline can be used to great effect for surfel and voxel rendering.

While in instancing mode, the GPU will reissue the same template geometry repeatedly. Each time, a special register, the *instance identifier*, is incremented and exposed to the Vertex Shader. For each instance, all vertices of the template will have the same instance identifier. A vertex program then uses this identifier to explicitly index per-instance (per-point) attributes from a buffer. Having retrieved the correct attributes, the vertex program performs the appropriate coordinate transformations. The hardware instancing pipeline is illustrated in Figure 1b.

The hardware offers three ways to specify and address per-instance attributes in buffers within the vertex program:

• **Uniform Buffers** expose per-instance data as regular shader uniform array variables. The maximum size of such buffers is rather limited (typically 64KB), however. This limitation can be circumvented by issuing multiple draw calls on multiple buffers (or subregions of a larger buffer).

• **Textures and Texture Buffers** embed per-instance data into texels which must be retrieved through texel-fetching instructions. The data has to be converted and encoded according to the pixel description of the texture. The maximum size is constrained by the video memory and the maximum texture resolution (and texel format) supported by the hardware.

• **Regular Vertex Buffers** feed per-instance data as regular vertex attribute shader variables. This dispenses the need for the instance identifier: the hardware itself takes care of loading per-instance data in the appropriated registers prior to invoking the vertex program. The amount of video memory available is the only buffer size restriction.

V. Surfel Rendering Techniques

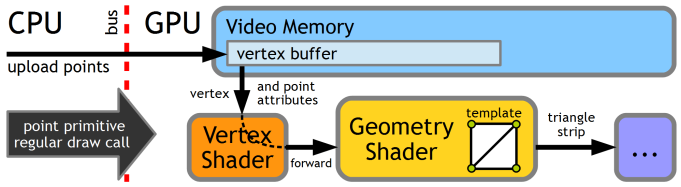
Surfels can be represented as oriented, texture-mapped quadrilaterals, much to the likes of billboards. However, while billboards are typically aligned with respect to view or world constraints, surfels align themselves to the curvature of the surface of the object they represent.

For all purposes, a generic template for a surfel can be specified as the flat quadrilateral enclosed by the extreme vertices $q_{min} = (-1, -1, 0)$ and $q_{max} = (+1, +1, 0)$. When rendering, this canonical template will be rotated, scaled and translated according to the attributes of the surfel. Each surfel is assumed to have the following attributes: a position p, a unit-length normal vector $\vec{n}$, a half-edge length r and a color. A circular, rotationally-invariant Gaussian-smoothed monochromatic texture mask is also assumed for shading. Refer to Figure 2 for the complete surfel rendering pipeline.

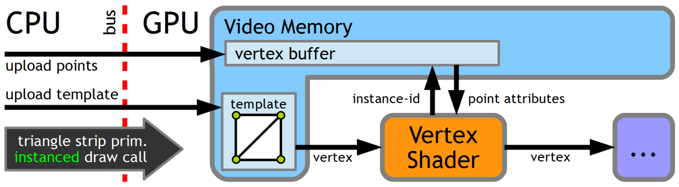
A. Surfel Alignment and Transformation

The process of splatting a surfel begins by aligning, stretching and positioning its billboard according to its surfel attributes, as illustrated in Figure 2a. This is done by post-multiplying each of the four vertices of the template by the following matrix product:

$$T(p) \cdot S(r, r, 1) \cdot R(\vec{u}, \vec{v}, \vec{n})$$



(a) Using the Geometry Shader unit.



(b) Using the instancing pipeline.

Fig. 1: Hardware-accelerated geometry amplification and instancing.

where T, S and R denote transform matrices for translation, scale and rotation, respectively, in homogeneous space. Expanding the formula yields:

$$\begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r & 0 & 0 & 0 \\ 0 & r & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the unknown vectors $\vec{u}$ and $\vec{v}$ above, this paper suggests the use of a dual cross product formulation:

$$\vec{u} = \vec{a} \times \vec{n} \qquad \text{and} \qquad \vec{v} = \vec{n} \times \vec{u}$$

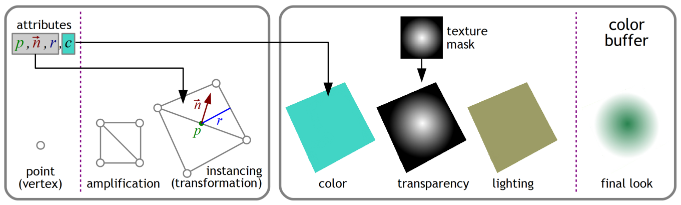
where $\vec{a} = (0, 1, 0)$. The cross products above are applicable as long as $\vec{n}$ is not parallel to $\vec{a}$, at which the vector $\vec{a}' = (0, 0, -1)$ takes place. In addition, one must ensure that both $\vec{u}$ and $\vec{v}$ are normalized prior to the computation of the rotation matrix above; this can be secured by normalizing $\vec{u}$ after its computation, thus eliminating the need for normalizing $\vec{v}$.

Note that this dual cross-product formulation is viable because surfels are assumed to be shaded later as disks through a rotationally-invariant circular texture mask. Had this not be the case, each point would have to include tangent and binormal vectors to replace $\vec{u}$ and $\vec{v}$ accordingly. In this paper, surfel billboards are also assumed to be isotropic; otherwise, the scaling matrix would also have to be modified to account for distinct half-edge lengths r_x and r_y. Also note that the matrix product above produces coordinates in object space. These coordinates should be further transformed by the usual world, view and projection matrices for proper rendering.

B. Shading Surfels

Once the quadrilaterals have been transformed, the pipeline performs back-face culling and schedules front-facing geometry for rasterization. Each fragment of a rasterized surfel is then shaded based on its color. Transparency is determined by sampling a circular, Gaussian-smoothed monochromatic texture mask. To that end, texture coordinates have to be assigned to each vertex of the template when it is first specified. Lighting can be performed in a per-instance, per-vertex or per-fragment basis. Finally, the shaded fragment is forwarded to the later stages of the pipeline and, if the fragment carries through the depth-test, blended on the color buffer. The entire surfel shading process is depicted in Figure 2b.

At this point, it is already possible to present on screen a snapshot of the scanned object. However, sharp square-shaped boundary artifacts are likely to appear, as can be seen in Figure 3a. This is due to alpha-blending and the relative rendering order of the surfels. One simple way to address such problem is by disabling depth buffer operations completely.



a) geometry processing stage b) fragment shading stage

Fig. 2: The surfel rendering pipeline.

Such an aggressive solution eventually leads to another sort of artifacts, allowing for distant surfels to be blended on top of closer ones, as shown in Figure 3b. The proper solution for both problems is to render the surfels in a back-to-front fashion. Unfortunately, due to the large quantity of surfels involved, view-dependent surfel sorting would substantially hurt the performance.

A more viable approach is a two-pass rendering algorithm [4], [5]. Initially, *visibility splatting* is performed, where only the depths of surfels are rendered. The depth buffer produced by this first pass (Figure 3c) will serve as a *read-only* resource for depth tests during the subsequent pass. In the second pass, surfels are issued for rendering again, but this time only shading is performed with no depth being outputted (Figure 3d). This technique, however, is only suitable for hermetic surfaces, which are difficult to reconstruct while scanning fast moving objects with the proposed framework.

Due to the irregular, non-hermetic nature of the surfaces inherent in the 4D media being addressed, proper surfel splatting coverage is difficult. Increasing the surfel splat size improve the results, but not without reintroducing sharp square-shaped boundary artifacts, as can be observed in Figure 3e.

As a solution, this paper proposes increasing the splat size of the surfels *only* during the *second* rendering pass. The advantages of this simple approach are twofold: it helps on covering surface holes, while at the same time yielding an even smoother look to the rendered object, as demonstrated in Figure 3f. The disadvantage is an increase in fragment overdraw, which is only a concern at extreme close-ups when pixel fillrate performance can become a rendering bottleneck.

VI. Sparse Voxel Octrees

Representing point clouds as sparse voxel octrees brings many advantages. First, it is possible to discard point coordinates, since the geometry is implicitly represented by the hierarchy. Second, voxelization can be used to filter noise locally and simplify the point cloud. Lastly, voxel hierarchies allow for automatic multi-scale representations, level-of-

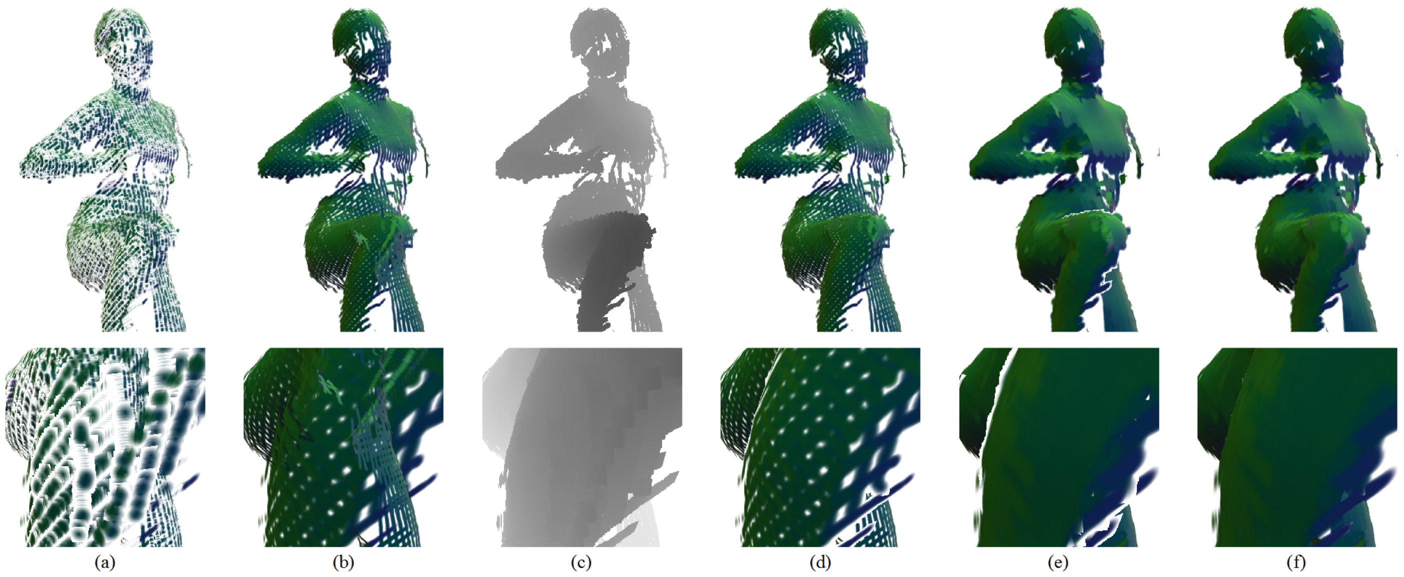


Fig. 3: Surfel splatting on a pose of the "kick" 4D point cloud dataset: a) regular splatting; b) splatting without depth; c) two-pass splatting: visibility pass; d) two-pass splatting: shading pass; e) fully enlarged splatting (2x); f) proposed enlarged splatting (2x).

detail (LOD) control and progressive rendering. Octrees also subdivide space regularity, thus requiring little bookkeeping information internally. This facilitates the process of traversing, modifying and storing the data structure.

Most of this section is intended as a gentle introduction to sparse voxel octrees. The contents to follow serve as a guide to the process of compiling and rendering simple SVOs built from raw point cloud data. Encoding of normal vectors and colors within the octree hierarchy will not be addressed in this paper. The interest reader is directed to [11] for a more in-depth discussion on the aspects not covered here.

A. *Building the Octree Hierarchy*

An octree is a three-dimensional extension of a quadtree, which itself is a two-dimensional extension of a binary-tree. Each node of the octree spawns up to eight children, each child describing the space occupancy of a particular octant. Space is subdivided uniformly by halving its boundaries in each of the three mutually perpendicular planes. This means that each of its eight octants have the exact same volume, hence uniquely defining a non-intersecting partition of the whole space.

The first step is to compute the axis-aligned bounding box (AABB) surrounding the whole point cloud. The length of the longest direction is selected as the edge length of the cube that will enclose the entire octree. The AABB is then appropriately scaled in all directions to reflect the new length. This ensures that all voxels will have cubic appearance instead of rectangular prisms.

Starting from the root of the octree, each point is classified as belonging to one of its internal octants. If there are points occupying an octant, a new node is assigned to that octant. Recursion follows at the newly created node with its own enclosing points. The recursion will only stop if an octant is empty or when a certain stop criterion is established. Consult Figure 4 for an example of octree construction.

There is no universal rule for the stop condition. A common approach is to stop subdivision once the octree reaches an arbitrarily defined depth. Despite being empirical, this is tedious since each pose of the 4D media might be best represented with a different maximum depth. Another approach is to stop subdivision once a certain number of points, at most, is contained within the octant. This, however, causes isolated points to occupy comparatively larger octants that, when rendered, appear incompatible with the expected majority of smaller octants, as can be seen in Figure 5.

As an alternative, this paper proposes a simple yet very effective stop criterion for subdividing point clouds. The rationale is that an octree is subject to further subdivision only if the total amount of octants at the deepest level is comparatively lower than the size of the point cloud. More formally, the condition $oct(h)/n < 1 - \delta$ drives the subdivision, where $oct(h)$ is the number of occupied octants at level h, and n is the total amount of points in the entire point cloud. The constant δ is a threshold that determines the acceptable loss during voxelization. A threshold of 0% implies on an ideal 1:1 ratio between terminal octants and points, while higher thresholds produce coarser but more compact volume hierarchies.

In practice, due to noise and irregularities imposed by the scanning process, maximum subdivision is unnecessary and higher thresholds can be used as filters. The stop condition just described is largely automatic and dynamic, adjusting itself according to the geometrical characteristics of each point

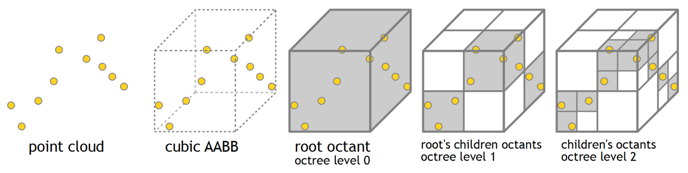


Fig. 4: Octree subdivision example of a simple point cloud.

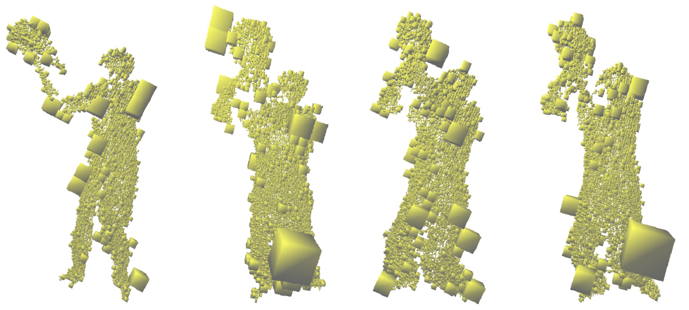


Fig. 5: Different poses of the “basketball” 4D point cloud dataset: incoherent voxel size due to isolated points producing larger octants when using an occupancy-based stop criterion.

cloud. Besides, it enforces that all leaf nodes will lie in the same octree level, thus giving them the same consistent size and aspect. This last feature is also particularly useful since it allows for very compact, breadth-first storage of octrees using only a single 8-bit mask per octree node.

B. Voxel Rasterization

Prior to rendering the voxels, the first step is to identify a potential set of visible voxels. This is done by comparing the view frustum of the camera against the octree hierarchy. Starting from the root octant, if it intersects with the frustum, the next level is traversed and the process repeats for each occupied octant. Octree traversal stops when there are no more subdivisions left or if the projected area of the octant on the screen is less than the area of a pixel. This ensures automatic LOD selection. For efficiency, frustum-octant intersections can be implemented as frustum-sphere intersections, with the diagonal of the octant serving as the diameter of the sphere.

Every time the traversal is halted due to one of the two aforementioned conditions, the position and size of that voxel are appended into a buffer. These attributes are determined implicitly as the octree is traversed since the AABB enclosing the entire octree is known. Once the octree is fully traversed and the buffer complete, any of the hardware instancing methods described in Section IV can be used for rasterizing the voxels. The canonical template for a voxel is the cube enclosed by the extreme vertices $c_{min} = (-1, -1, -1)$ and $c_{max} = (+1, +1, +1)$. Refer to Figure 6 for a point cloud rendered through voxels at different LOD.

VII. Results and Discussion

This section compiles and analyzes the performance of the different techniques presented in the paper. Additional rendering results are also provided in Figures 7, 8 and 9.

The overall characteristics of the investigated datasets are described in Table I. Performance results were profiled based on the following hardware and software configurations:

- **CPU**: Intel Core i5-2540M 2.60GHz with 4GB RAM
- **GPU**: NVIDIA Quadro 3000M with 2GB VRAM
- **Operating System**: Windows 7 Enterprise 64bit SP1
- **Graphics and Shaders**: OpenGL/GLSL 4.2

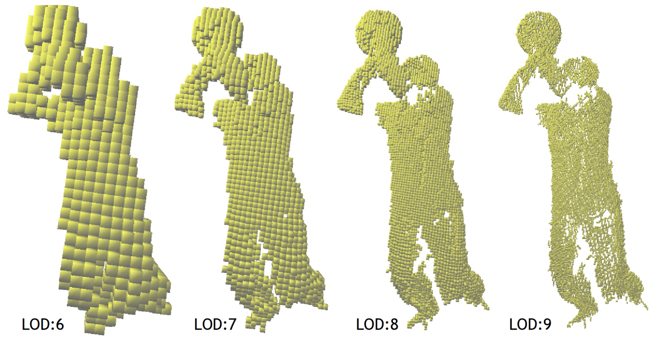


Fig. 6: A single pose of the “basketball” 4D point cloud dataset rendered via voxels at increasing level of detail (octree depth).

dataset	poses	points per pose (average)
baseball	115	88,315
basketball	114	87,003
kick	150	101,931
soccer (juggle)	116	92,352
soccer (shoot)	60	60,976

TABLE I: Characteristics of the investigated datasets.

High-resolution performance counters were used for CPU profiling, while OpenGL Timer Query objects were used for GPU profiling. Rendering performance was obtained by averaging individual frame rendering times at distinct view-points. Each viewpoint was rendered in 1920x1280 pixels of resolution, with 25% to 85% of the pixels being covered by shaded fragments. The rendering techniques described in this paper were also successfully tested on a slower computer equipped with an older NVIDIA GeForce 8600M GT (256MB) graphics card, but for brevity these results will not be listed.

Surfel rendering performance using the proposed two-pass enlarged surfel splatting algorithm of Section V-B is listed in Table II (2x splatting). The table shows the average frame rendering time, in milliseconds, for each of the instancing methods investigated in Section IV. From the results, the Geometry Shader approach was the fastest. An explanation for this is that the hardware may have a fast rendering path for offloading quadrilaterals from the Geometry Shader unit – one of its original design motivations was accelerating billboard-based particle effects. The use of regular vertex buffers was the slowest amongst the buffering methods offered by the hardware instancing pipeline. This is surprising because it is the method that more naturally shares and exposes per-instance attribute data between the application and the shader program. Another interesting fact is that uniform buffers performed faster than the others even though several draw calls were necessary.

dataset	Geometry Shader	Hardware Instancing Pipeline		
		uniform buffer	texture buffer	vertex buffer
baseball	4.518	7.461	7.873	13.11
basketball	3.673	6.387	6.782	12.15
kick	5.177	8.218	8.933	14.96
soccer (juggle)	4.921	7.843	8.236	14.40
soccer (shoot)	3.323	5.397	5.658	9.099

TABLE II: Average surfel rendering time (per-frame) using hardware-accelerated geometry amplification and instancing.

Structural characteristics, build time and traversal time of sparse voxel octrees are listed in Table III. The use of $\delta = 0\%$

dataset	δ	actual loss	octree			performance (ms)	
			depth	nodes	leaves	build	traverse
baseball	2%	0.65%	11	192,299	87,737	28.47	15.67
	10%	3.60%	10	107,161	85,137	21.21	11.72
	50%	32.6%	9	47,638	59,523	13.17	6.732
basketball	2%	0.71%	11	189,559	86,383	28.4	15.05
	10%	4.19%	10	106,198	83,361	19.34	11.60
	50%	32.7%	9	47,618	58,579	12.83	6.417
kick	2%	0.96%	11	201,762	100,948	33.77	16.22
	10%	8.15%	10	108,137	93,625	23.18	14.24
	50%	38.6%	9	45,586	62,551	16.2	7.458
soccer (juggle)	2%	0.61%	11	206,292	91,789	30.36	16.85
	10%	3.38%	10	117,056	89,235	21.08	12.73
	50%	30.9%	9	53,249	63,807	13.62	7.053
soccer (shoot)	2%	0.49%	11	141,737	60,676	20.08	10.55
	10%	2.88%	10	82,519	59,217	14.73	9.137
	50%	28.2%	9	38,717	43,802	10.07	5.076

TABLE III: Sparse Voxel Octree construction summary.

is not listed as it yields aggressive subdivision with several dozens, sometimes hundreds, of subdivision levels. At such levels, numeric precision can become a problem. In addition, too many nodes have to be allocated to hold a comparatively much lower number of actual leaf octants. Rendering is also unlikely to benefit from such extreme subdivisions since, at normal viewing stances, LOD traversal would halt at much coarser levels based on the screen-projection size criterion. The "actual loss" field in Table III represents the occupancy ratio $oct(h)/n$ of Section VI-A at the deepest octree level reached (the "depth" field) when the δ stop criterion was met. From the observed loss ratios, $\delta = 5\%$ reflects a good default threshold value suitable for the 4D media produced by the framework. Refer to Figures 6 and 7 (bottom) for illustrative examples of the effects of δ and the maximum depth of the octree.

Build and traversal of octrees requires CPU cycles, and their reported performance is based on a single-threaded execution context. Multi-threaded executions of these algorithms are possible, but not considered in this paper. The hierarchy can be constructed offline, in which case the octree can be serialized in breadth-first order using only 1 byte per node. Octree traversal is necessary when a new pose is to be rendered or whenever the viewpoint changes.

Voxel rasterization time is given in Table IV, for each of the instancing methods investigated in Section IV. Contrary to the surfel case, the Geometry Shader approach was the slowest. This reinforces the supposition that the hardware has a fast path for quadrilaterals in the Geometry Shader unit. For the case of a more complicated geometric profile (like a voxel's cube) the general-purpose hardware instancing pipeline performs better. As was the case with surfels, amongst the buffering strategies of the hardware instancing pipeline, uniform buffers were the fastest even with the increased draw call footprint, and regular vertex buffers were the slowest.

dataset	octree depth	Geometry Shader	Hardware Instancing		
			unif. buff.	tex. buff.	vertex buff.
baseball	10	3.594	2.639	2.784	5.788
basketball	10	3.454	2.549	2.773	5.599
kick	11	3.914	2.831	3.167	6.229
soccer (juggle)	10	3.763	2.742	2.961	6.083
soccer (shoot)	10	2.466	1.835	1.927	4.071

TABLE IV: Average voxel rendering time (per-frame) using hardware-accelerated geometry amplification and instancing.

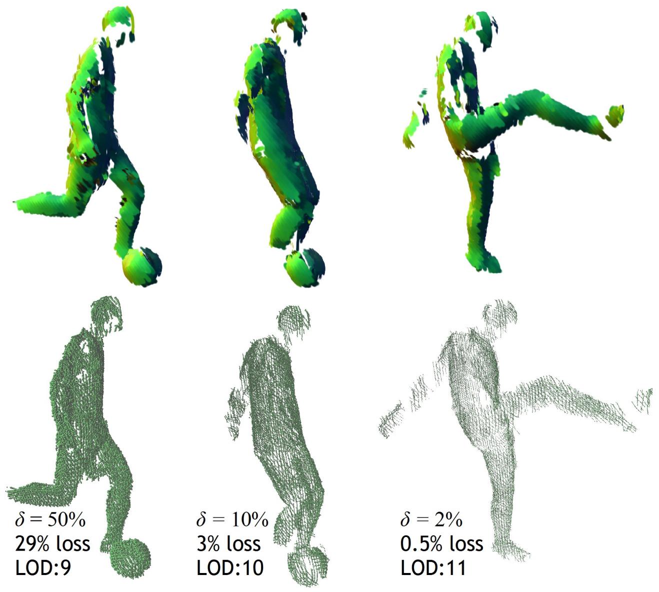


Fig. 7: Different poses of the "soccer (shoot)" 4D point cloud dataset rendered through surfels (top row) and voxels (bottom row). Distinct LOD were used for the voxel images.

VIII. Conclusion and Future Work

This paper established a framework for acquiring, compressing and displaying point clouds of moving objects scanned at high speeds. In particular, surfel and voxel rendering techniques for these point clouds were described. More specifically, modern graphics hardware features were exploited for real-time amplification and instancing of point primitives.

Due to surface reconstruction irregularities inherent in the framework, additional efforts were accounted during the design of the rendering system. One such efforts was the introduction of a novel surfel splatting strategy for the two-pass surfel splatting algorithm. Another accomplishment was the proposal of a simple yet compelling stop criterion for octree subdivision driven by raw point clouds.

Both the surfel and voxel rendering techniques described in this paper are effective for rendering the 4D media of the established framework. Sparse voxel octrees are specially attractive for allowing automatic LOD control and progressive rendering. The latter feature is notably useful for streaming data through a network, as is the case in telepresence applications. The downside is the additional processing time required for constructing and traversing the octrees.

As future work, normal and color information must be incorporated to the octree hierarchy. The use of splat size during octree construction must also be considered. Multi-threaded implementations of octree building and traversal algorithms also have to be investigated. Finally, octrees might become viable approach for compressing the 4D media by exploiting spatial and temporal coherence in the dataset. Entropy reduction techniques and prediction-based encoding schemes for octrees are also subject of further research.

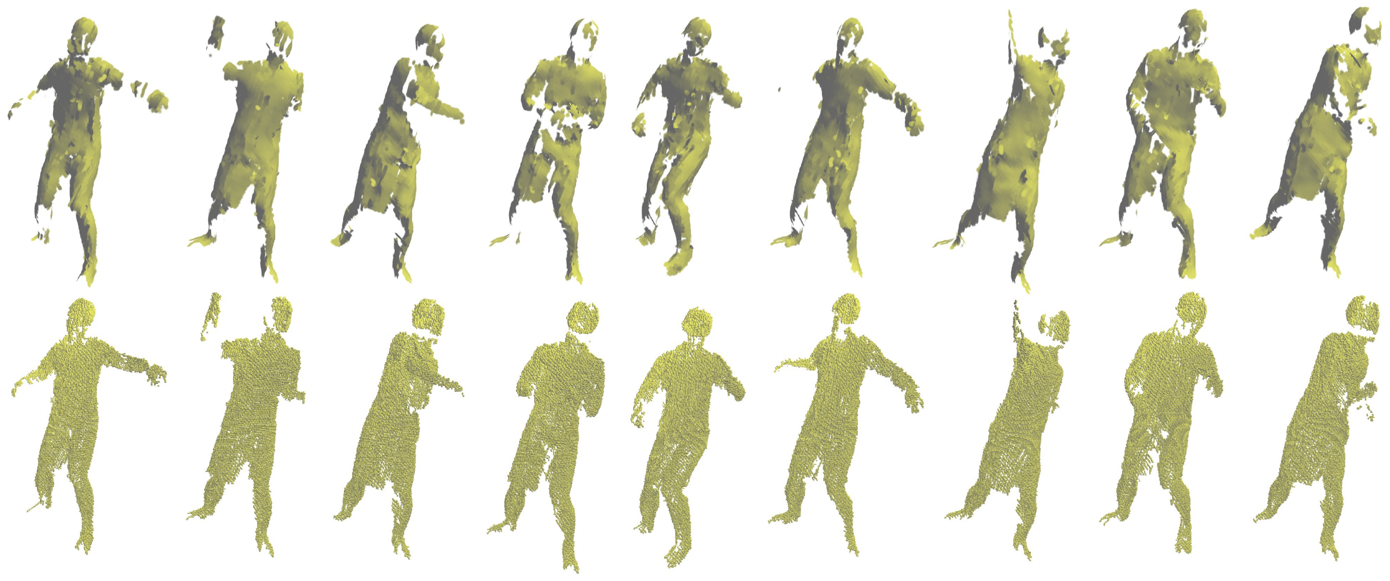


Fig. 8: Different poses of the "baseball" 4D point cloud dataset rendered through surfels (top row) and voxels (bottom row).

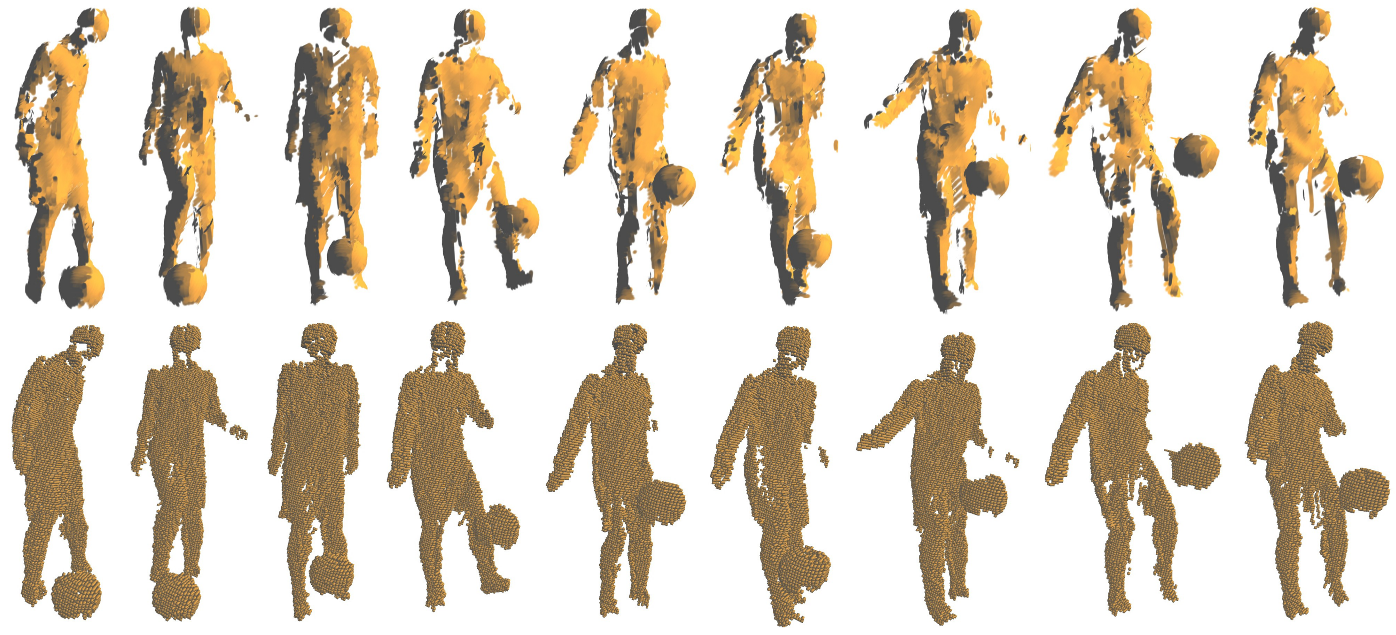


Fig. 9: Different poses of the "soccer (juggle)" 4D point cloud dataset rendered through surfels (top row) and voxels (bottom row).

ACKNOWLEDGMENT

This work was supported in part by SCOPE No.101710002 and NEXT program No.LR030 in Japan.

REFERENCES

[1] H. Pfister, M. Zwicker, J. van Baar, and M. Gross, "Surfels: surface elements as rendering primitives," 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. 335–342.

[2] P. W. C. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proceedings of the 1995 ACM SIGGRAPH symposium on Interactive 3D graphics*, ser. I3D '95. New York, NY, USA: ACM, 1995, pp. 95–ff.

[3] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "Surface splatting," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 371–378.

[4] L. Ren, H. Pfister, and M. Zwicker, "Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering," in *Computer Graphics Forum (Eurographics 2002)*, vol. 21, no. 3, Sep. 2002, pp. 461–470.

[5] G. Guennebaud and M. Paulin, "Efficient screen space approach for hardware accelerated surfel rendering." in *VMV*, T. Ertl, Ed. Aka GmbH, 2003, pp. 485–493.

[6] S. Yamazaki, R. Sagawa, H. Kawasaki, K. Ikeuchi, and M. Sakauchi, "Microfacet billboarding," in *Proceedings of the 13th Eurographics workshop on Rendering*, ser. EGRW '02. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 169–180.

[7] L. Kobbelt and M. Botsch, "A survey of point-based techniques in

computer graphics," *Computer and Graphics*, vol. 28, no. 6, pp. 801–814, Dec. 2004.

[8] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 55–63.

[9] R. Furukawa, R. Sagawa, H. Kawasaki, K. Sakashita, Y. Yagi, and N. Asada, "One-shot entire shape acquisition method using multiple projectors and cameras," in *Image and Video Technology (PSIVT), 2010 Fourth Pacific-Rim Symposium on*, nov. 2010, pp. 107 –114.

[10] I. Daribo, R. Furukawa, R. Sagawa, H. Kawasaki, S. Hiura, and N. Asada, "Efficient rate–distortion compression of dynamic point cloud for grid-pattern-based 3d scanning systems," *3D Research*, vol. 3, pp. 1–9, 2012.

[11] S. Laine and T. Karras, "Efficient sparse voxel octrees – analysis, extensions, and implementation," NVIDIA Corporation, NVIDIA Technical Report NVR-2010-001, Feb. 2010.