Reyes Rendering on the GPU

Martin Sattlecker*
Graz University of Technology

Markus Steinberger†
Graz University of Technology

Abstract

In this paper we investigate the possibility of real-time Reyes rendering with advanced effects such as displacement mapping and multidimensional rasterization on current graphics hardware. We describe a first GPU Reyes implementation executing within an autonomously executing persistent Megakernel. To support high quality rendering, we integrate displacement mapping into the renderer, which has only marginal impact on performance. To investigate rasterization for Reyes, we start with an approach similar to nearest neighbor filling, before presenting a precise sampling algorithm. This algorithm can be enhanced to support motion blur and depth of field using three dimensional sampling. To evaluate the performance quality trade-off of these effects, we compare three approaches: coherent sampling across time and on the lens, essentially overlaying images; randomized sampling along all dimensions; and repetitive randomization, in which the randomization pattern is repeated among subgroups of pixels. We evaluate all approaches, showing that high quality images can be generated with interactive to real-time refresh rates for advanced Reyes features.

CR Categories: I.3.3 [Computing Methodologies]: COMPUTER GRAPHICS—Picture/Image Generation I.3.7 [Computing Methodologies]: COMPUTER GRAPHICS—Three-Dimensional Graphics and Realism;

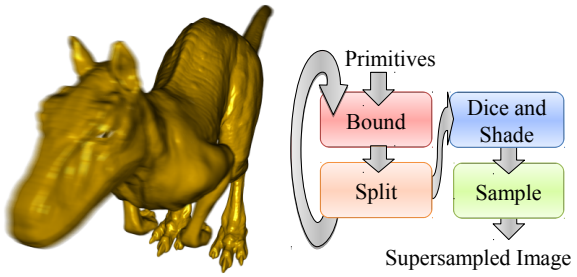
Keywords: Reyes, GPGPU, depth of field, motion blur, displacement mapping

The Reyes (Render Everything You Ever Saw) image rendering architecture was developed by Cook et al. in 1987 [Cook et al. 1987] as a method to render photo-realistic scenes with limited computing power and memory. Today it is widely used in offline renderers like e.g. Pixar's Renderman. Reyes renders parametric surfaces using adaptive subdivision. A model or mesh can, e.g., be given as a subdivision surface model or as a collection of Bezier spline patches. As a direct rasterization of such patches is not feasible, Reyes recursively subdivides these patches until they cover roughly a pixel or less. Then, these patches are split into a grid of approximating quads which can be rasterized easily. The Reyes rendering pipeline is divided into five stages. These stages are not simply executed one after another, but include a loop for subdivision, which makes Reyes a challenging problem with unpredictable memory and computing requirements. The pipeline stages are visualized in Figure 1(b) and listed in the following:

Bound culls the incoming patch against the viewing frustum and possibly performs back-face culling. If these tests do not discard the patch, it is forwarded to split or in case it is already small enough for dicing, directly to dice.

Split U/Vsplits the patch into two smaller patches. For Bezier patches, e.g., the DeCasteljau algorithm can be used. The resulting patches are then again processed by bound, to either undergo splitting again or go to dice.

*e-mail: m.sattlecker@student.tugraz.at
†e-mail: markus.steinberger@icg.tugraz.at



(a) Killeroo (b) Reyes Pipeline

Figure 1: (a) Image of the Killeroo model rendered with our real-time GPU Reyes implementation. (b) The Reyes pipeline is recursive, which makes it difficult problem for GPU execution.

Dice divides the patch into a grid of micropolygons. Each micropolygon is then processed by the Shade stage.

Shade computes the shading equations for each grid point of the diced micropolygons.

Sample rasterizes the micropolygon, interpolates the grid colors, and writes the result to the output buffer.

In the recent years the Graphics Processing Unit (GPU) has been developed from a fixed function pipeline chip to a massively parallel general purpose processor. With languages like NVIDIA CUDA it is now possible to run arbitrary algorithms on the GPU. Reyes was designed as a parallel algorithm from the beginning. However the recursive Bound and Split loop makes it difficult to implement it efficiently using the traditional approach of one kernel per pipeline stage. Thus, we use a queue-based Megakernel approach to solve this problem.

Based on this first implementation, we investigate displacement mapping in the context of GPU Reyes. Displacement mapping is a common technique to add details to a scene without adding additional geometry. This displacement can take place in the Dice stage. Then, we compare different micropolygon sampling algorithms, nearest neighbor sampling for each micropolygon and a proper rasterization method using the bounding box (BB) of each micropolygon.

As a final point, we add Depth of Field (DoF) and Motion Blur (MB), which are two important techniques to add realism to rendered scenes. Due to an exposure time > 0 physical cameras depict fast moving objects blurry. Virtual cameras however always have an exposure time of 0. To simulate motion blur, samples from different discrete times are considered. Similarly, due to an aperture > 0 physical cameras depict objects out of focus blurry. Virtual cameras have an aperture of 0. To simulate DoF, samples on a virtual camera lens are considered.

The remainder of this paper is structured as follows: The following section discusses related work. In section 2 we present our implementation of Reyes. In section 3 we describe the algorithms

used for displacement mapping, micropolygon rasterization, depth of field and motion blur. In section 4 we present results and compare the different depth of field and motion blur algorithms, followed by the conclusion.

1 Related Work

Reyes In 1987 Cook et al. [Cook et al. 1987] introduced the Reyes image rendering architecture, designed for photo-realistic results. It is based on the idea of adaptive surface subdivision. The algorithm was designed to be run in parallel. In combination with the introduction of general purpose computation on the graphics card this gave rise to several GPU based Reyes implementations.

In 2008 Patney and Owens [Patney and Owens 2008] were the first to move the complete bound/split loop to the GPU. Their implementation uses a breadth-first approach. Three kernel launches are needed for each level of the split tree. These introduce overhead due to CPU - GPU communication. The dicing and shading stage were also computed on the GPU. Micropolygon rendering was done by OpenGL.

The first algorithm that implemented the complete Reyes pipeline within a general purpose GPU environment was RenderAnts [Zhou et al. 2009]. They introduced additional scheduler stages for dicing, shading and sampling. RenderAnts supports Renderman scenes and is capable of executing Renderman shaders. Advanced effects such as depth of field, motion blur and shadow mapping are also supported.

In 2009 Aila et. al. [Aila and Laine 2009] showed that a persistent kernel using a queue can perform significantly faster than the traditional approach of one thread per work item. In 2010 Tzeng et al. [Tzeng et al. 2010] used this approach to implement the first GPU Reyes pipeline that uses a persistent Megakernel for the bound and split loop. The rest of the pipeline is split into four different kernels. Their implementation uses distributed queuing with task donation for work distribution. They were the first to achieve interactive frame rates on a single GPU.

Nießner et al. [Nießner et al. 2012] presented a method for fast subdivision surface rendering for the DirectX pipeline. They use compute shaders and the hardware tessellation unit for subdivision. While they achieve high performance, their approach is still an approximation to the limit surface using the hardware tessellation unit. Nießner et al. [Nießner and Loop 2013] extended this approach with displacement mapping. They use a tile based texture format with an overlap to eliminate cracks, and mip-mapping to eliminate undersampling artifacts.

Steinberger et al. [Steinberger et al. 2014] introduced Whippletree, an approach to schedule dynamic, irregular workloads on the GPU. This approach is well suited for the irregular bound and split loop of the Reyes pipeline. They also present an implementation of Reyes, which is the basis of the application in this paper.

Motion Blur and Depth of Field Real-time DoF and MB algorithms often use a screen-space approximation. While producing overall good results, they can produce visible artifacts along sharp depth discontinuities [Fernando 2004].

The OpenGL accumulation buffer can be used to simulate DoF and MB [Haeberli and Akeley 1990]. For this method multiple rendering passes using the same scene and different points in time and positions on the camera lens are needed. They are then added to each other using the accumulation buffer. This approach has the overhead of multiple complete rendering passes of the entire scene.

Fatahalian et al. [Fatahalian et al. 2009] showed that for micropolygons rasterization an algorithm using all the pixels in the bounding box is superior to common polygon rasterizers. Fatahalian et al. [Fatahalian et al. 2009] also introduced a DoF and MB algorithm designed for micropolygon rendering. The algorithm divides the screen into small tiles and assigns different samples to each pixel inside a tile. It trades memory consumption and ghost images with noise in the output image. This algorithm eliminates the need for multiple render passes introduced by the accumulation buffer method. Therefore, the overhead of multiple perspective transformations and shading computations for each polygon is eliminated. In 2010 Brunhaver et al. [Brunhaver et al. 2010] presented a hardware implementation of this algorithm. They show, that an efficient implementation of motion blur and defocus is possible.

Munkberg et al. [Munkberg et al. 2011] introduced a hierarchical stochastic motion blur algorithm. The algorithm traverses the bounding box of a moving triangle in a hierarchical order. The screen is divided into tiles and before sampling a tile, the temporal bounds are computed to minimize the number of sample tests. They use different sample positions and sample time for each pixel. These sample positions are computed on the fly.

2 Simple GPU Reyes

We implemented a first GPU Reyes pipeline using CUDA and Whippletree [Steinberger et al. 2014], similar to the implementation described by Steinberger et al. [Steinberger et al. 2014]. Whippletree uses a persistent Megakernel approach. The stages of the algorithms are implemented as C++ classes and called procedures. A procedure can spawn the execution of another procedure by inserting it into a queue. When a procedure finishes execution, a new work item is pulled from a queue.

For the following description we assume cubic Bezier patches as input geometry. Note that we also support subdivision surfaces. However, the subdivision algorithms are a bit more complicated. Thus, we focus on the easier to explain Bezier patch algorithm.

The stages of the Reyes pipeline are implemented as procedures. One procedure computes the first bound stage of the pipeline. The patches are then passed to a combined Split and Bound procedure. When ready for dicing, the patches are passed to a combined dice and shade procedure. The output is written to GPU memory, which is then displayed and if necessary downsampled via OpenGL. The procedures are described in the following paragraphs.

Bound is executed for every input patch of the model. This procedure clips a patch, if it is completely outside the view frustum. It also decides, if a patch needs to be split. If the size of the screen space BB is below a certain threshold in x and y direction, the patch is forwarded to dicing. If not, the patch is forwarded to the SplitU or SplitV procedure. This procedure uses 16 threads per input patch. Each thread is responsible for one control point.

Split U/V and Bound are actually two separate procedures. One for the split in U direction and one for the split in V direction to reduce thread divergence. Here, the patch is split in halves using the DeCasteljau algorithm. Then, the same checks as in the Bound procedure are executed for each of the two new patches. They are then either clipped, forwarded to dicing, or forwarded to Split U/V. This procedure uses 4 threads per input patch. Each thread is responsible for one row/column of control points.

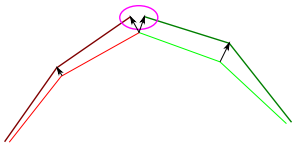


Figure 2: Using the approximative micropolygon normals for displacement mapping can result in holes between micropolygons. In this 2D example, red lines depict one micropolygons and the green lines the other. The approximations lead to different normals at the corner points and thus to holes after displacement mapping.

Dice and Shade first divides the input patch into 15×15 micropolygons, then the resulting 16×16 points are shaded. The micropolygons are then rasterized and the interpolated color is written to the output. This procedure uses $16 \times 16 = 256$ threads per patch. The rasterization uses one thread per micropolygon. The 16×16 corner points are computed with the DeCasteljau algorithm. First, 16 cubic Bezier curves are computed from the 4×4 control points, using one thread per curve control point. Then these 16 curves are subdivided into 16 points along each curve using one thread per corner point.

3 Advanced Effects for GPU Reyes

Based on the simple GPU Reyes implementation, we present different approaches to realize advanced rendering effects. We start with displacement mapping, before dealing with more advanced effects like real-time MB and DoF.

3.1 Displacement Mapping

In order to capture small details using Bezier patches only, a high number of patches would be needed. This has a large impact on performance and memory requirements, and it essentially renders surface subdivision useless due to the initially high patch count. An alternative to provide details in surface modeling is displacement mapping. A texture is projected onto the model. Each texel contains information on how much the surface should to be displaced. The displacement happens along the normal of the surface. Positive and negative displacements are possible. In Reyes, displacement mapping is applied after the dicing step: Each point of the grid is displaced before the micropolygons are shaded and rasterized. In order to apply displacement mapping, texture coordinates throughout the entire surface are needed. Every corner point of a patch needs a texture coordinate. When a patch is split, texture coordinates need to be assigned to the two new patches.

3.1.1 Normal Computation

Precise normal computation is important for displacement mapping, especially along common edges of patches after splitting. When normals are approximated, using neighboring points on the micropolygon grid, displacement mapping leads to holes as seen in Figure 2. The normal must be computed directly from the parametric surface instead. The normal of a surface at a specific point is equal to the normal of the tangent plane at the same point. Furthermore, the normal of the tangent plane can be computed by the cross product of two vectors on the tangent plane. We use the tangent in v direction and the tangent in u direction. To compute the tangent in u direction at a specific point (u_t, v_t), we first have to compute a cubic Bezier curve that contains this point and runs in u direction.The same is true for the v direction. A cubic Bezier patch is defined by 16 control points, which can be seen as 4 cubic Bezier curves in u and in v direction. A Bezier curve that runs through the point (u_t, v_t) in u direction is computed as follows. The DeCasteljau algorithm is used on all four curves in v direction to get the curve at v_t. The four new points define the needed curve in u direction. The tangent of the curve at coordinate u_t can then be computed using the DeCasteljau algorithm of a quadratic bezier curve [Shene 2011] :

$$tangent(u_t) = deCast(P2 - P1, P3 - P2, P4 - P3, u_t) \quad (1)$$

$P1$ to $P4$ are the control points of the curve at $v = v_t$. The same algorithm is used for the v tangent. From the two tangents, the normal is computed using the cross product.

The normal is computed for every grid point in the micropolygon grid. All needed Bezier curves in u direction are already computed in the dicing process. The curves in v direction are computed the same way and are also stored in shared memory. For this stage $4 \times 16 = 64$ threads are used. After this step one thread per grid point computes the tangents and subsequently the normal for the corresponding (u, v) coordinates using equation 1. After the displacement, the normals are approximated using neighboring points in the displaced micropolygon grid.

3.2 Micropolygon Rasterization

The dicing stage produces micropolygons that are approximately one subpixel in size. Therefore, a simple nearest neighbor filling produces acceptable results. For this approach only the pixel center, which is nearest to the middle point of a micropolygon is considered. If this pixel center lies inside the micropolygon and the Z-test passes, the shaded color of the micropolygon is written to the output. This is also one of the algorithms we implemented. The results were acceptable for simple patch rendering. But even there a reduction of the micropolygon size is needed to ensure that all pixels are shaded. When displacement mapping is enabled, the simple algorithm fails to produce acceptable results.To consider all pixels that are covered by a given micropolygon, we implemented a proper rasterization algorithm:

```
Compute the BB of the micropolygon
For each pixel center inside the BB
 If pixel center is inside the micropolygon
  If Z-test passes
   Write color and depth to output
```

This algorithm checks all pixel centers inside the bounding box (BB) of the micropolygon. The BB was chosen because it covers all pixels, and the ratio of samples inside the polygon versus samples outside the polygon is smaller than with stamp based methods, as shown by Fatahalian et al [Fatahalian et al. 2009]. The inside test divides the micropolygon into two triangles, and checks if the sample is inside one of them.

The rasterization is performed for all micropolygons of a diced grid in parallel. Parallel execution of the rasterization loop is not advisable, since the size of a typical micropolygon bounding box is only one pixel.

3.3 Motion Blur and Depth of Field

Motion Blur (MB) and Depth of Field (DoF) are two methods that add realism to a scene. First, we describe each method, then we investigate three algorithms to produce Motion Blur and DoF.

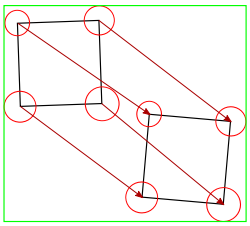


Figure 3: Extended bounding box of a square micropolygon. The red circles depict the circle of confusion. The red arrows depict the motion vectors of the corner points. The size of the bounding box increases dramatically, even for small motion vectors.

Motion Blur To simulate motion blur, samples from different discrete times are computed. This implementation uses a linear interpolation along a two dimensional motion vector. Each grid point is displaced independently. The vector is computed by transforming the points with a model-view-projection matrix from some time in the past. Samples are taken along this motion vectors.

Depth of Field To simulate depth of field, samples on a virtual camera lens are considered. This results in an circle of confusion (CoC) around a computed point. The diameter of the CoC depends on the distance from the camera, the aperture and the plane in focus. The diameter is computed for each point in the diced micropolygon grid. Each point is displaced by a sample vector multiplied by the radius of the CoC. A sample vector corresponds to a position on the lens. Sample vectors are chosen inside a circle with a diameter of 1 pixel.

We investigate three algorithms which produce these effects. In each algorithm the samples are stored as subpixels in the output image. These subpixels are later averaged. Each of the algorithms supports both motion blur and depth of field. If only one of the methods is activated, the circle of confusion or the motion vector is set to zero. The combination of the motion vector and the x and y position on the lens forms a 3D sample. The 3D sample positions are randomly selected, when one of the algorithms is activated.

Simple UVT Sampling is the simplest and fastest of the three algorithms. It however produces ghost images of the object if too few samples are chosen. The algorithm is similar to the accumulation buffer method [Haeberli and Akeley 1990]. Instead of an accumulation over multiple passes, the rasterization of a micropolygon is repeated multiple times. Each time a different 3D sample is used. The corner points of the polygon are moved according to the motion vector and the position on the CoC. Then, rasterization is performed.

```
Compute CoC and Motion Vector
i = 0
For each UVT sample
 for each corner point
  Add motion vector and position on CoC
 Rasterize micropolygon (Use subpixel i)
 i++
```

Interleave UVT (IL) is a more complicated and slower version of the simple sampling algorithm. It produces no ghost image, but noise instead. More samples are considered, but not every sample is present in every pixel.

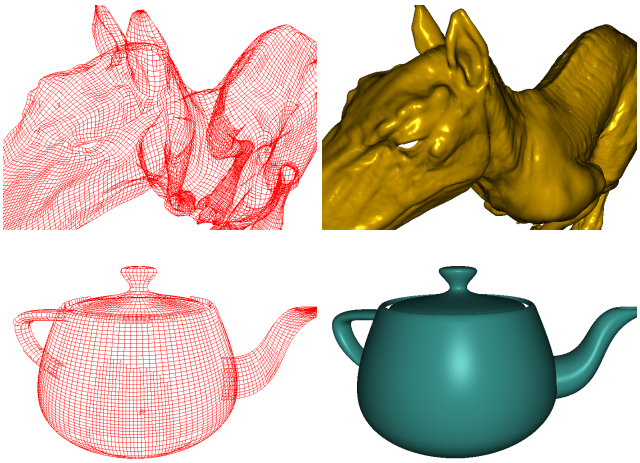


Figure 4: Rendering of the Killeroo and the Teapot model. The left image shows the patches before dicing and the right one shows the final rendering. The Killeroo model requires less subdivision, because it has a high initial patch count. In the teapot image a higher subdivision of patches that appear larger in the output image can be seen.

As in the simple method, the rasterization step is repeated for every 3D sample. The number of samples, and the rasterization itself is modified by the introduction of tiles. The screen is divided into tiles of size K squared, $K > 1$. Each tile has N unique UVT samples. Resulting in $M = \frac{N}{K \times K}$ samples per pixel. Each sample is assigned to a specific subpixel of a pixel in a tile. The BB is computed over the tiles of the output image. In each rasterization step, only the currently active sample is considered for inside testing. The sample position is given by the position inside the tile [Fatahalian et al. 2009].

```
Compute CoC and Motion Vector
For each UVT sample
 for each corner point
  Add motion vector and position on CoC
 Compute tile BB
 For each tile in BB
  If sample position is inside micropolygon
   If Z-test passes
    Write color and depth to output
```

Interleave UVT introduces a pattern noise with a period equal to the tile size. This pattern can be reduced by the permutation of the samples across the tiles in the output image. For this approach, P sample permutations are precomputed. The permutation for each tile is chosen as follows:

$$tilenumber = tile_y \cdot \frac{scenewidth}{gridsize} + tile_x, \tag{2}$$

$$perm = tilenumber + tile_x + tile_y \bmod P. \tag{3}$$

This formula was chosen over the *tilenumber* because common rendering resolutions have multiples of used tile sizes as the number of pixels per row. This would lead to the same permutation in every screen column and therefore a visible pattern.

The performance is similar to the simple algorithm using the same number of samples (e.g. 2×2 tiles $\cdot\, 16 = 64$ samples). This is because the bounding box of a typical micropolygon that only covers one pixel also covers one tile. Thus, the size of the bounding box and therefore the number of samples considered is approximately the same. The memory consumption however is much smaller.

There is however a small overhead for the computation of the tiles and the sample permutation.

Bounding Box Algorithm assigns different UVT samples to different output pixels. Unlike the other two, this algorithm does not rasterize the micropolygon for each 3D sample. It expands the BB of the micropolygon so that all possible positions in time and on the lens are considered in one rasterization step. Thus, it also assigns different 3D samples to different output pixels. It is necessary that every pixel has the same 3D samples for every rasterized patch. Otherwise, the Z-test for a specific subpixel might compare two patches from a different time or a different lens position. In order to assign different 3D samples to different pixels, we use a texture that contains the samples for every pixels. To reduce memory consumption, we repeat a 64×64 texture over the entire output image.

The algorithm performs a lookup for every pixel that might be covered by the micropolygon. The BB is extended by the maximal motion vector and the CoC around every corner point as seen in Figure 3. For every pixel in this bounding box, every UVT sample is considered. The micropolygon is moved according to the T sample. The sample point is moved along the negative UV sample direction. Then it is checked, if the sample position lies inside the moved micropolygon.

```
Compute extended BB
for each pixel P in the BB
 for each sample per pixel
  Lookup UVT sample for pixel and sample
  Move P in negative UV direction
  for each corner point
   Add motion vector
  If P is inside micropolygon
   If Z-test passes
    Write color and depth to output
```

The performance of this algorithm is heavily influenced by the size of the extended BB. Fast moving objects and out of focus objects will reduce performance.

4 Results

In this section we discuss the results achieved by the described algorithms. All tests were run using a machine with a AMD Athlon X2 270 CPU and a GeForce GTX680 graphics card. The program was compiled on Windows 7 using CUDA 6.0. To evaluate the tested algorithms in isolation, we test simple objects with a single colored background only. As test scenes we use the Utah Teapot (32 patches) and the Killeroo (11532 patches). In Figure 4 example renderings of the Killeroo and the Teapot model are shown.

4.1 Displacement mapping

In Figure 5 the effect of displacement mapping on the Killeroo model is shown. The performance cost of the displacement mapping is between 5% and 8%. For this small difference a lot of small details are added to the scene. Without displacement mapping additional geometry would be needed to show the same small details which would impact performance more significantly. Displacement mapping needs precise normals. Otherwise, holes appear as can be seen in Figure 6. The precise normal computation adds another 5-8% to the rendering time. This results in a overall performance loss of about 10-15% for displacement mapping.

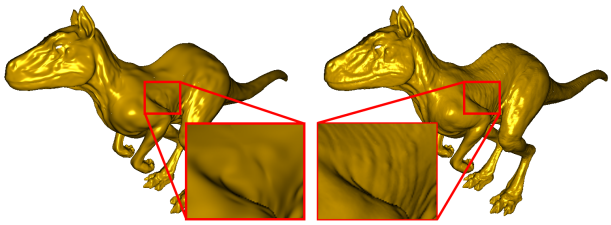
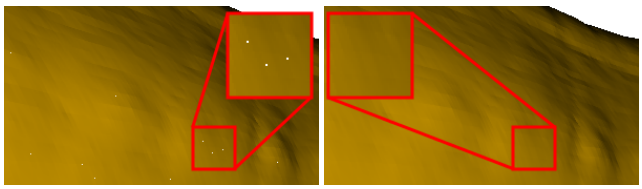


Figure 5: Displacement mapping (right) adds detail.



(a) Approximated normals (b) Computed normals

Figure 6: Comparison of normal computation for displacement mapping. Approximate normals produce holes.

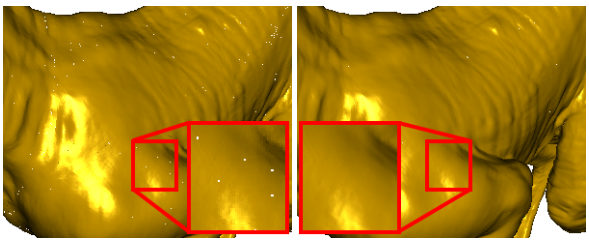


Figure 7: Comparison of the micropolygon rasterization methods. The naive method (left) produces holes. Rasterization of the micropolygons (right) eliminates them.

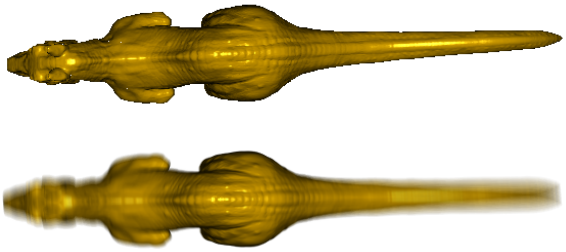
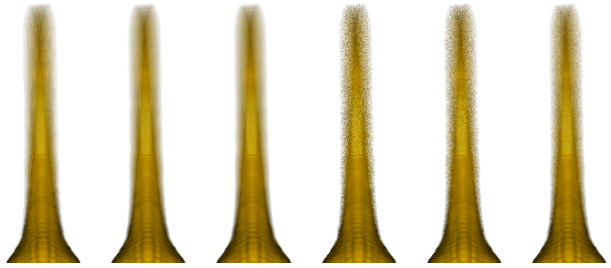
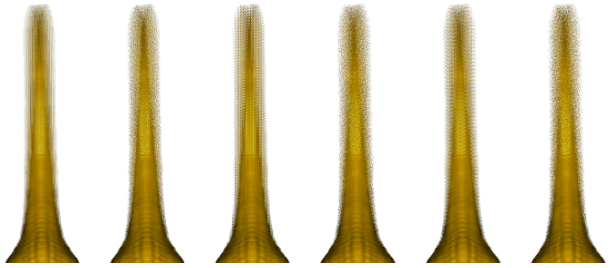


Figure 8: The upper image shows the full rendering, that was used in the MB comparison without motion blur. The lower image shows the same scene with MB enabled. The simple algorithm with 225 samples per pixels was used.



(a) S. 16 (b) S. 64 (c) S. 225 (d) BB 16 (e) BB 25 (f) BB 64

Figure 9: Comparison of the simple motion blur algorithm (S.) and the BB algorithm for motion blur. The numbers denote the number of samples for each pixel.



(a) 2x2 (b) 2x2 *p* (c) 3x3 (d) 3x3 *p* (e) 4x4 (f) 4x4 *p*

Figure 10: Comparison of the IL algorithm with different parameters. The numbers denote the tile size. *p* indicates, that 64 permuations of the samples were used. The number of samples is 16 for all examples.

4.2 Micropolygon rasterization

The naive implementation uses one sample per micropolygon. This approach produces holes. Especially, when displacement mapping is enabled as seen in Figure 7(a). This happens, because the displacement of the grid points in the dicing stage produces deformed micropolygons that cover more than one pixel. As it can be seen in Figure 7, rasterization eliminates the holes that occur through displacement mapping. The remaining holes are caused by small errors introduced in the Killeroo model during conversion.

The performance impact of the rasterization algorithm depends on the amount of rasterization done compared to the rest of the algorithm. The Killeroo model has a high initial number of patches. This means that little patch splitting is needed and that the patch count is high even when the screen size of the model is small. The performance costs for proper rasterization for this model are between 8.5% for small renderings and 16% when the model fills the whole screen. The teapot model, on the other hand, has a low initial patch count. This means that the amount of rasterization is more dependent on the screen space size of the model. The rasterization performance costs for this model are between 2% for small renderings of the model and 20% when the model fills the whole screen.

4.3 Motion Blur

In this section we compare the different motion blur algorithms with different parameters and against each other. For this comparison the Killeroo model was used while it spins around the z-axis and is viewed from above. For the visual comparison, only the tail of the model is shown. The renderings and performance measurements were performed showing the whole model as seen in Figure 8.

Simple In Figure 9 three examples of the simple motion blur algorithm are shown. In the first image ghost images of the tail can be seen. They start to disappear with a higher number of samples, but are still visible at the end of the tail using 64 samples per pixel. This high number of samples decreases the performance of the application, e.g., by a factor of 3.5 between 16 and 64 samples.

Bounding Box In Figure 9 three examples of the BB algorithm are shown on the right side of the page. They show the algorithm with different numbers of samples per pixel. The examples show no ghost images. There is, however, a very noticeable noise.

Samples	Simple	BB	IL 2x2	IL 3x3	IL 4x4
16	0.046	0.125	0.14	0.28	0.46
25	0.072	0.19	0.22	0.43	
64	0.16	0.49			
225	0.52				

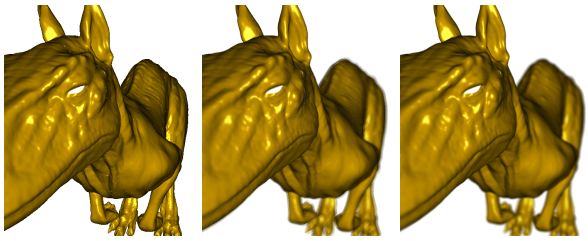
Table 1: Rendering time in *s* for the different MB algorithms using the Killeroo scene and a 800×600 viewport.

Interleave UVT In Figure 10 six examples of the Interleave UVT algorithm for motion blur are shown. Three different tile sizes were used. For each tile size an image with 1 and one with 64 permutations of the sample positions in the tiles was rendered. The algorithm without sample permutations shows a clear pattern with the same size as the tiles. The introduction of permutations increases the quality of the visual output significantly as seen in Figure 10.The difference in tile size is much more visible without permutations, but for a 2x2 tile size, even the permutated samples produce some pattern. This pattern disappears with larger tile sizes.

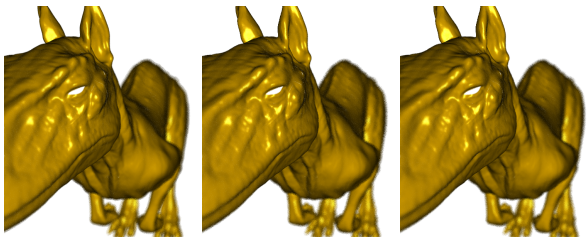
Comparison Visually, the IL and the BB algorithms produce similar results, when a large enough tile size and a permutated sampling pattern is used for the IL algorithm. They both produce random noise, opposed to the ghost images seen from the simple algorithm.

In Table 1 the performance of the different algorithms with different settings is shown. The render time for the same scene without any motion is 0.0067s. There is a factor of about 7 between disabled motion blur and the simple algorithm with only 16 samples. This means that the execution time is dominated by the rasterization stage when MB is enabled. The performance of the simple algorithm is similar to the performance of the IL algorithm with the same number of samples. For example, the performance of the simple algorithm with 64 samples is similar to the IL algorithm with a tile size of 2x2 and 16 samples. The computation time of the simple and the IL algorithm roughly scales with the number of samples.

For this example, the BB algorithm is much faster than the IL algorithm. The performance of the BB algorithm with 64 samples is approximately the same as the IL algorithm with 16 samples and a tile size of 4. However, this does not mean that BB is always faster than the IL algorithm. The performance is heavily influenced by the size of the extended bounding box. This means that for a larger motion blur BB will perform worse, whereas the runtime of the IL algorithm will stay the same.



(a) No DoF (b) Simple 16 (c) Simple 225



(d) IL 2x2 16 (e) BB 16 (f) BB 25

Figure 11: Comparison of the DoF algorithm outputs. The number denotes the number of samples per pixel, and the tile size for the IL algorithm.

#Samples	Simple	BB	IL 2x2	IL 3x3	IL 4x4
16	0.039	0.63	0.20	0.39	0.63
25	0.097	0.98	0.31	0.61	0.93
36	0.14	1.4	0.41		
225	0.79				

Table 2: Rendering time in *s* for the different DoF algorithms using the Killeroo scene and a 800×600 viewport.

4.4 Depth of Field

For DoF the same algorithms as for MB were used. The advantages and disadvantages of the algorithms are basically the same as with MB. Thus, we do not go into the details of the algorithms.Figure 11 shows the different algorithms, while Table 2 presents the performance measurements with different settings. The render time for the same scene without any motion is 0.014s.

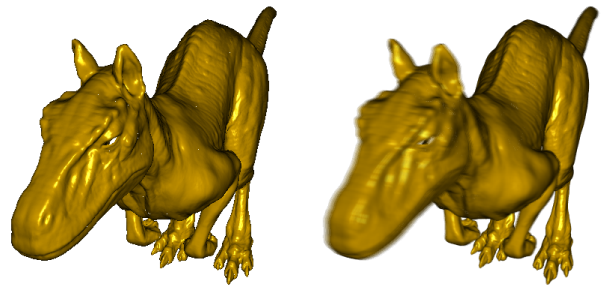
4.5 Combined Depth of Field + Motion Blur

Figure 12 shows renderings of the Killeroo model with combined DoF and MB. The advantages of the algorithms are the same as for the single effects. The simple algorithm shows ghost images around the blurred areas, whereas the other two algorithms produce noise in the same areas.

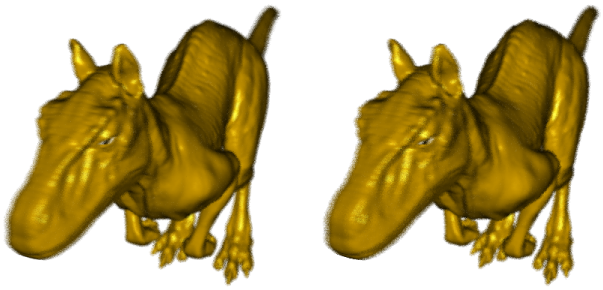
5 Conclusion

We have shown a realtime capable implementation of the Reyes algorithm on the GPU. The implementation uses a persistent Megakernel and queues. It can produce advanced effects, such as displacement mapping, motion blur and depth of field.

We have compared a naive nearest neighbor sampling algorithm



(a) None (b) Simple 16



(c) IL 2x2 16 (d) BB 16

Figure 12: Comparison of the combined DoF and MB algorithms. The number denotes the number of samples per pixel, and the tile size for the IL algorithm. The plane of focus is at the head of the model, and it rotates around its z-axis.

with a proper micropolygon rasterization approach. This algorithm considers all pixels inside the BB of a micropolygon. The impact on rendering performance for this algorithm is relatively low and it eliminates the holes created by the naive implementation. We investigated displacement mapping for Reyes. This is a method that adds small details to the scene without introducing new geometry. To get good results for displacement mapping, we have shown that a precise normal computation is crucial. Overall the performance impact of displacement mapping on Reyes is relatively small.

Furthermore, we have shown the differences between three MB and DoF algorithms. The simple algorithm needs high sample rates to counter the problem of ghost images. The IL and the BB algorithm eliminate this problem, but introduce noise.Using a large enough tile size IL and BB produce similar results with a random noise. The performance of the BB algorithm heavily depends on the length of the motion vectors and the size of the CoC. The performance of IL and the simple algorithm, is not influenced by these parameters. Therefore, the choice between the IL and BB algorithm depends on the amount of blur produced during rendering. MB and DoF need significant computational power, but for low sample numbers, the implementation achieves interactive framerates.

In the future we would like to expand our implementation of Reyes to support more complex scenes, e.g. through Renderman scenes and the Renderman shading language. We would like to show that an efficient rendering of photo realistic scenes within an autonomously executing persistent Megakernel is possible.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, 145–149.

BRUNHAVER, J., FATAHALIAN, K., AND HANRAHAN, P. 2010. Hardware implementation of micropolygon rasterization with motion and defocus blur. *Proceedings of the ....*

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *ACM SIGGRAPH*, 95–102.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *High Performance Graphics 2009*, 59–68.

FERNANDO, R. 2004. *GPU gems*. Addision-Wesley.

GORDON, J. Binary tree bin packing algorithm.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. *ACM SIGGRAPH 24*, 4, 309–318.

MUNKBERG, J., CLARBERG, P., AND HASSELGREN, J. 2011. Hierarchical stochastic motion blur rasterization. In *High Performance Graphics*, 107–118.

NIESSNER, M., AND LOOP, C. 2013. Analytic displacement mapping using hardware tessellation. *ACM Transactions on Graphics (TOG) 32*, 3, 26–26.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Transactions on Graphics (TOG) 31*, 1, 6–6.

PATNEY, A., AND OWENS, J. D. 2008. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph. 27*, 5, 143–143.

SHENE, C., 2011. Lecture notes for introduction to computing with geometry at michigan technological university.

STEINBERGER, M., KAINZ, B., KERBL, B., HAUSWIESNER, S., KENZEL, M., AND SCHMALSTIEG, D. 2012. Softshell: dynamic scheduling on gpus. *ACM Transactions on Graphics (TOG) 31*, 6-6, 161–161.

STEINBERGER, M., KENZEL, M., BOECHAT, P., KERBL, B., DOKTER, M., AND SCHMALSTIEG, D. 2014. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph. 33*, 6-6 (Nov.), 228:1–228:11.

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the gpu. In *High Performance Graphics*, 29–37.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. Renderants: Interactive reyes rendering on gpus. In *ACM SIGGRAPH Asia*, 155:1–155:11.