Vertex Transformation Streams

Youngmin Kim, Chang Ha Lee, and Amitabh Varshney

Email: {ymkim,chlee,varshney}@cs.umd.edu

Tel: +1-301-405-3132

Fax: +1-301-405-6707

Department of Computer Science

University of Maryland

College Park, MD 20742, USA

Abstract

Recent trends in parallel computer architecture strongly suggest the need to improve the arithmetic intensity (the compute to bandwidth ratio) for greater performance in time-critical applications, such as interactive 3D graphics. At the same time, advances in stream programming abstraction for graphics processors (GPUs) have enabled us to use parallel algorithm design methods for GPU programming. Inspired by these developments, this paper explores the interactions between multiple data streams to improve arithmetic intensity and address the input geometry bandwidth bottleneck for interactive 3D graphics applications. We introduce the idea of creating vertex and transformation streams that represent large point data sets via their interaction. We discuss how to factor such point datasets into a set of source vertices and transformation streams by identifying the most common translations amongst vertices. We accomplish this by identifying peaks in the cross-power spectrum of the dataset in the Fourier domain. We validate our approach by integrating it with a view-dependent point rendering system and show significant improvements in input geometry bandwidth requirements as well as rendering frame rates.

Key words: Stream programming, arithmetic intensity, geometry instancing, transformation encoding, streaming algorithms

1 Introduction

The recent evolution of graphics processing units (GPUs) into powerful and programmable stream processors is revolutionizing the way we look at the traditional graphics pipeline. Recent advances in the development of a stream programming environment for GPUs (1) and the use of stream programming for GPUs (2) have enabled graphics researchers and practitioners alike to view graphics operations in the context of data parallel semantics. The stream programming abstraction allows us to consider graphics primitives as streams of records and graphics operations as kernels that operate on such streams. This simple and yet compelling abstraction has not only had a powerful impact on graphics applications, it has also enabled a wide variety of applications from diverse disciplines such as scientific computing, machine learning, signal processing, computer vision, real-time audio, and computational biology to be mapped on to the GPUs. An important factor behind this success has been the power of the stream programming abstraction to embody, implicitly or explicitly, several important parallel algorithm design considerations such as data parallelism, task parallelism, coherence and latency of memory accesses, producer-consumer locality, and arithmetic intensity.

As the size of a floating-point unit on a 90 nm chip has decreased to almost 0.1% of its area, the challenge has gradually shifted away from trying to accommodate multiple processing units on a single chip towards maximizing the returns from the available bandwidth. In other words, arithmetic is cheap and bandwidth is the critical problem (3).

This paper presents a novel method to dramatically enhance the arithmetic in-

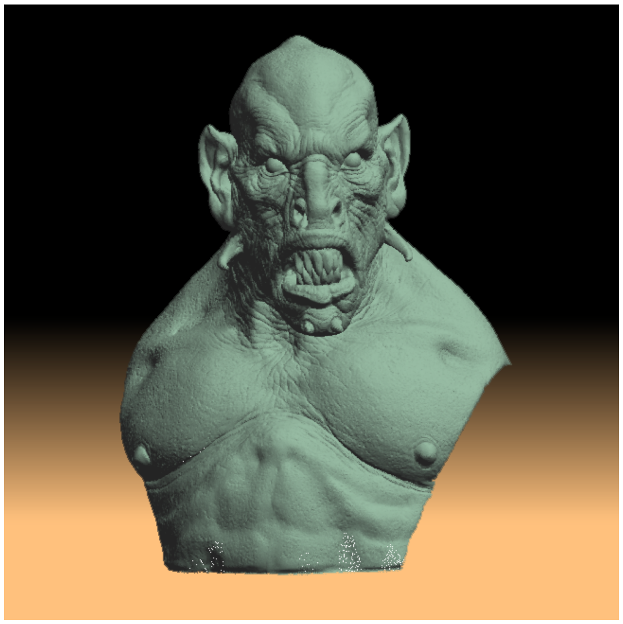


Fig. 1. *Rendering with Transformation Streams can dramatically improve the arithmetic intensity for conventional graphics rendering applications, such as view-dependent rendering. This image shows the Troll model rendered with our approach with 274% improvement in communication bandwidth and 20% improvement in frame rates.*

tensity (the compute to bandwidth ratio) (4), for vertex streams on the GPUs. The inspiration for our work lies in the idea that two interacting streams are significantly more powerful than a single stream. This basic idea has been around in computer architecture for a while (it was used in IBM 7950 as early as 1961 (5)) but its applications to graphics were not yet possible due to lack of programmatic and hardware support at the graphics processor level. The recent emergence of instance streams in modern GPUs (6) has allowed us to formulate and validate our ideas on representing geometry as two interacting streams of coordinates and transformations.

The main contributions of this paper are:

(1) **Interacting Streams:** We introduce the idea of interacting vertex and

transformation streams to encode general point cloud datasets and discuss how these streams can be decoded using modern vertex shaders.

(2) **Vertex-Transformation Pools:** We discuss how to efficiently build vertex and transformation streams from a pool of paired vertices and transformations.

(3) **Transformation Palettes:** We outline a method to identify the most common transformations that can map a set of vertices to itself using the Fast Fourier Transform.

(4) **View-dependent Transformation Streams:** We show how our approach of using transformation streams can improve the arithmetic intensity in a view-dependent rendering application.

2 Related Work

The graphics community has long faced the challenge of interactively exploring very large 3D graphics models while reconciling the mutually conflicting goals of scene realism and interactivity. A crucial bottleneck in this has been the input geometry bandwidth. Consequently, there has been a long history of work related to reduction of the geometry bandwidth to the graphics processor to achieve greater interactivity.

Triangle strips and triangle fans are amongst the earliest data-structures designed to address the input geometry bandwidth bottleneck. Although each triangle can be specified by three vertices, to maximize the use of the available data bandwidth it is desirable to order the triangles so that consecutive triangles share an edge. Such ordered sequences of triangles are referred to as triangle strips or triangle fans. Using such an ordering, only the incremental change of one vertex per triangle needs to be specified. These methods require sending $n + 2$ vertices for n triangles, instead of $3n$ vertices, potentially reducing the input geometry requirements by a factor of three.

Visibility-based culling and level-of-detail-based rendering reduce the input geometry by reducing the number of graphics primitives. Visibility-based culling schemes only send those primitives to the graphics processor that are visible or potentially visible (7). Level-of-detail-based rendering schemes send simpler, lower fidelity representations of an object whenever higher complexity representations are deemed unnecessary – such as when the object is being displayed on a small number of pixels on the screen or is otherwise perceptually less important (8).

Recent improvements in scene acquisition techniques such as laser scanning and computer-vision-based sensing have resulted in a growing collection of 3D graphics datasets that are based on points. Consequently, point-based rendering schemes (9; 10; 11; 12; 13; 14) have evolved as a viable alternative to triangle-based representations. The point primitives can be rendered as spheres (13), points with attributes (Surfels) (12), tangential disks (Surface splats) (15; 16; 17; 18; 19), tangential ellipses (20), quadratic surfaces (21), higher degree (3 or 4) polynomials (22; 23), using wavelet basis (24), and octree cells (12; 25; 26). These and several other techniques involve transmission of points and their attributes from the CPU to the GPU every frame. Points can also be rendered without any CPU involvement by storing the point geometry directly on the graphics card (17; 18; 27). Temporal coherence can be exploited by keeping track of the visible Surfels in the frame buffer of successive frames (28).

In some ways, the flavor of the approach presented in this paper is closer to that of triangle strips and triangle fans, that involve re-ordering the input list of primitives to succinctly represent them as a single data stream. However, it differs from all previous work in that it addresses the geometry bandwidth bottleneck by harnessing the power of multiple interacting streams of data, instead of a single stream. Our approach is complementary to, and can augment, existing schemes such as level-of-detail-based rendering that reduce the number of geometry primitives to be rendered.

3 Interacting Streams

Our goal in factoring an input list of vertices into two interacting streams of vertices and transformations is to reduce the input geometry bandwidth requirements and improve the arithmetic intensity of the participating streams. As shown in Figure 2, these two streams – the vertex stream and the transformation stream – can then be combined with each other on a GPU resulting in an output stream of vertices that is a tensor product of the input streams. Thus, in the best case, we might be able to factor n vertices into two streams of size $\sqrt{n}$ each, thereby reducing the input bandwidth requirements by a factor of $O(\sqrt{n})$.

Consider an idealized geometry of 16 points shown by spheres in Figure 3. Let the four white points comprise the vertex stream. Then using a set of four translations as shown in the figure, one can generate all the input points. We include the null translation $(0, 0, 0)$ for completeness.

The vertex transformation streams discussed above are ideal. In practice it

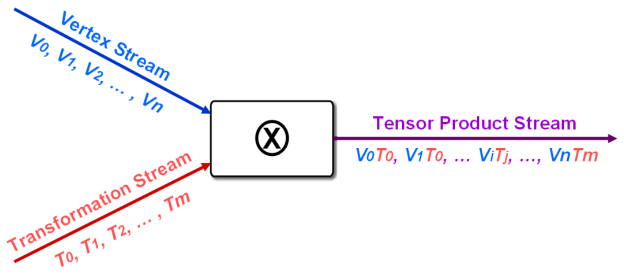


Fig. 2. *Stream Interactions: 'Vertex Stream' contains the mesh vertices, 'Transformation Stream' contains instance transformations that will act on the vertices in the vertex stream. The two streams are combined on the GPU and generate the 'Tensor Product Stream' which has the output vertices for rendering.*

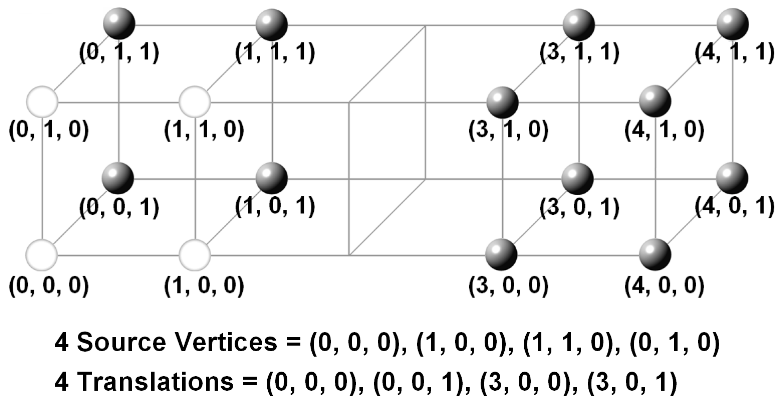


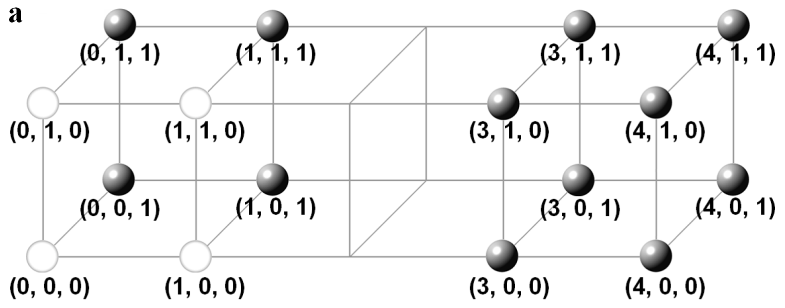
Fig. 3. *The vertex stream has the four white source vertices and the transformation stream has four translations. Each of the twelve black vertices can be reached by applying one of the three non-null translations to the white vertices.*

is rare to find such a perfect mapping between vertices and transformations. Even when we did find such mappings, they covered very small sets of vertices. To get larger interaction streams we decided to generalize our interactions between vertices and transformations. Instead of insisting that every vertex interacts with every transformation, we allowed some vertex transformation pairs to not contribute any vertex to the output stream. This simple generalization allowed us to greatly enhance the scope and size of the vertex transformation streams that our approach could identify. In Figure 4 we show the relationships between translations and vertices of two sets of geometries. A 1 indicates that the translation in the row interacts with the vertex of the column while a 0 indicates non-interaction. Our generalization allows us to construct interacting streams that can represent the slightly irregular geometry of Figure 4(b).

We have implemented stream interaction on current-generation GPUs using the geometry instancing features of the latest vertex shader model (`VS 3.0`). Consider the case of n vertices in stream 0 and m transformations in stream 1 as shown in Figure 2. To use the geometry instancing feature of (`VS 3.0`) we set the frequency of the vertex stream to m and the frequency of the transformation stream to 1. As shown in Figure 2, the vertex shader is first invoked with V_0T_0. This is followed by V_1T_0, V_2T_0, and so on. After all the vertices for the first transformation(T_0) have been processed, the pointer to vertex stream (stream 0) is reset and the pointer to transformation stream (stream 1) is incremented to T_1 (6).

In our implementation, we specify the frequency of the vertex stream to be the number of transformations in the transformation stream. Since the geometry instancing only allows us to achieve all-pairs interactions between the elements of the two streams, we encode the interaction information between the transformation and vertex streams using tags that we pass with the elements of each stream. The transformation stream's tag is just a short integer which indicates the index of that transformation in its stream. Thus the i^{th} transformation's tag has value i. A vertex's tag is an occupancy bit vector that encodes the translations that the vertex should interact with. Therefore, if we use an unsigned integer as the tag for a vertex, we can encode its interactions with up to 32 translations. Figure 5 shows the pseudo code of the vertex shader program for checking and implementing such interactions. Note that if

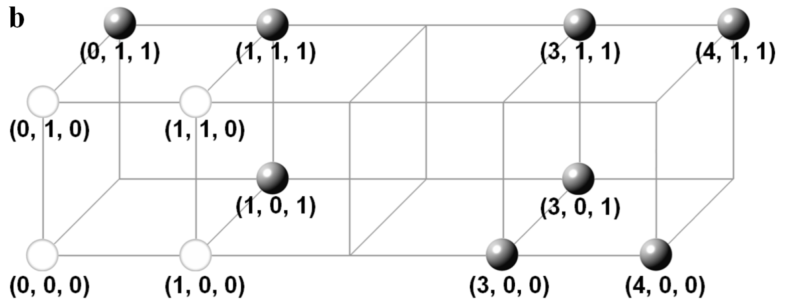
a



Translations \ Vertices	(0, 0, 0)	(1, 0, 0)	(1, 1, 0)	(0, 1, 0)
(0, 0, 0)	1	1	1	1
(0, 0, 1)	1	1	1	1
(3, 0, 0)	1	1	1	1
(3, 0, 1)	1	1	1	1

(a)

b

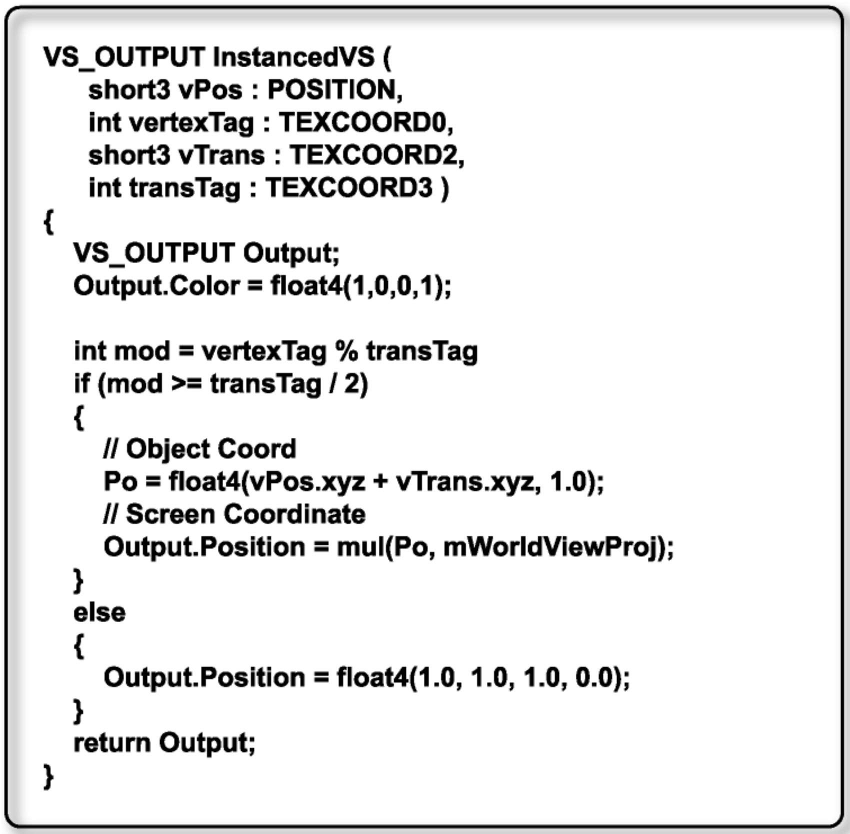


Translations \ Vertices	(0, 0, 0)	(1, 0, 0)	(1, 1, 0)	(0, 1, 0)
(0, 0, 0)	1	1	1	1
(0, 0, 1)	0	1	1	1
(3, 0, 0)	1	1	0	0
(3, 0, 1)	1	0	1	1

(b)

Fig. 4. *Interactions between the vertex stream and the transformation stream are represented by binary tables. In these vertex-transformation tables, a* 1 *indicates interaction between a vertex and a translation and* 0 *indicates no interaction. Figure (a) shows the table for the idealized point set of Figure 3. Figure (b) shows another point set and its vertex-transformation table.*

the interaction between a vertex and a transformation is not supposed to happen, our code sets the output vertex to infinity (in homogenous coordinates) and the vertex is then culled away. Unfortunately, the current generation of GPUs do not support bit-wise integer operations. As a result we had to emulate these bit operations by using floating-point operations that were available to us.


```
VS_OUTPUT InstancedVS (
     short3 vPos : POSITION,
     int vertexTag : TEXCOORD0,
     short3 vTrans : TEXCOORD2,
     int transTag : TEXCOORD3 )
{
   VS_OUTPUT Output;
   Output.Color = float4(1,0,0,1);

   int mod = vertexTag % transTag
   if (mod >= transTag / 2)
   {
      // Object Coord
      Po = float4(vPos.xyz + vTrans.xyz, 1.0);
      // Screen Coordinate
      Output.Position = mul(Po, mWorldViewProj);
   }
   else
   {
      Output.Position = float4(1.0, 1.0, 1.0, 0.0);
   }
   return Output;
}
```

Fig. 5. *Pseudo-code for a Vertex Shader program shows how we use vertex and transformation tags to determine if a pair of elements across vertex and transformation streams should interact, and if so how to generate the output stream vertices.*

4 Vertex Transformation Pools

In the previous section we discussed how we can generate an output stream of vertices as a tensor product of vertex and transformation streams. We also discussed how we can fine-tune (allow or disallow) interactions between the two streams by appropriately tagging the elements of the two streams and using vertex shader programs. In this section we shall discuss how to identify such streams from raw input point datasets.

Consider a model with n points, $\vec{p_i}$, $0 \leq i < n$. For any pair of points $\vec{p_i}$ and $\vec{p_j}$, $0 \leq i, j < n$, consider a transformation that maps $\vec{p_i}$ to $\vec{p_j}$. In general, there are an infinite number of such transformations. Therefore, let us restrict ourselves to the set of translations. Let translation $\vec{t_{ij}}$ be specified as $\vec{t_{ij}} = \vec{p_j} - \vec{p_i}$. For n points, we can identify n^2 such transformations. Now, for most real-life datasets we have observed that out of these n^2 transformations only $m << n^2$ are unique. We discuss some reasons for this transformation space coherence and give a method to identify such unique transformations in Section 5. Now consider a large *vertex transformation table* whose columns are n vertices and whose rows are the m unique transformations. The $(i, j)^{th}$ element of this table is a boolean value which is set to 1 iff $\vec{t_i} + \vec{p_j} = \vec{p_k}$ for some $k < n$. That is, if the i^{th} translation maps the j^{th} vertex to some other vertex in the input data, we flag this as an interaction we permit. Otherwise we set the $(i, j)^{th}$ entry's boolean value to 0. If $\vec{t_i} + \vec{p_j} = \vec{p_k}$, we say that point $\vec{p_j}$ *covers* point $\vec{p_k}$.

In order to get large vertex and transformation streams, our goal is to find the largest rectangle in the vertex-transformation matrix (after reordering rows and columns) such that the fraction of ones in it is higher than some threshold (δ). We refer to such a maximal reordered rectangle as a *vertex transformation pool*. Large values of δ tend to return very small vertex-transformation pools whereas small values of δ result in too many non-interacting vertex-transformation pairs which will later require culling. We have observed that $\delta = 0.5$ reconciles these goals well. Identification of the vertex transformation pools is an iterative process. After we find a vertex transformation pool, we zero-out its entries in the vertex transformation table, and repeat the process to get the next-best vertex transformation pool.

4.1 Finding the First Vertex Transformation Pool

At first glance, the problem of finding good vertex transformation pools resembles the edge-maximizing bipartite clique problem (29; 30), where the rows are one side of the graph, the columns are the other, and there is an edge between i and j if the $(i, j)^{th}$ entry's boolean value is 1. However, our problem is different because we would also like to include some 0 entries in the pool as long as it allows us to increase the overall fraction of 1's in the pool. A polynomial-time optimal solution to this problem seems unlikely, so we have used a greedy heuristic.

We first find the vertex $\vec{p}_j$ that has the most 1-values in its column. Let the number of transformations for $\vec{p}_j$ be k. We next restrict ourselves to the k rows for which the column for $\vec{p}_j$ has a 1. We then sort all the vertices by the sum of 1-values they have in these k rows and process them in the decreasing order of their sums. For each vertex $\vec{p}_i$, we determine the number of vertices it can cover that were previously not covered. If the number of newly covered vertices is greater than a threshold, we include $\vec{p}_i$ in the pool. In all our experiments we have set the threshold to be 25% of the value of k for the current pool. We have observed that this gives us vertex-transformation pools with the fraction of ones in the pool, $\delta \approx 0.5$.

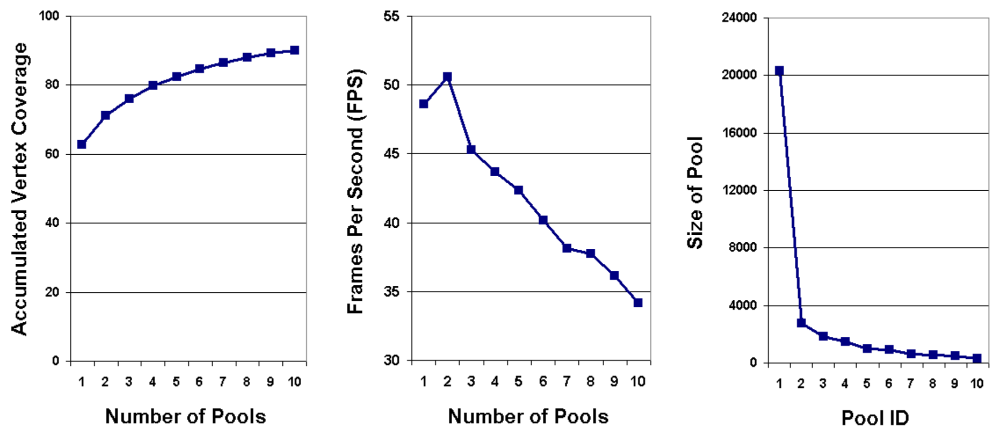
It turns out that there can be redundant coverage amongst the vertices in a pool. Thus, it is possible that $\vec{t}_i + \vec{p}_j = \vec{t}_{i'} + \vec{p}_{j'} = \vec{p}_k$. This is wasteful in that we might end up processing the same vertex multiple times. We handle this by not counting such previously covered vertices in our quest to maximize the size of our vertex transformation pools.

4.2 Updating for Subsequent Pools

After identifying a vertex transformation pool we update the vertex transformation table to discard the covered vertices. Since we prioritize the vertices based on the number of their 1-entries, the number of vertices that a given vertex covers is actually its importance level for inclusion in future pools. For each vertex $\vec{p}_k$ that our identified pool covers, we decrease the weights of all other vertices that could cover $\vec{p}_k$. Thus, for each covered vertex $\vec{p}_k$ we set all those (i, j) entries in the vertex transformation table to 0, for which $\vec{t}_i + \vec{p}_j = \vec{p}_k$. We can do this updating efficiently if each vertex maintains a list of all other vertices that it can cover. Thus, let $\vec{p}_k$ have the list of vertices $(\vec{p}_{k1}, \vec{p}_{k2}, \ldots, \vec{p}_{kl})$ that can be reached from itself using its translations $(\vec{t}_{k1}, \vec{t}_{k2}, \ldots, \vec{t}_{kl})$. Each of these covered vertices $\vec{p}_{ki}$ must have the reverse translation $(-\vec{t}_{ki})$ in its own translation lists. Therefore, we can decrease the importance of those covered vertices by one by just resetting the appropriate elements in the vertex transformation table.

4.3 Deciding the Number of Pools

To show how the number of pools affects the overall performance of this algorithm, we have plotted the effect of the number of pools against the vertex coverage and the final rendering time for the Stanford's David in Figure 6.



(a) (b) (c)

Fig. 6. *Tradeoffs between the number of pools, vertex coverage, and rendering time for Stanford's David model. Figure (a) shows the vertex coverage and figure (b) shows the rendering time as we increase the number of pools. Figure (c) shows the changes of the size of each vertex transformation pool, which is the product of the number of translations and the number of vertices in each pool.*

As we increase the number of vertex transformation pools, they can cover more vertices, but the overall rendering performance drops. There are two reasons for this. First, using additional blocks incurs the overhead of using `drawCall( )`. We have observed that using too many `drawCall( )`s makes the applications CPU bound. Second, as we identify more pools, the size of each pool, which is the product of the translations and the vertices in it, gets smaller. For small pools, the overhead of using instancing for drawing outweighs the benefit of transferring less data to the GPU. Based on these considerations, we decided to select only the first two pools for our results. The vertices that are not covered by these two pools are rendered without any geometry instancing using conventional rendering.

5 Transformation Palettes

We have thus far discussed how to identify maximally-sized vertex-transformation pools and how to convert them into interacting vertex and transformation streams. In this section we discuss how to identify the most common transformations amongst vertices. As we discussed earlier in Section 4, if we consider a model with n points, $\vec{p}_i$, $0 \leq i < n$ and restrict ourselves to translations, it is possible to get n^2 transformations amongst all pairs of points. In the worst case, each of these n^2 transformations can be unique and we will not be able to benefit from a transformation-based coding of the input data. Fortunately, real data does have plenty of such coherence due to several factors – input data symmetries, coherence in procedural or simulation data generation, coherence due to acquisition device characteristics, and even coherence due to quantization algorithms. In a number of real-life datasets, we observe that certain local geometries may appear in the same configuration in another part of the model. These symmetries are obvious for architectural CAD (repeated doors, windows, furniture), mechanical CAD (repeated sub-assemblies, bolts, cylinders), molecular CAD (repeated amino acids, nucleic acids, alpha helices, beta sheets), and terrain layouts for games (trees, grass, flowers). Our algorithm can easily detect such symmetries in the transformation space. What is more interesting, however, is that for several cases, such repeated patterns might not even be visually obvious. We have observed that a large fraction of 3D point geometry representing real-life datasets from 3D scanners can be efficiently expressed in this way. In Figure 7 we show one example of a frequent translation our algorithm discovered in the Stanford's David model.

The problem of finding the set of two point pairs which specify the same

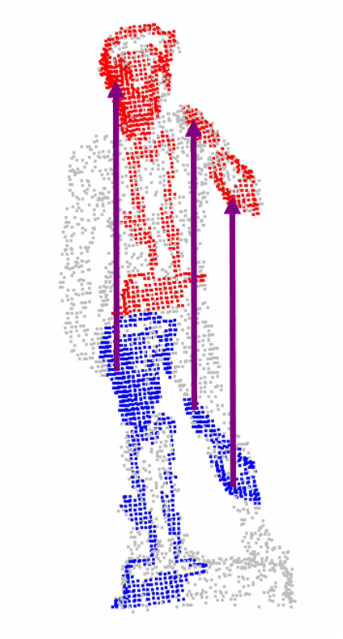


Fig. 7. *An example of frequently occurring common translations of vertices identified by our algorithm in Stanford's David model. If we quantize the David model on a* 128^3 *grid to* $150K$ *quantized points we find the translation* $(62, -6, -5)$ *ocurring* 1261 *times as shown.*

translation $\vec{t}$ can be reduced from 3SUM problem. The 3SUM problem can be solved by a simple $O(n^2)$ algorithm (31) and recent advances present a subquadratic randomized algorithm (32). Using a finite precision model, a 3SUM hard problem can be solved in $O(n \log n)$ time using the FFT (Fast Fourier Transform) (33). Here we used the FFT to find the common translations efficiently.

We explain our method in the simplified one-dimensional case. For a point set $P = \{p_0, p_1, p_2, ..., p_{N-1}\}$, $0 \leq p_i \leq M$, we can think of a polynomial $A(x)$ where the exponent of each term is the coordinate of each point, and we can think of another polynomial $B(x)$ where the exponent of each term is the negative value of the exponent of $A(x)$.

$$A(x) = \sum_{i=0}^{M} a_i x^i, \text{ where } a_i = \begin{cases} 1, \text{ if } i \in P \\ \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

$$B(x) = \sum_{i=0}^{M} a_i x^{-i}, \text{ where } a_i = \begin{cases} 1, \text{ if } i \in P \\ \\ 0, \text{ otherwise} \end{cases} \tag{2}$$

Let $C(x)$ be the polynomial representing the multiplication of polynomials $A(x)$ and $B(x)$. The exponent of each term in $C(x)$ can be interpreted as the translation between two points in P, and the coefficient of that term indicates the frequency of occurrence of the translation. In other words, c_i is the number of points in P which can be translated from other points in P by translation i.

$$C(x) = \sum_{i=-M}^{M} c_i x^i \tag{3}$$

The multiplication of two polynomials can be computed in $O(n \lg n)$ time by converting polynomials into *point-value representations* using the FFT, and then creating the coefficient representation of the multiplication of two point-value polynomials using the inverse FFT (33). Because $B(x)$ is the transposed image of $A(x)$, the FFT of $C(x)$ can be computed by transposing the multiplication of the FFT of $A(x)$ by the complex conjugate of the FFT of $A(x)$. For 3D points, we use 3D FFT with the same algorithm as for the 1D case. We note that this is same as computing the cross-power spectrum in the Fourier domain, a technique widely used for image registration (34).

We have implemented our method using FFTW package (35) for computing the FFT and the inverse FFT, and tested this on several 3D datasets. We are currently using FFTW since we are dealing with quantized values of points that is guided by the precision necessary for the view-dependent rendering application discussed in the next section. After 3D FFT stage, we identify the most common transformations and label them as the *transformation palette*. We currently identify the most common 256 translations and use them in building the vertex-transformation tables and interacting streams of vertices and transformations.

6 View-dependent Rendering with Transformation Streams

View-dependent rendering has introduced the concept of rendering different regions of a scene at varying detail based on their perceptual significance. Our view-dependent rendering algorithm is similar to the ones generally used for triangle meshes (8) and points. We first build a binary hierarchy over the input points by following a principal-component-analysis (PCA)-based partitioning (36). The PCA of a set of n points in a 3D space gives us the mean μ, an orthogonal frame f, and the standard deviation σ of the data. The terms μ and σ are 3D vectors and we refer to their i-th component as μ^i and σ^i respectively, where $\sigma^i \geq \sigma^j$ if $i > j$. The frame f consists of three 3D vectors with the i-th vector referred to as f^i.

The *distortion* of a partitioning is defined as the sum of the distances of the points from the partition's mean (36). In our partitioning scheme we reduce this distortion by using k-means clustering with $k = 2$. We initialize the two starting means (centers) for the k-means algorithm by doing a PCA over the points and choosing $\mu_p + \frac{\sigma_p^1}{2} f_p^1$ and $\mu_p - \frac{\sigma_p^1}{2} f_p^1$ as the initial guesses. This is a

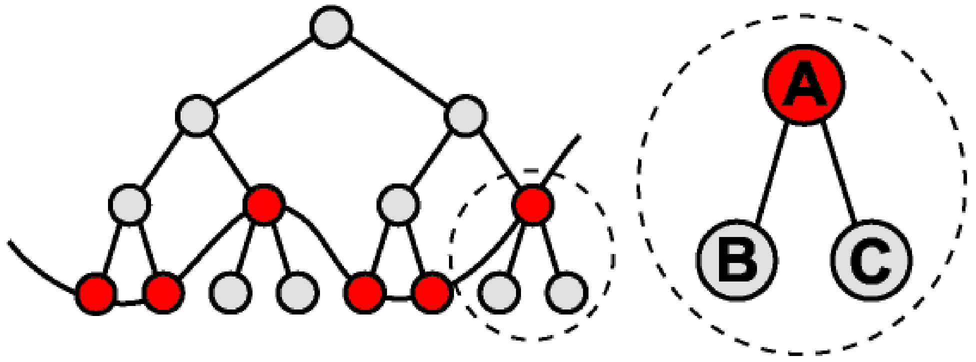


Fig. 8. *View Dependent Rendering. The entire tree is a complete binary tree. Red nodes were cut in view-dependent manner. Node 'A' can be expanded into two nodes, 'B' and 'C' for the fine details.*

reasonable assumption since the data varies maximally along f_p^1. The k-means clustering algorithm then iterates over the twin steps of partitioning the point set according to the proximity of each point to the two means and then updating the two means according to this partitioning. (37) use a geometric way to separate the point set for their point-based simplification hierarchy. They separate along the principal direction f_p^1 with the separating plane passing through the mean μ_p. Their approach is equivalent to the first iteration of our clustering scheme. Subsequent iterations then successively reduce the distortion. We stop iterating when the difference in the distortion between two successive iteration is less than 10^{-7} or when the number of iterations is more than 30, whichever happens earlier.

Next, for each node in the binary tree we carry out the steps mentioned earlier in the paper – we identify the most common transformations appropriate for that node, we build vertex-transformation pools, and identify the transformation vertex streams. At the end of this pre-processing step we store the transformation vertex streams with each node in the binary hierarchy.

At run time we maintain an active list of nodes representing a cut in this binary

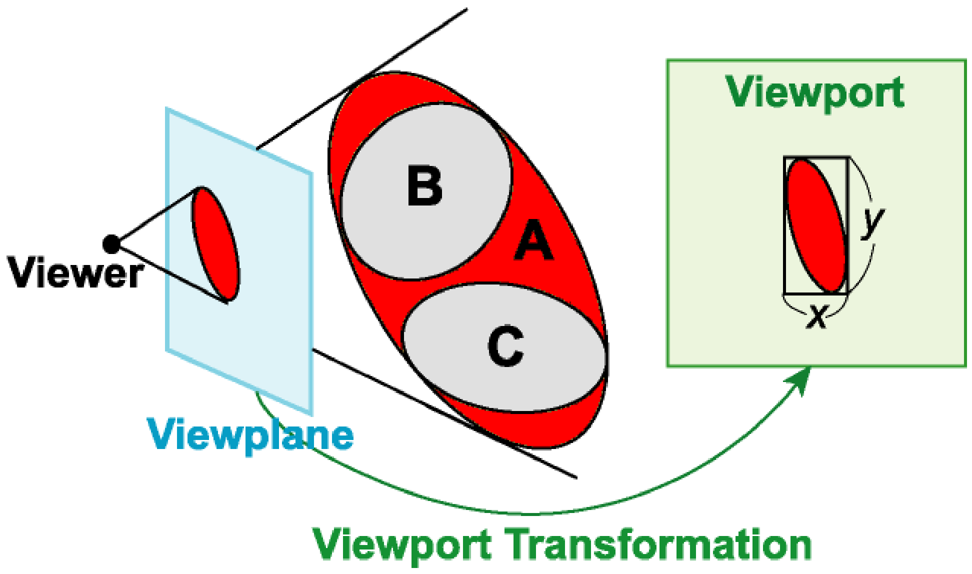


Fig. 9. *Expansion of a node. Consider the node A in Figure 8. We compute how many pixels the bounding box of the node A occupies when it is projected to the window and use it to decide whether to split to its children B and C or not.*

hierarchy as shown in Figure 8. The points from the active nodes comprise a level of detail appropriate to a given view. We merge sibling nodes if they project to a screen-space area below a threshold and split a parent node into its children if the projected screen-space area is too large. We use the resolution of the projected screen-space area of a node to guide the quantization of the vertices in that node. Thus if a node projects onto a screen-space area no larger than $\frac{128}{\sqrt{2}} \times \frac{128}{\sqrt{2}}$ pixels, a 7-bit vertex quantization along each of x, y, and z axes suffices. While rendering we transform these 7-bit quantized values using the μ, f, and σ values for the node to locate them in the appropriate 3-space.

For each frame we sequentially visit each of the active nodes and decide whether for the given view parameters it will be appropriate to render it directly, or merge it with its siblings, or refine it to its children. Once an appropriate level of detail for a node has been finalized, we use the vertex transformation streams associated with that node to render the points contained in that node.

7 Results

We have validated the results of our approach to efficiently identify and use interacting streams of vertex and transformation data on a number of 3D graphics models. We have run our experiments on a 1.6 GHz Pentium IV Windows PC with 2 GB RAM with a NVIDIA GeForce 6800 Ultra AGP graphics card. We have used the geometry instancing hardware feature in vertex shader 3.0 model and used `DrawIndexedPrimitive( )` command in DirectX 9.0 API.

We have compared our results along two dimensions of performance – the improvement in CPU-GPU communication bandwidth and the improvement in the frame rates. We have measured the frame rates under a constant GPU memory usage model. In this model we allocate a fixed amount of memory on the GPU for rendering using our transformation streams method and using the conventional point rendering method.

Let us assume there are n vertices and m translations in a vertex transformation pool and f is the fraction of the vertices that are actually displayed from the pool (the fraction of unique vertices covered by the pool without redundancy). Then, the pool covers $f \times n \times m$ vertices. Assume we need A_v bytes for a vertex and A_t bytes for a translation in our transformation streams model, and A_n bytes for a vertex in the traditional point rendering model. Our transformation streams model will therefore use $(n \times A_v + m \times A_t)$ bytes instead of the traditional model's ($f \times n \times m \times A_n$) bytes.

In our experiments $A_v = 26$, $A_t = 12$, and $A_n = 8$. Here A_v is the sum of the bytes for indexing (2 bytes), the number of bytes for a vertex coordinates (8 bytes), and the number of bytes for vertex tagging (16 bytes). A_t is the sum of the number of bytes for translation coordinates (8 bytes) and the number of bytes for translation tagging (4 bytes). `SHORT4` data type is the most compact representation we can use in DirectX to represent vertices and translations, even though we only needed 3 shorts (6 bytes). For a fair comparison between our transformation streams model and the traditional model, we did not exploit the extra short for tagging purposes. The reason we need so many bytes for tagging purposes is because the current-generation GPUs do not support bit-operations for integers in vertex shaders. Therefore, in our experiments we limited ourselves to four floating-point tags per vertex. In each 4-byte floating-point tag we used 22 bits of the mantissa as a fixed-point integer. This allowed us to represent up to 88 translations in a vertex-transformation pool. We decided to limit ourselves to four floating-point vertex tags because of two reasons. First, increasing the number of floats costs more in bandwidth to the vertex shader. Second, using more than four floats required us to differentiate amongst multiple inputs to the vertex shader, thereby adding one more conditional branch in addition to the one shown in Figure 5. If in future we are allowed to pass 32-bit unsigned integers directly to vertex shaders we can save 4 bytes for A_v while at the same time increase the number of translations covered to 96. In Section 3, we had proposed using 16-bit translation tags. The reason why we instead use two shorts (8 bytes) in our implementation is that we currently pass the remainder and the quotient for the modulo operation in the vertex shader (Figure 5) to reduce the number of floor and

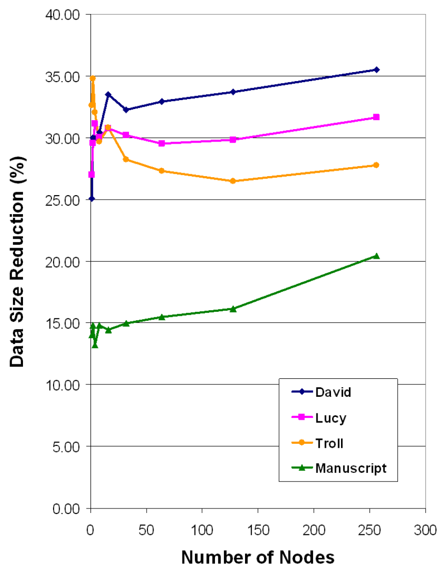


Fig. 10. *Communication bandwidth improvement for CPU-GPU transmission for transformation streams model compared to traditional point rendering.*

divide operations.

To give you a flavor of our results, our approach achieved $n = 150$, $m = 88$ (the limit due to four floats discussed above), and $f = 0.32$ for Stanford's David model. Figure 10 shows the data transmission gains from using transformation streams as we increase the number of nodes in the hierarchy. We used a 8-level hierarchy for view-dependent rendering. The gains shown include the overhead of sending singleton vertices that our transformation streams model could not cover.

For comparison of frame rates between the transformation streams model and the traditional point rendering model, we used a fixed amount of GPU memory. Let A_{pools} be the amount of data used by the vertex transformation pools in the transformation streams model. For both models we draw A_{pools} amount of data from GPU vertex buffers and the rest from conventional memory.

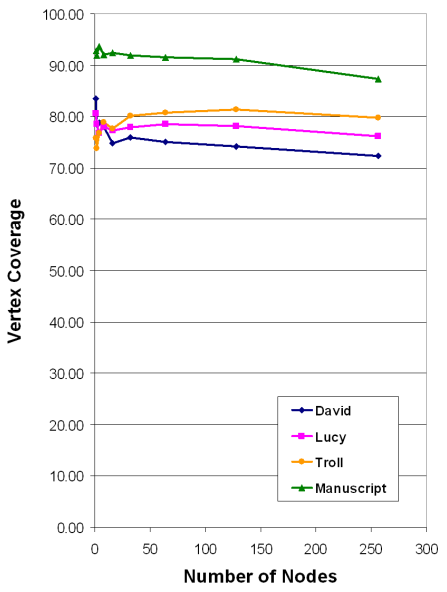
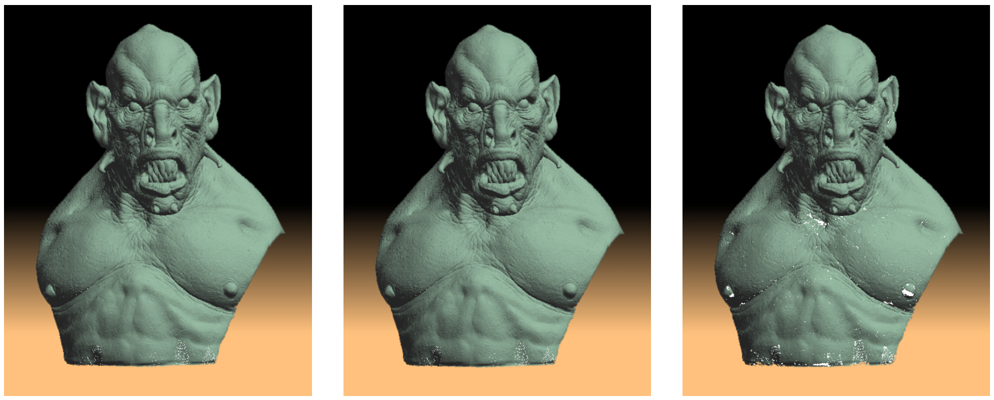


Fig. 11. *Coverage of vertices by transformation streams model when using two vertex transformation pools.*

Our method currently processes geometry without appearance attributes. An easy way to draw models with appearance attributes using our method is to use texturing. The use of texturing results in only a 10% overhead with our method. Visual results of our approach are shown in Figure 12, 13, 14. We have achieved 200% to 500% improvement in communication bandwidth to the GPU and 17% to 32% improvement in frame rates. The left and the center columns of Figures 12, 13, 14 show the conventional rendering and the rendering by Transformation Streams, respectively. As shown in Figure 11, two vertex transformation pools can cover about 80% of all the vertices. The right column of Figures 12, 13, 14 shows the rendering of the vertices covered by the vertex-transformation pools.



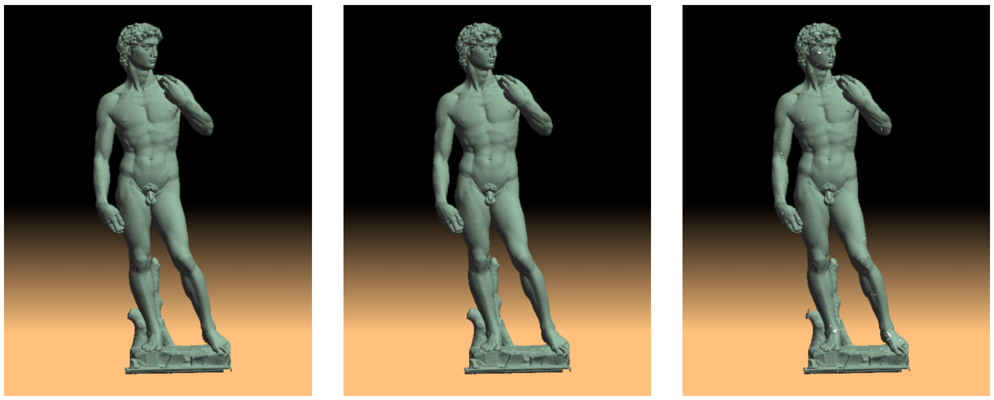
(a) Conventional	(b) TS	(c) Coverage by pools
1.69M Verts	1.69M Verts	1.37M Verts
20.3FPS	24.3FPS	N/A
12.92MB	3.45MB	997KB

Fig. 12. *The left image shows the conventional rendering, the center image shows the rendering by transformation streams and the right image shows the vertices covered by the vertex-transformation pools for the Troll model. We report the number of vertices rendering, the frame rates achieved, and the per-frame communication bandwidth required for the conventional approach and our approach.*

8 Conclusions and Future Work

We have proposed a novel method for representing 3D point data using interacting streams of vertices and transformations. We have validated this method for accelerating conventional view-dependent rendering applications for points.

Although our method achieves a factor of two to five improvement in the communication requirements to the GPU, our frame rates do not improve by a similar factor. As the graphics community engages in image synthesis for ever larger 3D point datasets and as the gap between processing speeds and



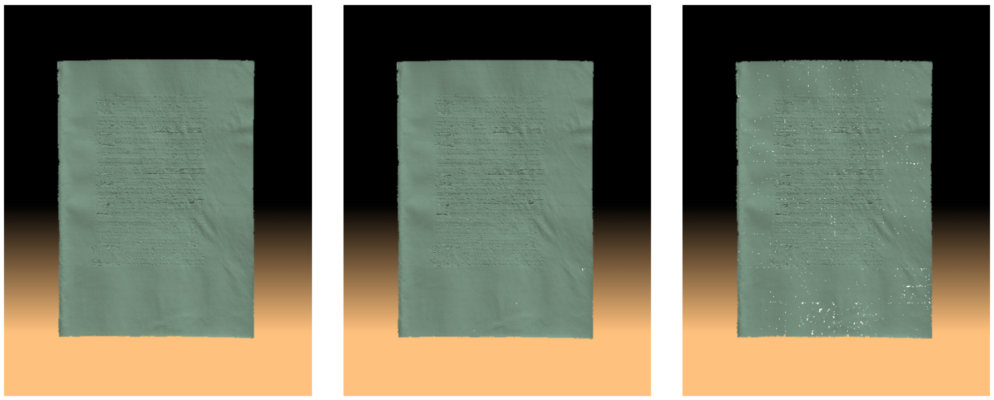
(a) Conventional	(b) TS	(c) Coverage by pools
1.17M Verts	1.17M Verts	878K Verts
29.8FPS	35.0FPS	N/A
8.90MB	2.92MB	712KB

Fig. 13. *The result of rendering Stanford's David. The numbers have the same meaning as Figure 12.*

memory access times grows ever wider, the impact of our method on graphics rendering performance should rise even further.

One of the important considerations in our method is the space required to encode the boolean interaction matrix. Recent advances in efficiently compressing boolean matrices (38) are relevant to such encodings and suggest a fruitful direction for future research. At present the GPU programmability and the geometry instancing framework offer limited flexibility in exploring sophisticated boolean interaction matrix compression techniques. Still, such compression techniques could greatly assist in remote visualization applications.

We hope that our approach of transformation streams will provide a road-



(a) Conventional	(b) TS	(c) Coverage by pools
1.02M Verts	1.02M Verts	926K Verts
32.9FPS	43.4FPS	N/A
7.75MB	1.25MB	565KB

Fig. 14. *The result of rendering XYZ RGB's Manuscript. The numbers have the same meaning as Figure 12.*

map for future research into how one can use multiple interacting streams in the stream-programming abstraction to map other problems of interest on the GPUs. Our current results appear promising for combining stream programming abstractions with traditional procedural graphics approaches. We believe these are first steps towards more general combinations of multiple data streams for processing geometry, appearance, and physical simulations.

Acknowledgements

We would like to thank the reviewers for several helpful comments and suggestions that have greatly helped in making our submission better. We would also like to acknowledge several valuable discussions we have had on this topic with David Mount, Aravind Kalaiah, Zhiyun Li, and Xuejun Hao. This work has been supported in part by NSF grants: CCF 04-29753, CNS 04-03313, and CCF 05-41120.

References

[1] I. Buck, T. Foley, D. Horn, J. S. K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Transactions on Graphics 23 (3) (2004) 777–786.

[2] P. Hanrahan, Stream programming environments, in: GP2: 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, 2004, pp. A–4.

[3] B. Dally, Stream processors vs. GPUs, in: GP2: 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, 2004, pp. A–13.

[4] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, I. Buck, Merrimac: Supercomputing with streams, in: SC, 2003, p. 35.

[5] G. A. Blaauw, F. P. Brooks, Jr., Computer architecture: concepts and evolution, Addison-Wesley, Reading, MA, USA, 1997.

[6] B. Dudash, Mesh instancing, Tech. Rep. 00000-001-v00, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050 (2004).

[7] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, F. Durand, A survey of visibility for walkthrough applications, IEEE Transactions on Visualization and Computer Graphics 9 (3) (2003) 412–431.

[8] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, R. Huebner, Level of Detail for 3D Graphics, Morgan Kaufman, 2003.

[9] J. P. Grossman, W. J. Dally, Point sample rendering, in: Rendering Tech-

niques '98, Eurographics, Springer-Verlag Wien New York, 1998, pp. 181–192.

[10] M. Levoy, T. Whitted, The use of points as a display primitive, in: Technical Report 85-022, Computer Science Department, UNC, Chapel Hill, 1985.

[11] D. Lischinski, A. Rappoport, Image-based rendering for non-diffuse synthetic scenes, in: G. Drettakis, N. Max (Eds.), Rendering Techniques '98, Eurographics, Springer-Verlag Wien New York, 1998, pp. 301–314.

[12] H. Pfister, M. Zwicker, J. van Baar, M. Gross, Surfels: Surface elements as rendering primitives, in: Proceedings of SIGGRAPH 2000, ACM Press / ACM SIGGRAPH, 2000, pp. 335–342.

[13] S. Rusinkiewicz, M. Levoy, QSplat: A multiresolution point rendering system for large meshes, in: Proceedings of SIGGRAPH 2000, ACM Press / ACM SIGGRAPH, 2000, pp. 343–352.

[14] J. Shade, S. Gortler, L. He, R. Szeliski, Layered depth images, in: Proceedings of SIGGRAPH 98, ACM Press / ACM SIGGRAPH, 1998, pp. 231–242.

[15] M. Zwicker, H. Pfister, J. van Baar, M. Gross, Surface splatting, in: Proceedings of SIGGRAPH 2001, ACM Press / ACM SIGGRAPH, 2001, pp. 371–378.

[16] L. Ren, H. Pfister, M. Zwicker, Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering, in: Eurographics'02, 2002, pp. 461–470.

[17] M. Botsch, L. Kobbelt, High-quality point-based rendering on modern GPUs, in: Pacific Conference on Computer Graphics and Applications, 2003, p. 335.

[18] G. Guennebaud, M. Paulin, Efficient screen space approach for hardware

accelerated surfel rendering, in: T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, R. Westermann (Eds.), Proceedings of the Conference on Vision, Modeling and Visualization 2003 (VMV-03), IEEE Signal Processing Society, Berlin, 2003, pp. 485–494.

[19] R. Pajarola, Efficient level-of-details for point based rendering, in: Proceedings IASTED Computer Graphics and Imaging Conference (CGIM), 2003, pp. 141–146.

[20] J. Wu, L. Kobbelt, Optimized sub-sampling of point sets for surface splatting, Vol. 23, 2004, pp. 643–652.

[21] A. Kalaiah, A. Varshney, Modeling and rendering points with local geometry, IEEE Transactions on Visualization and Computer Graphics 9 (1) (2003) 30–42.

[22] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, C. Silva, D. Levin, Point set surfaces, in: IEEE Visualization 2001, 2001, pp. 21–28.

[23] S. Fleishman, D. Cohen-Or, M. Alexa, C. T. Silva, Progressive point set surfaces, ACM Transactions on Graphics 22 (4) (2003) 997–1011.

[24] T. Welsh, K. Mueller, A frequency-sensitive point hierarchy for images and volumes, in: IEEE Visualization'03, 2003, pp. 425–432.

[25] J. C. Woolley, D. Luebke, B. Watson, Interruptible rendering, in: SIGGRAPH'02 Technical Sketch, ACM Press / ACM SIGGRAPH, 2002, p. 205.

[26] M. Botsch, A. Wiratanaya, L. Kobbelt, Efficient high quality rendering of point sampled geometry, in: S. Gibson, P. Debevec (Eds.), Proceedings of the 13th Eurographics Workshop on Rendering (RENDERING TECHNIQUES-02), Eurographics Association, 2002, pp. 53–64.

[27] C. Dachsbacher, C. Vogelgsang, M. Stamminger, Sequential point trees, ACM Transactions on Graphics 22 (3) (2003) 657–662.

[28] G. Guennebaud, L. Barthe, M. Paulin, Deferred Splatting, Computer Graphics Forum 23 (3) (2004) 653–660.

[29] R. Peeters, The maximum edge biclique problem is NP-complete, Discrete Applied Mathematics 131 (3) (2003) 651–654.

[30] S. Khot, Ruling out PTAS for graph min-bisection, densest subgraph and bipartite clique, in: IEEE Symposium on Foundations of Computer Science (FOCS), 2004, pp. 136–145.

[31] A. Gajentaan, M. H. Overmars, On a class of O(n^2) problems in computational geometry, Computational Geometry Theory Applications 5 (1995) 165–185.

[32] I. Baran, E. D. Demaine, M. Patrascu, Subquadratic algorithms for 3SUM, in: WADS, 2005, pp. 409–421.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press, 1996.

[34] B. S. Reddy, B. N. Chatterji, An FFT-based technique for translation, rotation, and scale-invariant image registration, IEEE Transactions on Image Processing 5 (8) (1996) 1266–1271.

[35] M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: ICASSP conference proceedings, Vol. 3, 1998, pp. 1381–1384.

[36] R. O. Duda, P. E. Hart, D. G. Stork, Pattern Classification, 2nd Edition, John Wiley & Sons, Inc., New York, 2001.

[37] M. Pauly, M. H. Gross, L. Kobbelt, Efficient simplification of point-sampled surfaces, in: IEEE Visualization, 2002, pp. 163–170.

[38] D. S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, S. Venkatasubramanian, Compressing large boolean matrices using reordering techniques, in: International Conference on Very Large Data Bases (VLDB), 2004, pp. 13–23.