REAL-TIME ADAPTIVE POINT SPLATTING FOR NOISY POINT CLOUDS

Rosen Diankov*, Ruzena Bajcsy+
Dept. of Electrical Engineering and Computer Science
University of California, Berkeley
rdiankov@cmu.edu, bajcsy@eecs.berkeley.edu+*

Keywords: hardware rendering point-clouds

Abstract: Regular point splatting methods have a lot of problems when used on noisy data from stereo algorithms. Just a few unfiltered outliers, depth discontinuities, and holes can destroy the whole rendered image. We present a new multi-pass splatting method on GPU hardware called Adaptive Point Splatting (APS) to render noisy point clouds. By taking advantage of image processing algorithms on the GPU, APS dynamically fills holes and reduces depth discontinuities without loss of image sharpness. Since APS does not require any preprocessing on the CPU and does all its work on the GPU, it works in real-time with linear complexity in the number of points in the scene. We show experimental results on Teleimmersion stereo data produced by approximately forty cameras.

1 Introduction

Vision based applications that extract and display 3D information from the real world are becoming pervasive due to the large number of stereo vision packages available. The 3D information can be used to store and view virtual reality performances so that viewers are free to choose the viewpoint of the scene. A much more exciting topic called *teleimmersion* is to use the 3D information as a form of real-time communication between people at remote locations. In both these cases, users expect image quality that is indistinguishable from reality. However, teleimmersion requires filtering and displaying the 3D information at real-time speeds whereas in viewing performances offline, the 3D information can be preprocessed before hand, so real-time filtering is not an issue. We concentrate on the teleimmersion case where the 3D information is in the form of point clouds computed by stereo vision. Stereo algorithms produce very coarse point clouds with a lot of information missing due to occlusions and lighting conditions. Even if stereo algorithms could compute perfect depths with no outliers, the low resolution from the image will be apparent once the user gets close to the surfaces.

Research in the past has tackled point cloud rendering and filtering from several different directions. One popular method called point splatting represents each point as a small gaussian distribution. By summing up and thresholding these distributions in image space, the algorithm produces very smooth images (Elliptical Weighted Average Splatting (L. Ren and Zwicker, 2002) and (H. Pfister and Bross, 2000)). Other algorithms represent the neighborhood of each point by patches (Kalaiah and Varshney, 2001). (M. Levoy and Fulk, 2000) and (R. Pajarola, 2004) cover the case where the point clouds are composed of millions of points and the CPU and GPU have to be synchronized together. There has been a lot of work in converting point clouds to mesh data (R Kolluri and O'Brien, 2004) using triangulation based methods. Since there are usually people in teleimmersion scenes, researchers have fit skeletons to the point cloud data (Remondino, 2006) and (Lien and Bajcsy, 2006). The display component just uses the skeleton to deform a prior model of an avatar. But one reason for choosing to work with point clouds instead of meshes is that point cloud rendering is much more efficient than mesh rendering, and there has been work done in reducing meshes to point clouds(Stamminger and Drettakis, 2001). The filtering and reconstruction step in most of these algo-

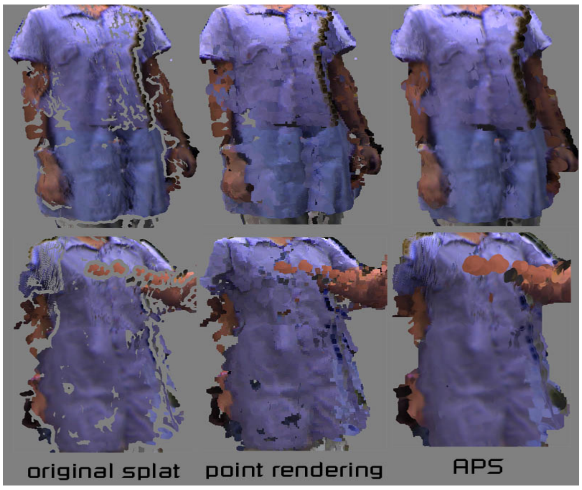

Figure 1: An example comparing the output of previously proposed splatting algorithms (left) with the current proposed APS method (right). Note that the original point cloud (middle) is pretty noisy and some information cannot be recovered at all.

rithms is inherently slow and is usually done offline. The real-time performance is usually attributed to the rendering component. Instead of fitting surfaces to this data in order to remove outliers and fill holes, we filter the point clouds directly on the GPU with no processing.

The Adaptive Point Splatting algorithm most closely resembles Elliptical Weighted Average Splatting (L. Ren and Zwicker, 2002) in that it does all filtering on the GPU in image space. In EWA, if a point cloud was computed by densely sampling from a 3D model without adding any noise to the point positions, the final point rendering would look just as sharp as the surface rendering. The intuition behind the algorithm is that each point represents a probability distribution of its color in 3D space. When the point is rendered, its color distribution is projected in the image plane. Then all the distributions of all points are combined and normalized to form the final image. If the combined probability is low for any pixel, that pixel is discarded. Different depths of the points are taken care of by first computing the depth of the nearest points to the camera for the whole image. For a given pixel, only points within a certain distance to the nearest depth contribute to the final pixel's color. It is in the final combination steps that EWA and other point splatting algorithms break down with noisy point clouds. The reasons will be explained in Section 2.

An ideal rendering algorithm needs to do three operations to the point clouds: remove outliers, fill holes, and smooth the colors on the surface while maintaining sharpness. Any type of CPU preprocessing like detecting outliers or filling holes will never run in real-time if there are more than 50,000 points. While hierarchical processing can speed up the asymptotic time, it will not work because it takes a significant amount of time to organize the data in the right structures before real work can be done on it. Organizing the data in volumetric structures, might be faster. However, a large amount of memory is needed to maintain even a 1000x1000x1000 cube; resolution is obviously poor. Even ignoring the time issue, it is still not clear how to fill holes with these data structures. Without strong prior information, it is impossible to fit surfaces to arbitrary 3D point clouds in real-time in order to figure out if a hole is present or not, but holes become pretty clear when looking at the rendered data in 2D. Since the three operations only require analysis of the local space around each point, they can be massively parallelized. Hence, the approach we take is to use modern GPUs to do all necessary processing in the image space (Figure 1).

2 Point Splatting Review

Let a point cloud consist of N points $\{X_i\}$ with colors $\{C_i\}$ and define a simple multivariate gaussian distribution around each point with covariance $\{\Sigma_i\}$. Then the color $C(X)$ at any point X in the 3D space is defined by

$$C(X) = \frac{\sum_i^N C_i P_i(X)}{\sum_i^N P_i(X)} \quad (1)$$

$$P_i(X) \propto \frac{1}{|\Sigma_i|^{0.5}} exp\{-\frac{1}{2}(X - X_i)^T \Sigma_i^{-1} (X - X_i)\} \quad (2)$$

Here, the probability that the point belongs to solid space is given by $P(X) = \sum_i^N P_i(X)$. Applying a simple threshold to $P(X)$ and tracing out the resulting surface by an algorithm like Marching Cubes will generate a geometric surface; however, this method is slow and cannot be done easily using the GPU only.

Fortunately, most of the work can be ported to the 2D projection space (L. Ren and Zwicker, 2002). Let Σ_i^* be the projection[1] of Σ_i, and $\bar{x}_i$ by the projection of X_i. Then for any point x in the image, the color can be computed in the same way as in Equation 1. The advantage is that an explicit surface doesn't need to be computed, the disadvantage is that the depth information gets lost and points that project to the same pixel but have different depths can interfere with each other. To partially solve this, EWA uses a 3-pass algorithm on the point clouds (Figure 2). To simulate

[1]The projection of a multivariate gaussian distribution is again gaussian.

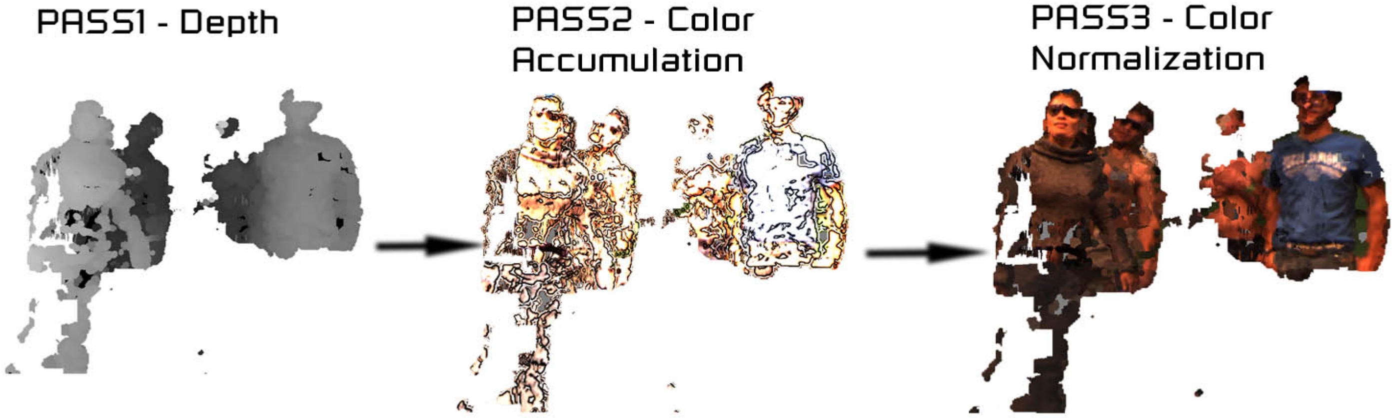

Figure 2: The original splatting method uses 3 passes: the first pass computes the depth buffer, the second pass renders the points that only pass this depth buffer, the third pass normalizes the accumulated colors.

Pass	Render Target	Z/Stencil Buffer	Z Test	Z Write
Visibility – fill depth buffer	None	D	off	on
accumulation – additive blend	A	D	on	off
normalization	C	None	off	off

Figure 3: GPU render states for the 3-pass splat algorithm.

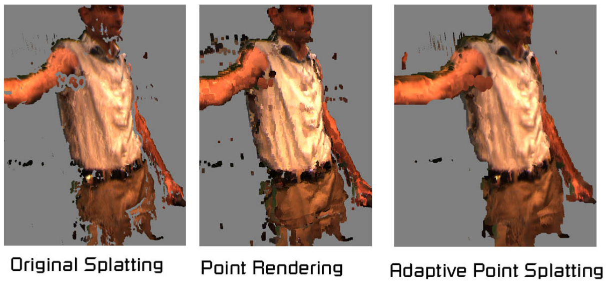

Figure 4: Artifacts occur with the original splatting method when there are depth discontinuities in the surface.

a probability surface, each point is represented by a simple quad whose texture coordinates are used to extract the probability from its projected gaussian distribution. The quads can be oriented to always face the camera, or can be given a static orientation if the surface normal is known apriori. There are various ways to compute a gaussian probability on the GPU: a simple texture can be used as a lookup table, or the probability can be computed directly in the GPU pixel shaders by applying Equation 1. We use the texture mapping method because it is faster and much easier to get working.

The first pass computes the depths of all the nearest points, so color information is not needed. The second pass only renders points that are close to the previously computed depth buffer. The probabilities and colors are combined with additive blending. The third pass normalizes each color with respect to the probability and performs the thresholding so that points with low probabilities are discarded. See Figure 3 for a description of the GPU state during these passes.

The problem with this simple approach is that points don't get filtered and depth discontinuities cause artifacts (Figure 4). The reason is beacuse the nearer pixels modify the depth buffer regardless of what the accumulated probability will be. During the second pass, only the front pixels are allowed to be rendered. The third pass sees the low accumulated probability and discards the pixel without considering pixels on the next nearest layer. This problem can be easily solved by employing an approach similar to depth peeling (Everitt, 2001). Depth peeling is an algorithm that can dynamically segment the view-dependent depth layers on the GPU. In the case of splatting, if the front layer yields low probabilities, then the next nearest layer will be considered for rendering. Holes still remain a problem and cannot be filled easily with the traditional point splatting method. To reduce holes, the gaussian variance can be increased or the cutoff threshold can be decreased so that fewer points are rejected; however, the image will get blurry and halos around surfaces will appear. Also, outliers will be rendered as big splats on the image. The outlier problem can be solved by decreasing the variance or increasing the cutoff threshold, but obviously that will introduce holes. Clearly there doesn't exist parameters in the simple 3-pass splatting algorithm that will solve one problem without introducing another.

3 Adaptive Point Splatting

The intuition behind APS is to perform the 3 EWA passes multiple times increasing the variance of the probability distributions each time. Pixels rendered in one iteration will not be changed anymore in the next iterations[2]. This scheme ensures that the color in pixels that have a high $P(x)$ doesn't get blended heavily with its neighbor colors, so image sharpness is preserved. Future iterations only deal with pixels with lower $P(x)$, so the variance can be increased without sacrificing image sharpness(Figure 5). This adaptive variance scheme allows the projection to cover a bigger area. Also, depth discontinuities won't be present because the aggregated cutoff threshold is very low.

The halos around the surfaces due to the low threshold are solved by first computing a mask in image space of what pixel should be discarded regardless of probability. The mask makes sure that edges are preserved, holes are filled, and outliers are filtered out. Therefore, the actual point filtering never modifies the real point cloud and is highly view-dependent. This makes APS more robust to noise because it can be hard and time consuming to determine an outlier just from its neighbors in 3D. The APS algorithm can be summarized as:

1. *Compute the black and white mask M by rendering the points to a low resolution texture.*
2. *Repeat 3-5 times. For each succeeding iteration.*
   (a) *Render the depth map for pixels where M is set.*
   (b) *Accumulate the probability and colors where M is set.*
   (c) *For each pixel that passes the probability threshold, normalize it and reset the corresponding pixel in M so that it gets ignored in succeeding iterations.*
   (d) *Increase variance for each point's distribution.*

4 GPU Implementation Details

The APS algorithm itself is very easy to understand in pseudocode, but the real challenge is to perform all computation steps in the GPU as fast as possible. It is very important to carefully manage the render targets for each pass so that the CPU never stalls having to wait for the GPU to finish rendering. Most graphics drivers these days buffer the GPU commands so that the GPU renders the scene while the CPU can perform other tasks like receive and decompress the data from the net. Transferring a render target from video to system memory is avoided at all costs. In order for the final point cloud to be rendered along with other background geometry, each point will have to have a depth value. Because final points are determined by the normalization phase, an image processing pass, it means that their depths will have to be manually set in the normalization pixel shader from a depth texture. The normalization phase will write the final points directly to the real back buffer instead of using a dummy render target that will later have to be rendered to the back buffer. Figure 7 details the whole process and necessary GPU states. One point worth noting is that any number of iterations and mask computation steps can be achieved with just 6 different render targets, which makes APS memory friendly.

[2]Each iteration consists of the 3 EWA passes.

Figure 6: A mask is used to make sure points are filtered and holes are filled. Note that the hole filling phase preserves the outer edges.

4.1 Computing the Mask

The masking effects in each iteration are implemented through the hardware stencil buffer. In order to fill the stencil buffer with the appropriate bits, a mask texture has to be produced. First the point cloud is rendered with traditional point rendering to a small render target[3]. Then the render target is used as a texture by an *erode* shader that gets rid of outliers. This result is then used by a *holefill* shader. For a pixel to be set, the *holefill* shader makes sure there are sufficient pixels on all four sides of the given pixel. Edges are preserved and holes between the arms are correctly ignored. *Holefill* is applied two to four times so that large holes can be filled. All these passes are achieved

[3]Masks don't need to have that much resolution, so we can get away with small render targets for this phase. We use 256x192 targets when the real back buffer dimensions are 1024x768.

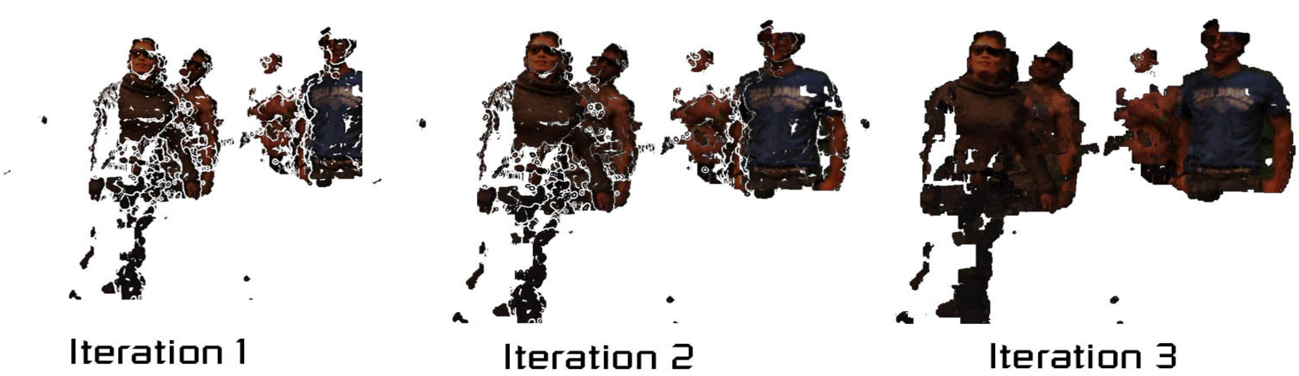

Figure 5: Adaptive Point Splatting performs multiple iterations over the point cloud. In each iteration, it relaxes the conditions for pixel rejection.

by double-buffering two textures: setting one as a render target while the other is set as a texture and vice versa (Figure 6).

Once the mask texture is computed, a stencil buffer of the original back buffer resolution needs to be filled to be used by the iteration phase. To fill the stencil buffer, set the targets, disable color writing, clear the stencil buffer, and render a quad that fills the whole surface. Set the stencil buffer operation to write the first bit for every successful pixel. The shader should read from the final mask texture and **discard**[4] pixels if the corresponding texture color is zero. This will make sure only pixels that are set in the original texture mask have their stencil bit set.

4.2 Executing an Iteration

During all 3 passes, the stencil buffer must be set to only accept pixels whose corresponding stencil bit is set. In the last normalization pass, the stencil buffer also needs to reset the stencil bit of the accepted pixels so they aren't modified in future iterations. The first two passes are generally the same as described in (L. Ren and Zwicker, 2002) except that the first pass also renders the depth of each pixel to a separate texture. This depth texture will then be used in the normalization phase to ensure that accepted pixels have the correct depth to be clipped properly by the background geometry rendered up to this time. In the second pass, we use the alpha channel to accumulate the probabilities and the RGB channels to accumulate the color. One important implementation detail when dealing with 16bit floating point render targets is the precision. There are only 10 bits of precision, so appropriate scaling in the probabilities is needed to make sure floating point errors don't occur.

[4]In Cg, there is a discard command that implements this behavior.

Pass	Render Target	Z/Stencil Buffer	Texture	Z Test	Z Write	Stencil Write	Stencil Test
Render point cloud	M_1	None	None	Off	Off	Off	Off
Erode	M_2	X	M_1	Off	Off	Off	Off
Dilate (2-4 times, flip targets)	M_1	X	M_2	Off	Off	Off	Off
Write to Stencil	None	D	M_1	Off	Off	Write **1**	always
Visibility	T_D	D	None	Off	On	Off	equal
Accumulation	A	D	None	On	Off	Off	equal
Normalization	C	None	A, T_D	Off	Off	Write **0**	equal

Repeat 3-5 times

D - depth buffer
T_D - depth render target
A - accumulation target
C - final color target
$M_{1,2}$ - temp mask targets

Figure 7: The render states for all the passes associated with the Adaptive Point Splatting algorithm. Any number of iterations can be performed with 6 separate render targets; therefore the memory complexity is constant with respect to the iterations.

Altogether $3 * num\_iterations + 6$ passes are needed for the whole algorithm.

5 Experimental Results

All our experimental results were done on teleimmersion data from UC Berkeley (of California Berkeley,) where 48 cameras were capture the scene in real-time. This data is then streamed to one computer acting as the renderer. Besides missing information and the typical rough estimates of the depths due to the real-time stereo algorithm, camera calibration errors persist. Sometimes three cameras could have face information, but merging them won't produce a coherent face because the three faces could be 1-2cm off. The colors are also offset a little due to the differing lighting from each different location. In each frame, the point clouds had roughly 75,000 points. On a 2Ghz machine with a GeForce 6600 card rendering at 1024x768 resolution and performing three hole filling passes and three iterations per frame, the algorithm runs at 10.5 frames per second. Since the points come from an image and a disparity map, we estimate the normal of the surface of each point by fitting a plane to the neighboring points. We orient the quad associated with each point to face the computed local normal vector. Although computing the normals has a small performance hit, we find that the resulting image quality is much better than with having the quads always face the viewer. Figure 8 shows more comparisons of Adaptive Point Splatting and traditional point cloud rendering.

6 Conclusion

We presented a GPU technique to deal with noisy point clouds from stereo data. Such point clouds will always be present since the stereo algorithms have to run in real-time. The main incentive to use the GPU for all processing is that the CPU is usually overloaded with many other tasks. Structuring the rendering algorithm such that no stalls occur between the GPU and CPU means that the CPU doesn't block on any rendering calls and both processor can truly run in parallel. Due to a GPU's parallel processing nature, GPU computing power seems to grow at a much greater rate than CPU computing power; therefore, GPU based algorithms will speed up at a greater rate if their performance is GPU bound, which Adaptive Point Splatting is. This type of filtering is impossible to do in real-time on today's CPUs.

REFERENCES

Everitt, C. (May 2001). Interactive order-independent transparency. In *http://developer.nvidia.com/attach/6545*.

H. Pfister, M. Zwicker, J. v. B. and Bross, M. (July 2000). Surfels: Surface elements as rendering primitives. In *SIGGRAPH Proceedings*.

Kalaiah, A. and Varshney, A. (June 2001). Differential point rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering*.

L. Ren, H. P. and Zwicker, M. (2002). Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Eurographics*.

Lien, J. and Bajcsy, R. (2006). Model driven compression of 3-d tele-immersion data. In *Technical Report, EECS Department, Universityi of California, Berkeley*.

M. Levoy, K. Pulli, B. C. S. R. D. K. L. P. M. G. S. A. J. D. J. G. J. S. and Fulk, D. (July 2000). The digital michelangelo project: 3d scanning of large statues. In *SIGGRAPH Proceedings*.

of California Berkeley, U. http://tele-immersion.citris-us.org.

R Kolluri, J. S. and O'Brien, J. (2004). Spectral surface reconstruction from noisy point clouds. In *Eurographics Symposium on Geometry Processing*.

R. Pajarola, M. Sainz, R. L. (June 2004). Extreme splatting: External memory multiresolution point visualization. In *UCI-ICS Technical Report No. 04-14*.

Remondino, F. (2006). Image-based modeling for object and human reconstruction. In *Ph.D Thesis ETH No 16562, Institute of Geodesy and Photogrammetry*.

Stamminger, M. and Drettakis, G. (June 2001). Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering*.

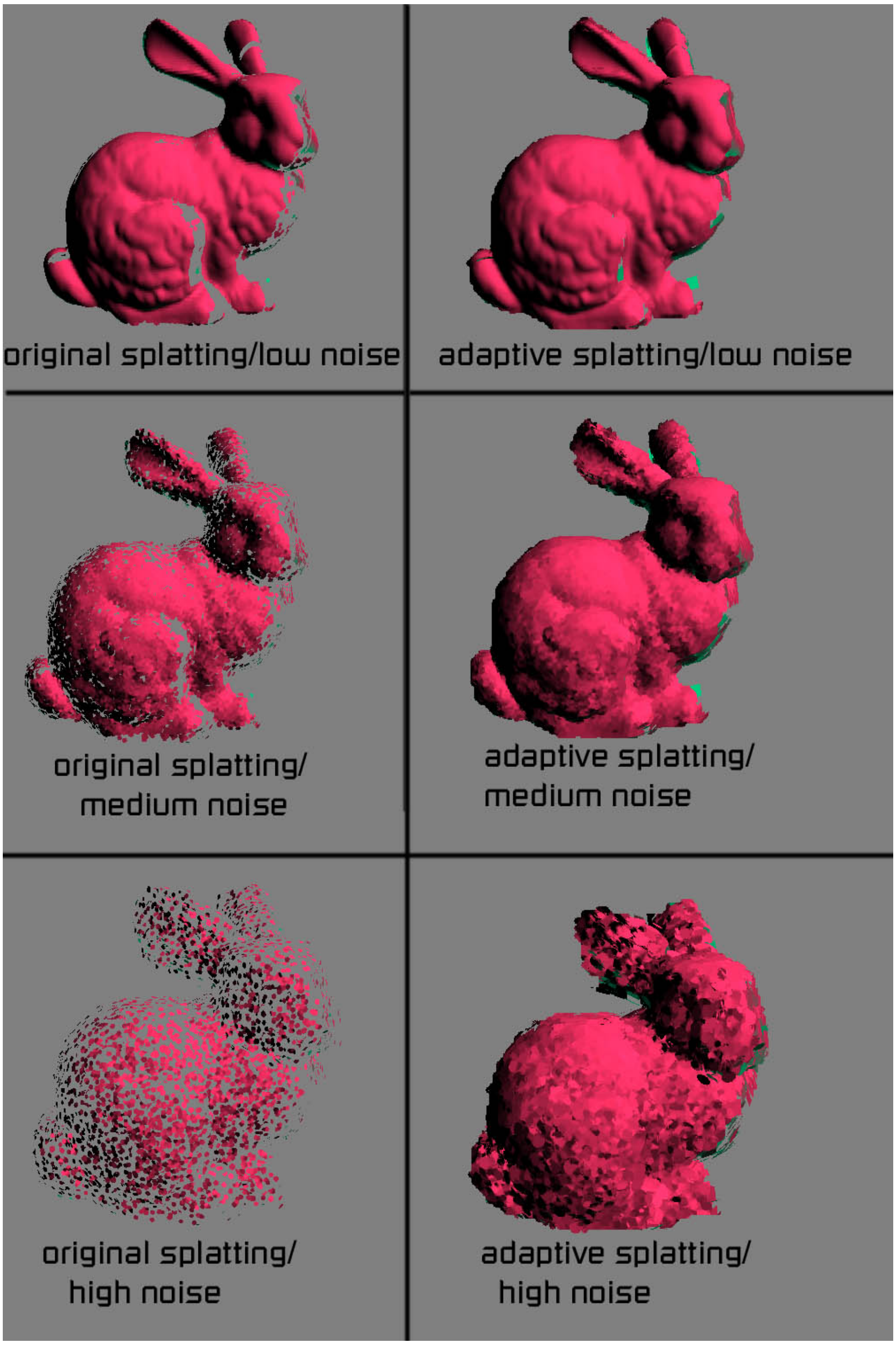

Figure 8: Example comparing the original splatting algorithm (left) with APS (right). Artificial noise was added to the point clouds on the second and third rows in order to simulate similar irregularities as in the stereo data. Even with a lot of noise, the APS rendering of the Stanford bunny looks solid and runs in real-time.