Representation of Real-life 3D Models by Spatial Patches

Denis V. Ivanov, Yevgeniy P. Kuzmin
Department of Mathematics and Mechanics, Moscow State University
Moscow, Russia

Abstract

The commonly used solution for real-life 3D model representation is polygonal spatially consistent geometry, with texture, and, optionally, bump or displacement maps attached. Although the idea of displacement mapping is well known, there are just a few approaches to its efficient implementation. In this paper we present a technique that allows for efficient representation and rendering of real-life 3D models by getting a new angle on the displacement mapping concept. We introduce a new primitive that is defined as the range image of a small part of the model's surface; therefore, it is called a spatial patch. The whole model is just a collection of patches with no connectivity information between them. Such a representation can be directly acquired by 3D scanning machinery, and stored in a compact uniform form. It also allows for efficient visualization. In this paper we present some aspects of spatial patch rendering technique that utilize conventional z-buffer and benefit from modern features of computing units. We also discuss our experience in representing 3D models by spatial patches, provide some practical results and set up directions for future work. Our preliminary evaluation of the technique makes us believe that spatial patch technology can be efficiently used in a wide range of applications dealing with real-life 3D data.

Keywords: *Rendering Primitive, Displacement Mapping, Spatial Patch, and 3D Model Representation*

1. INTRODUCTION

Three-dimensional computer graphics has recently become ubiquitous at the consumer level due to the creation of affordable 3D hardware accelerators. These accelerators, ranging from high-end workstations to low-priced boards, are used for efficient visualization of triangles, which have been traditionally considered as a basic rendering primitive. Being very simple in shape, triangles seem to have met the right balance between descriptive capabilities and computational burden. Thus, the exponential growth of computing power, observed by Gordon Moore, is expected to allow for the rendering of 3D models of much better quality at the same speed. However, processing scenes consisting of a large number of small triangles leads to certain problems, such as bandwidth bottlenecks and excessive floating-point requirements [3]. A number of techniques have been developed in order to overcome these limitations.

Texture mapping, originally proposed by Catmull [2], is now supported by most of 3D engines. This technique makes it possible to use larger and fewer triangles by filling polygon interiors with colors taken from an attached image, called a texture. Although this strategy allows for rendering more naturally looking models at interactive speeds, it works satisfactorily only for flat or slightly curved surfaces, such as walls and tables, and often fails representing complex shapes. For example, many objects in real-time games, which are considered the "killer application" for 3D graphics, exhibit noticeable artifacts revealing their polygonal nature.

In order to simulate the roughness of polygonal surfaces, a technique called bump-mapping can be applied. Bump-mapping, invented by Blinn [1], does not change the underlying geometry of a model, but produces shading effects as if the polygons were wrinkled. Obviously, additional computational efforts are required to realize these effects, while they only seem to be helpful for small variations of the surface that is observed from the close to normal direction. However, the silhouette of a model remains unchanged; besides, the human vision system, having stereoscopic capabilities, is likely to recognize the proposed deception at close distances.

Further improvement of bump-mapping leads to performing actual displacement of surface elements in addition to appropriate shading computation. Although this idea is simple and well known, there can be found quite a few implementations in literature. The proposed solutions are based on volumetric textures [8], ray casting [9] or remeshing of the initial geometry [4]. The latter technique seems to be the most amenable to hardware acceleration; however, its output is a collection of smaller triangles forming dense meshes based on original polygons.

Obviously, polygonal representation of model geometry is not the only possible solution. There exist a large number of others, including implicit surfaces, NURBS, or subdivision surfaces, which still require additional knowledge about material and texture for realistic rendering. Besides, higher order primitives are usually decomposed into triangles before being visualized by graphics hardware.

Another important thing to discuss with respect to surface-based representation is modeling. Since we usually require that computer models look very much like real objects, accurate acquisition of real-life 3D data is likely to be involved. Several scanning technologies have recently been developed to the level that allows archiving quite sufficient results [7]. Commercially available 3D scanners, based on time-of-flight, structured light, and other principles, produce point clouds or, more commonly, regular depth fields with spatial resolution as small as fractions of millimeter. Attached with a camera or 1D color sensor, this machinery is capable of generating range images (images with depth information for every pixel) at reasonable speed. Unfortunately, further registration of multiple scans and generation of a spatially connected, surface-based representation, which is required for efficient rendering, takes a much longer time and often degrades the acquired data.

Range images possess several important qualities. First, being directly acquired by 3D scanning systems, they seem to be quite natural for representing fragments of a surface. Second, the corresponding data can be stored in a uniform manner, unlike today's practice where textures, bump maps, and geometry information are usually stored separately. Finally, range images have regular structure, which makes it possible to apply various optimization techniques during rendering. Thus, they are amenable to acquisition, storage, and efficient rendering.

Guided by these considerations, in [5] we first proposed a new primitive for 3D model representation and rendering. This primitive, called a spatial patch, is defined as a range image of a small part of the surface taken in the direction close to the normal. In this context, the whole model is just a collection of spatial patches. No connectivity information between neighbor primitives is required for such a representation.

In [5] we described in details one of the approaches to rendering of spatial patches. This approach utilizes conventional z-buffering strategy and is designed to benefit from advanced features of modern processing units, including “single instruction multiple data” (SIMD) operations and on-chip caches. In this paper we highlight some of the important aspects of the proposed rendering technique.

Since we had no 3D scanning machinery at our disposal, we generated spatial patches from models represented by other means, such as polygonal meshes and point clouds. In this paper we also introduce some ideas that we used for development of the conversion algorithms and discuss our experience in operating with 3D models represented by spatial patches.

Our practical results showed that significantly magnified views of models represented by spatial patches might experience some artifacts in the areas where neighbor patches overlap. However, the proposed representation proved to work very well for rendering with screen resolution comparable to resolution of range images treated as spatial patches.

2. DEFINITIONS

The concept of range image is well known in computer graphics literature. It is usually defined as a raster image attached with per-pixel distances from the viewing point to the surface. A spatial patch is defined in the same manner as follows.

Given the model orthonormal coordinate system, a **spatial patch** is defined by the origin point P, orthogonal frame ($\Delta x,\Delta y,\Delta z$), and a rectangular $m\times n$ array of $(c,d)_{i,j}$ pairs, each representing a point $N_{i,j}=P+i\Delta x+j\Delta y+d_{i,j}\Delta z$ of color $c_{i,j}$ on a surface (Figure 1).

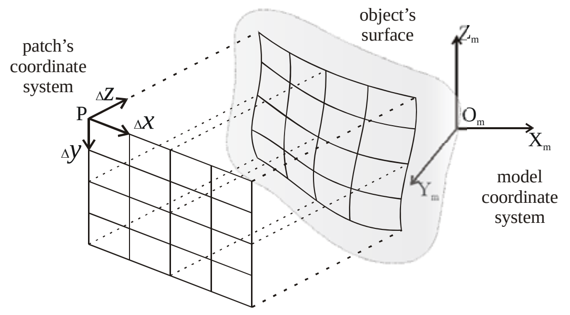


Figure 1: Spatial Patch definition

It is important that all patches of a model are defined in one coordinate system, referred to as model space. An orthogonal frame ($\Delta x,\Delta y,\Delta z$) of each patch corresponds to its local coordinate system, called patch space. We also define world space with the origin placed in the eye and z-axis parallel to the viewing direction, and screen space as the result of the perspective projection. The world-screen transformation maps the viewing frustum into an axis-oriented cube with the center at (0,0,0) and sides equal to 2, as it is suggested in OpenGL [10].

In practice, we use the standard RGB888 format for color representation and signed 8-bit integers for displacements, which, in total, result in a 32-bit storage requirement for a node. Such a low displacement precision makes sense if patches are relatively small compared to the model; thus, only small surface variations are expected. Pure black color is reserved for transparent nodes, including those whose displacement falls out of the [-128,127] range. If more accurate geometry or an alpha channel is required we use RGBA (8 bits per channel) and 32-bit displacements, thereby doubling the storage size. The origin point and the frame of a patch are stored separately using floating-point numbers.

3. RENDERING WITH Z-BUFFER

In [5] we presented the logical structure of the rendering unit that renders spatial patches generating appropriate data for z-buffer. It should be mentioned that patches can also be rendered by means of the standard graphics pipeline if they are considered as regular meshes. This solution would involve neither texture- nor bump-mapping, but would produce a large number of small triangles, which usually leads to bottlenecks and high computational requirements. Having this option in mind, we may think of the proposed rendering unit as an extension to the commonly used pipeline. As using the traditional rendering pipeline is optional it is shown in gray on Figure 2.

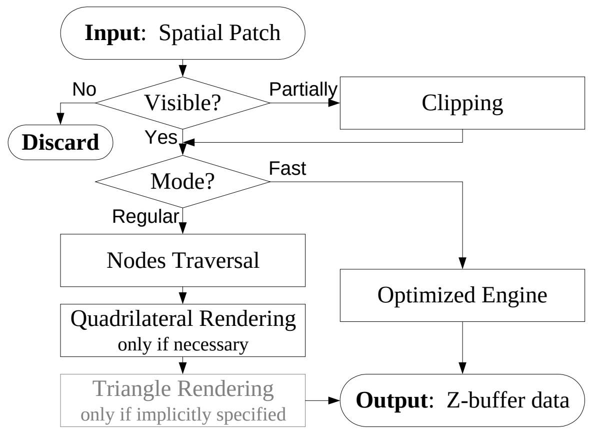


Figure 2: Logical structure of rendering unit

The rendering unit (Figure 2) is designed as a sequence of stages, which are executed sequentially to visualize spatial patches on a displaying device.

In the first stage, the unit verifies the visibility of the spatial patch, and discards it if the answer is negative. This is done by checking the relative position of a bounding solid, i.e. box or sphere, and the visible frustum. If partial visibility is determined, the patch is recursively clipped by the frustum, and the residual parts are considered new smaller patches. However, we would like this condition to happen as rarely as possible because clipping is relatively expensive operation; thus, we use the common technique of allowing patches to extend outside the viewing region by several percent, which significantly decreases the probability of ‘partially visible’ event.

When the patch is guaranteed to lie within the visible frustum, the decision is made on whether or not optimized techniques can be used to render it, or whether a regular sequence of required procedures should be executed. The importance of this decision is based on the fact that under certain conditions some of the regular procedures can be omitted without any influence to the result. Thus, having detected these conditions, the rendering unit can process the patch several times faster.

The regular procedures of patch rendering include two major stages: nodes traversal and cell rendering. As the rendering unit is designed to reproduce the original geometry of a patch, it should find the projections of all nodes onto the screen. Seemed to be computationally expensive from the first sight, this operation can be efficiently implemented in practice exploiting the patch's properties of being small and having regular structure. These properties make it possible to find perspective projection of the patch's bounding box and approximate node positions by trilinear interpolation between its corners. Applying this strategy drastically reduces the number of arithmetic operations by omitting per-node perspective projection. The price that is paid for speed increase is absence of perspective correctness within the patch; however, for patches that are much smaller in size than the visible frustum the distortion is not perceptible.

The node traversal stage ends up with screen coordinates of each node, which can be directly placed into the z-buffer. However, the question arises: What happens between the nodes? Obviously, if magnification occurs, we expect to see gaps between visualized pixels. Considering the patch as regular mesh, the task, then, is to render its cells provided that the nodes have been already rendered. The common solution would be to divide each quadrilateral into two triangles, and flush them into the standard pipeline. Leaving quality issues of this strategy aside, we run into dealing with large number of small triangles, which leads to high computational costs. In order to render the produced quadrilaterals efficiently and more naturally from visual point of view we proposed in [5] to apply the commonly used strategy of subdividing the geometry recursively until the desired level of accuracy is reached. There were proposed two schemes of quadrilateral subdivision. The first one finds bilinear interpolation between 4 cell corners, while the other one constructs smoother surface that is C^1-continuos over a patch. The bilinear scheme requires no more than 1 addition and 1 bitwise shift per each new node, while the smooth one requires about three times this number of operations. However, for significantly magnified view the C^1-continuos interpolation leads to more good-looking surfaces.

Shading can be implemented utilizing either Phong or Gouraud approaches - the rendering unit can light patch's nodes and interpolate final colors within each cell, or it can interpolate normals (bilinearly or smoothly) and perform per-pixel shading. As reconstruction of a normal vector is a relatively expensive operation, which involves cross product and normalization, it was proposed to reconstruct normals in local orthogonal patch space where much fewer arithmetic computations are required due to patch regularity. In this case, the light sources should be converted to patch space in order to light nodes properly.

The other technique that can significantly reduce the number of computations required for normal reconstruction is buffering of normal vectors once they are computed and reusing them during further rendering. The fact that all patches of a model are likely to have similar or equal $|\Delta z|/|\Delta x|$ and $|\Delta z|/|\Delta y|$ ratios increases the probability for each normal to be computed many times, which ensures a gain in performance. The price for this gain is the allocation of memory, which can be as small as 512 Kb since internal symmetries and some reduction in precision can be exploited efficiently. If patches representing a model have various configuration, which does not allow reuse of normal vectors, they can be divided into groups considering similarities of their $|\Delta z|/|\Delta x|$ and $|\Delta z|/|\Delta y|$ ratios. Then, normals buffering can be applied within each group separately.

The technique that was proposed for optimized engine is based on the determination of the fact that no magnification of patch's cells occurs during rendering. In such event, no quadrilateral interpolation stage is executed. However, regular algorithm verifies this condition for each cell separately, which requires per-cell computations. On the other hand, it appeared to be relatively cheap to detect those patches that do not require cell interpolation on a per-patch basis. As the node traversal stage can also be optimized for such patches they can be efficiently rendered by a specially designed procedure.

It should be mentioned that the proposed rendering strategy for spatial patches can be implemented using parallelism on different levels. The SIMD (Single Instruction Multiple Data) approach can be applied not only to coordinate triples or quads, but also to the whole node data in many cases. Thus, for example, coordinates and colors may be interpolated in the same manner in the quadrilateral rendering stage. Significant increase in performance is expected if additions and bitwise shifts on 6-tuples or 12-tuples are executed in parallel by a single instruction. Besides, two parallel computing units can be used for quadrilateral subdivision process.

From the global point of view, each spatial patch is a separate object that is expected to be relatively small. This property makes it possible to divide the frame buffer into several rectangular areas, often called chunks, and provide each one with a separate rendering unit. This approach is highly efficient if patch clipping caused by internal partitioning occurs rarely. Introducing small internal guard bands can help a lot in this case.

Thus, the proposed rendering unit appears to efficiently exploit the advantageous properties of spatial patches and, in practice, proved to work quite well in simulators. More details on the presented above techniques can be found in [5].

4. MODELS GENERATION

We had no 3D scanning machinery at our disposal; therefore, we had to produce spatial patches by other means. As textured polygonal 3D models are the most convenient for dealing with, they were chosen as the major source data for production of spatial patches.

There exist two general approaches to acquisition of spatial patches from the models represented by their surfaces. The first approach utilizes ray-tracing strategy in order to produce range images; the second is based on surface analysis and selection of the areas that are most suitable for conversion to spatial patches. Although implementation of ray-tracing seems to be more natural for simulation of range finders, it provides no control over resulting range image (spatial patch), which may have unpredicted discontinuities and gaps. Thus, our choice was in favor of the second approach, which, in our opinion, could result in more optimal partitioning into patches by utilizing connectivity information of mesh elements. In retrospect, this strategy appeared to have certain disadvantages, as well.

As we also wished to convert some models represented by point clouds, which have no explicitly defined surface, we implemented a simple algorithm based on production of range images from them. Some details of this approach are discussed in Section 4.5.

4.1 Conversion Algorithm

The idea of the conversion algorithm is partitioning the surface of a model, which is defined by a triangular connected mesh, into fragments that can be represented efficiently by spatial patches.

The algorithm is implemented in three stages, which are creation of preliminary areas, creation of secondary areas, and merging. On the first stage an arbitrary face is selected and used as a seed element from which the corresponding area is grown. Areas are grown by adding adjacent faces iteratively, on each step selecting the one whose normal deviate least from the normals of already selected faces. The growing process stops if there is no adjacent face such that the adjusted set of normals lies within a solid cone of predefined size. By applying this procedure the whole surface is partitioned into connected, relatively smooth submeshes.

These submeshes, then, can be converted to patches, since the corresponding surfaces can be uniquely projected onto a plane provided that deviation threshold for normals is set to a small value. However, these areas usually have very complex blot-like shape, and converting them into patches would result in significant overlaps and data overuse.

In order to get areas of better shape, the central face of each preliminary submesh is used as a seed element for the secondary partitioning stage. On this stage the areas are grown simultaneously yielding to a more convenient shapes.

In practice, we observed that better result may be obtained if relatively small deviation threshold for normals is used on the first two stages, and resulting small areas are then merged using analogous strategy.

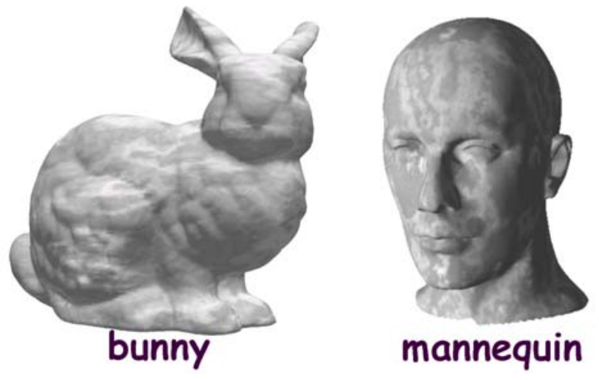


Figure 3: The 2 models represented by spatial patches

Using the described above algorithm, we converted large number of polygonal models into spatial patches. Among them were the well-known bunny, provided by Stanford University, and the model of head mannequin, provided by Hugues Hoppe from Microsoft Research (see Figure 3). Both models were not initially textured, so we applied some kind of stone texture using standard mapping utilities. Some statistics on the original and the produced geometrical data is present in Table 1.

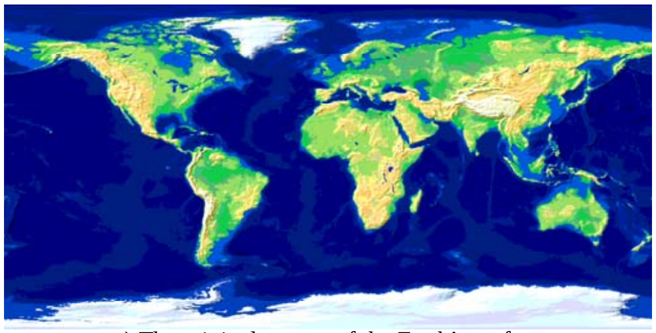
Table 1: Statistics on the bunny and mannequin models

		bunny	**mannequin**
mesh	Number of nodes	34 834	40 289
	Number of triangles	69 451	80 448
s.patches	Number of patches	1 526	1 145
	Number of nodes	441 014	330 905
	Data size (Mb)	1.83	1.37

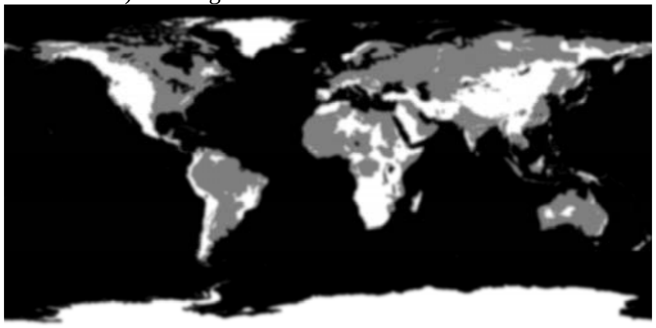
4.2 Generation of the Earth Model

In this section we describe the process that was implemented in order to generate the model of the Earth. We had no real data of the Earth's relief; therefore, we decided to make the model more expressive to demonstrate the advantages of spatial patches. It should be mentioned that the resulting relief is just a broad-brush approximation of the Earth's surface.

The first step was generation of the height map for the Earth's surface. We decided to estimate height from color of surface elements. The ocean, which has dominating blue color, was kept flat. The offsets of the land parts were divided into several levels depending on the hue of the corresponding color – the green areas were treated as plateaus, the brown ones as mountains. The obtained height field was smoothened by standard image processor, which treated height values as gray shades. The result of this operation is shown on Figure 4 (black areas correspond to the sea level, gray shades represent different heights above the sea level).



a) The original texture of the Earth's surface

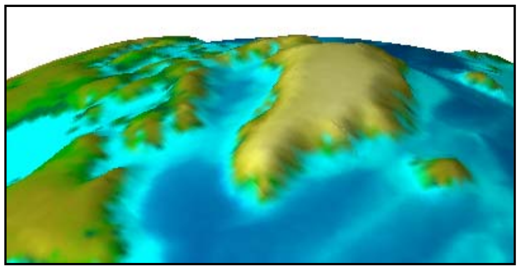


b) Produced height map

Figure 4: **Estimated height field of the Earth's surface**

After the height map had been generated, the appropriate polygonal model could be produced. The starting point was a connected triangular mesh of about 40.000 triangles of similar size which represented a complete surface of a sphere (Figure 5a). Each node of this mesh was assigned with texture coordinates according to the standard cylindrical projection of a rectangular bitmap to a spherical surface (the texture on Figure 4a gave some clues for cylindrical nature of the required projection). Then, each node was displaced along the corresponding normal vector to the value defined by its texture coordinates and the height map. Bilinear interpolation between the 4 nearest height map elements was used to produce the final displacement.

The resulting mesh along with the corresponding texture was provided to the conversion algorithm described in Section 4.1. The 3 models of different resolution were produced in order to evaluate descriptive power of spatial patches with respect to their density. The algorithm was set to extract patches of 17×17 nodes in size, and the resulting models consisted of 383, 1128, and 2712 separate patches. Figure 5b shows the model of 1128 patches visualized with the rendering engine described in Section 189. More close view of the top of the globe, which demonstrates that the land parts are actually raised above the sea level, is shown in Figure 5c.


c) Zoomed in part of the surface

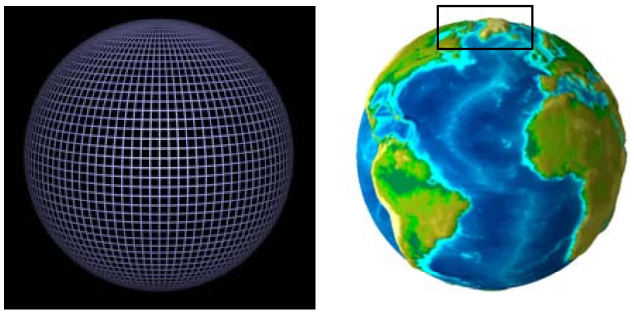

a) Mesh of a spherical surface b) Produced spatial patches

Figure 5: The model of the Earth

Table 2 summarizes some statistics on the produced models. While the original triangular mesh of a sphere consists of about 40.000 nodes, the produced patches have several times greater number of nodes in total. One may say that neighbor patches should overlap in order to represent consistent surface; however, the produced models are expected to demonstrate more dense geometry. The bitmap that was used as a texture for the surface is 1024×512 pixels in size, which requires 1.5 Mb for True Color 24bpp representation. As many pixels were actually projected to single nodes in polar areas of the globe, Earth_1128 model seems to comprise nearly all the initial data from visual point of view. In practice, we could not find any noticeable difference between the original model and Earth_1128 rendered from reasonable viewpoints.

Table 2: Statistics on the Earth models

Name	# of patches	# of nodes	Data size (Mb)
Earth_383	383	110 687	0.45
Earth_1128	1 128	325 992	1.35
Earth_2712	2 712	783 768	3.17

4.3 The Perfect Sphere

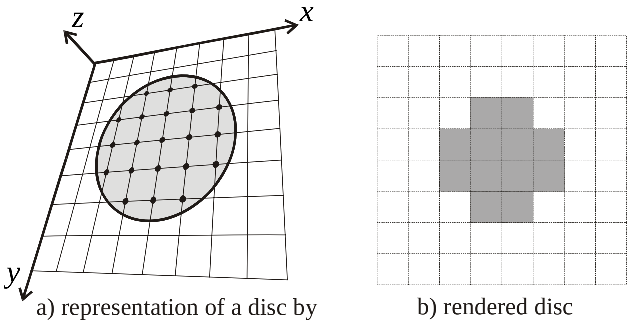
The concept of spatial patches was proposed for efficient representation of real-life bumpy objects; however, it has enough descriptive power for accurate representation of artificial surfaces, as well. To prove this fact, we generated a collection of spatial patches corresponding to a complete spherical surface of radius equal to 128 units. This model was called a ‘perfect sphere’.

To build up the model, we selected 1362 almost evenly distributed points on the surface utilizing spherical coordinates. At each point we then reconstructed a spatial patch, which had z-axis collinear to the normal to the sphere, and the lengths of the local frame vectors ($\Delta x, \Delta y, \Delta z$) equal to 1, 1, and 0.01, respectively. Displacing the nodes according to the formula of a sphere, we obtained a model of a complete surface.

The displacement precision in the patches is 0.01; therefore, it is expected that being rendered on a screen of about 25000×25000 pixels in size, the model would look as if it had been visualized using the explicit definition. The displacements are represented with 8 bits, and the patch’s frame can be defined by 40 bytes (10 floating point numbers), which, in total, result in less then 440 Kb of data. The conventional triangular mesh representation with such accuracy would take much more space to be stored.

4.4 Holes on Sharp Edges

The problem that we ran into while representing several models by spatial patches is possible appearance of small gaps between two surfaces having common curved edge. This problem has the same nature as aliasing effect for raster shapes. Indeed, given a flat disc in model space (Figure 6a), the corresponding spatial patch will naturally have *z*-axis collinear to disc’s normal, and the nodes whose (*x*,*y*) coordinates lie within the disc will be set to the corresponding value, while all other nodes will be left transparent. During rendering the quadrilaterals that have at least one transparent node are not rendered resulting in aliasing effects on edges (Figure 6b). This effect can be partially dealt with by interpolating alpha channel within semi-transparent quadrilaterals or by increasing patch resolution; however, the perfect circular shape cannot be represented.



a) representation of a disc by spatial patch b) rendered disc (low resolution, no alpha)

Figure 6: Accuracy of shape representation

Unfortunately, the described above effect takes place in a large number of artificial models originally represented by polygons. The simplest one is the cylinder shown on Figure 7. Increasing patch density to the level of screen resolution would remove the holes; however, there would be a huge number of wasted nodes lying on the flat parts of the model in this case. The proposed solution that we used to overcome the problem is detection of the sharp edges and generation of small patches that have intermediate orientation with respect to the faces adjacent in the selected edges. These patches should be of higher resolution compared to the other ones since they represent more complex geometry. The additional patches and the corrected model of a cylinder are shown on Figure 8.

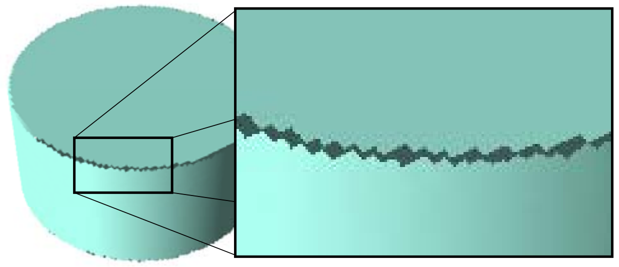
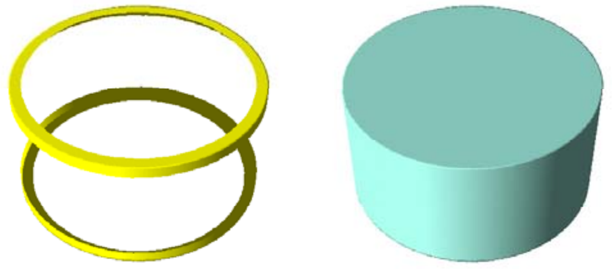


Figure 7: Holes on sharp edges



a) additional patches on edges b) the resulting model

Figure 8: The corrected model of a cylinder

4.5 Dealing with point clouds

Point clouds do not have explicitly defined surfaces, and building surfaces from them seems to be a complicated task; therefore, the conversion algorithm discussed in Section 4.1 cannot be applied to such representation. For this reason, we implemented simple strategy that is based on acquisition of large range images from the cloud. The algorithm produces several range images from the angles usually evenly distributed on a unit sphere, and extracts rectangular regions that can be efficiently represented by spatial patches. Production of a range image in this case is implemented by flushing all points into z-buffer.

As the source data is a collection of separate points, its image in z-buffer may have certain gaps. We tried to setup parameters of the rendering viewport so that the appearing gaps were not larger then 1-2 pixels, and then filled the gaps by interpolating the *z* coordinate of neighbor elements. In addition, it was sometimes useful to apply some kind of smoothening filters to the range data stored in *z*-buffer since point clouds acquired by range finders often had certain noise.

One of the models that were converted from point cloud representation is the head shown on Figure 9. The corresponding point cloud was provided by Max-Planck-Institute for Computer Science, where it had been acquired by a 3D range finder. The number of produced nodes is actually less than the number of original points in the cloud. The surface appears to be bumpy, which probably reflects the details of the original bust; however, spatial patch representation proved to represent it sufficiently.

# of original points	915 498
# of patches	837

# of original points	915 498
# of patches	837
# of nodes	361 244

Figure 9: Head model

5. PRACTICAL RESULTS

In order to evaluate the proposed 3D model representation, we implemented the rendering algorithms based on z-buffering approach in the scope of the spatial patch rendering unit simulator. Its primary objective was to collect as much statistics as possible; thus, it was not designed to make use of any particular CPU architecture. The simulator stores all statistics in plain HTML format, which can be visualized in bars and diagrams by our custom-made viewer.

Some of the scene characteristics as well as rendering statistics are shown in Table 3. All models were rendered in a 500×500 pixel viewport by the rendering unit simulator. The two bottom rows show how many normal vectors were actually computed (as a percentage of the total number), if a buffering technique and pre-ordering of patches within a scene were applied. Thus, patch ordering results in dramatic increase in performance. We have to note that if all patches have identical configuration, i.e. they are produced by calibrated scanning device, then the table of normals can be pre-computed, and used for all patches, which reduces the number of normal computations to zero.

Table 3: Model characteristics and rendering statistics

	bunny	**man.**	**earth**	**head**
Patch nodes	441 014	330 905	325 992	361 244
Interpolated nodes	1 076 888	863 600	541 462	263 589
Norm. w/buffer	76%	76%	55%	46.6 %
Norm. w/b, ordered	2.8 %	3.7 %	1.9 %	18.2%

Figures in the *interpolated nodes* row of Table 3 shows that the models were actually magnified when rendered in the 500×500 pixel viewport. For this reason, the global test, described in Section 3, determined no patch as being suitable for optimized rendering in the given configuration. When the viewport was narrowed to 150×150 pixels, then about 85-95% of all patches did not require interpolation, and 45-85% of them passed the criterium and were rendered by the optimized procedure. Thus, on average, the criterium eliminated nearly half of the occurrences while it is computationally cheap. However, more accurate, but more expensive, tests may be used, as well.

The rendering unit is also capable of rendering triangles by conventional scan line algorithms that is used in most of the graphics boards and APIs. Thus we could render each patch by conventional means, dividing each quadrilateral cell into two triangles. A statistical analysis showed that for the presented models and viewing configurations the average number of operation required in triangular mode is double the number of operations required for bilinear interpolation. That proves the fact that rendering of many small triangles leads to high computational requirements, since in our case the triangles were actually several pixels in size.

6. CONCLUSION AND FUTURE WORK

In this paper we discussed some aspects of using spatial patches for 3D model representation and rendering. This primitive is based on the well-known concept of range image; however, each patch is expected to represent a small, relatively smooth fragment of a model surface for better performance. The proposed strategy unites the techniques such as texture and displacement mapping in one concept, for which it is thought of as highly efficient for model representation.

We presented the logical structure of a rendering unit for spatial patches. Based on the conventional *z*-buffer, it is designed to gain maximal benefit from spatial patch regularity and uniformity. It can be efficiently implemented using currently available hardware solutions, such as SIMD instructions and on-chip fast cache. However, even more increases in performance are expected if more features amenable to parallel computing are available.

Since neighboring spatial patches of a model are not spatially connected, noticeable artifacts may appear in the areas where several patches overlap if significant magnification occurs. This effect is mostly caused by low (8-bit) displacement precision that we used in our models and perspective incorrectness of interpolation. Such magnified views require more dense patches for accurate visualization.

Therefore, spatial patches are best suited for visualization with nearly the same scale as they were captured. Minification can also be supported by various filtering techniques similar to mip-mapping. The particular implementation is beyond the scope of this paper and is regarded as future work.

The other problems that are not discussed in this paper are anti-aliasing and backside culling. Aliasing effects can be dealt with by common supersampling strategies. The only solution for culling seems to be storing some kind of a bounding cone for patch surface and disregard patches that have their cones point away from the eye. However, there might be better solutions.

The direct rendering strategy with the *z*-buffer is not the only one that can be applied efficiently to spatial patches. In [6] we describe our experience in implementing ray-tracing approaches for the proposed representation. The results proved that spatial patches have many advantages, including regular structure and small size, which are quite important for efficient rendering by this type of techniques.

We have not yet paid attention to data compression for scenes represented by spatial patches. As the corresponding data stream may exhibit certain redundancies caused by displacement and texture similarity, various compression techniques might be applicable efficiently. Some progressive approaches might be used, as well.

In conclusion, we consider spatial patch as a primitive that is natural for 3D scanning, since no post-processing is basically required. The scenes represented by spatial patches often require less storage space than polygonal models with textures of the same quality. Rendering approaches, direct (*z*-buffer) and realistic (ray tracing), can be implemented efficiently benefiting from existing and future hardware solutions. Therefore, the introduced concept can be used in many applications dealing with real-life 3D graphics.

7. ACKNOWLEDGEMENTS

This work was carried on by Computer Graphics Group at the Mathematics Department of MSU under research agreement with Intel Technologies, Inc. We thank Jim Hurley and Alex Reshetov (Intel Technologies, Inc.) for their constant interest in this work. Authors also thank Alex Yakovlev, Victor Lempitsky and Genya Belkov for development of rendering unit simulator and model converters. We are grateful to Hugues Hoppe (Microsoft Research) for allowing us to use the mannequin data set, and to Leif Kobbelt (Max-Planck-Institute for Computer Science) for providing us with the head point cloud. The bunny model is the courtesy of Stanford University.

8. REFERENCES

1. J. F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics* 12(3), pp. 286-292, 1978.
2. E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. thesis, University of Utah, Salt Lake City, December 1974.
3. M. Deering. Data Complexity for Virtual Reality: Where do all the Triangles Go? *IEEE Virtual Reality Annual International Symposium (VRAIS)*, pp. 357-363. Seattle, September 1993.
4. S. Gumhold, and T. Hüttner. Multiresolution Rendering With Displacement Mapping. *Proceedings of the 1999 Eurographics/SIGGRAPH workshop on Graphics hardware*, pp. 55-66, 1999.
5. D. Ivanov, and Ye. Kuzmin. Spatial Patches – A Primitive for 3D Model Representation. *Computer Graphics Forum (Proc. of Eurographics'01)*, 2001.
6. V. Lempitsky, D. Ivanov, and Ye. Kuzmin. Adaptive ray tracing on spatial patches. *In this proceedings*, 2001.
7. Mark Levoy et al. The Digital Michelangelo Project: 3D Scanning of Large Statues. *ACM Computer Graphics (Proc. of SIGGRAPH'2000)*, pp. 131-144, 2000.
8. F. Neyret. Modeling, Animating, and Rendering Complex Scenes Using Volumetric Textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), pp. 55-70, January 1998.
9. J. W. Patterson, S. G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10(2), pp. 129-139, June 1991.
10. M. Woo et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Eddison-Wesley Pub, 3rd ed., 1999.

About the authors

Dr. Denis V. Ivanov, Scientist – Denis@fit.com.ru
Dr. Yevgeniy P. Kuzmin, Senior Scientist – Yevgeniy@fit.com.ru

Computational Methods Lab.
Mathematics and Mechanics Dept.
Moscow State University,
Vorobyovy Gory, Moscow, Russia, 119899