Hardware accelerated ray tracing for diffusion curves

Mika Zeilstra[1] , **Amal Parakatt**[1] , **Elmar Eisemann**[1]
[1]TU Delft

Abstract

This paper accelerates the rendering of diffusion curves using the ray tracing cores found on modern NVIDIA graphics cards using the method described by Bowers et al. [2]. This method approximates the final result of the Poisson equation and in this paper is accelerated using Optix and dedicated ray tracing hardware. Using this method yielded a render time decrease of around 8 times between the Quadro P1000 and RTX 2060, while retaining the complex colour gradients and ease of use of traditional diffusion curves solved using the Poisson equation.

1 Introduction

From old pixel sprites to mipmaps, insufficient image fidelity is a problem that comes up in a host of different situations. In these situations vector graphics offers a solution. Vector graphics are an improvement over standard representations in the sense that they do not store an image at an explicit resolution. This is possible because vector graphics store their graphical information as a set of shapes and definitions on how to colour the areas that they surround or sometimes how to colour the area around the shapes.

Research into vector graphics is roughly divisible into two separate directions. Firstly Improving the efficiency of rendering more traditional vector graphic representations such as Support Vector Graphics (SVG) [17], PostScript and Portable Document Format (PDF). Secondly to find novel solutions to make the format more flexible the most notable example being diffusion curves [12].

Traditional vector graphics are usually tedious to use since they do not support complex colour gradients. These graphics use paths and simple shapes as a boundary which then can be filled using a flat colour fill or simple linear or radial gradients [17]. This means that we need a lot of primitives to make a realistic looking image. Needing a lot of primitives is undesired since it negates the effect of compact storage and makes creating images in the format a tedious task.

Diffusion curves set out to solve this problem by not defining the colour areas of the image explicitly but instead assigning a colour to each side of a path. These colours are then diffused into the image by solving complex differential equations. A visual aid for this process can be dropping food colouring into water and using the final result as the image. This method gives stunning visual results, but getting to the final image is slow ranging from 0.5 [4] to 4 seconds [1].

An ideal solution would be a method which has the rendering speed of traditional vector graphics and the flexibility of diffusion curves. This would allow the user to interact freely in real time with the final image instead of the current standard of a limited detail version, while still maintaining the ease of use of diffusion curves.

This solution might be possible with the approximation described by Bowers, Leahey and Wang. They showed that if we consider the curves light sources in a 2d global illumination problem we can get a reasonable approximation of the solution acquired using the Poisson Equation [2]. Their method achieved an impressive speed-up of 2x, this however is still far from the ideal solution. Recent developments in hardware acceleration might be able to give another significant speed-up. Namely, the NVIDIA RTX cards with their dedicated ray tracing cores can help us [11].

This leads to the research question :

> *Can the method of Bowers, Leahey and Wang to approximate the final product of diffusion curves be used in combination with the dedicated ray tracing hardware to achieve real time rendering of diffusion curve based graphics ?*

Thus, the main contribution of this work is to see if the performance of rendering diffusion curves can be further improved without significant loss of image quality. To achieve this Optix will be used which can access the dedicated ray-tracing cores on the NVIDIA RTX cards and has not yet been used.

Achieving this goal in combination with the ray tracing aspect would mean that the image can also be rendered at an arbitrary point in a very short amount of time. With random-access rendering at a good speed, diffusion graphics could be integrated into 3d rendering where its graphics could be used as textures as shown by Sun. et al. [16] and Jeschke, Cline and Wonka [5].

Increasing the speed of diffusion curve rendering would also allow instant feedback for drawing software which uses this as their primitive. With instant feedback diffusion curves become usable by artists since they do not have to guess for the end-result.

The rest of this paper is structured as follows: Previous work will be discussed in Section 2, the concept of ray tracing diffusion graphics will be explained in Section 3, how to integrate this with Optix will be discussed Section 4, the results will be shown in Section 5, Section 6 discusses the reproducibility, Section 7 will reflect on the results, and lastly Section 8 will conclude the research.

2 Related work

Diffusion curves were originally formulated by Orzan et al. [12] they are defined as lines with zero width and a colour on either side. These colours define the initial condition for a partial differential equation called the Poisson equation and the colour gradient across the curve serves as the boundary condition for this equation. Solving this PDE gives an image with a colour gradient which is as low as possible at every point. This produces an image with a smooth colour gradient everywhere except where were the curves define discontinuities.

While research which focuses on accelerating traditional vector graphics using the GPU is plentiful [7][8][10]. Research into diffusion curves is scarce and reseach into accelerating the rendering of diffusion curves is even rarer.

Methods described in papers focusing on path rendering acceleration such as stencil then cover [7] and scanline rasterization [8] rely on comparatively simple in-out checking thus their techniques are not applicable for the purposes of diffusion curves. These methods work for traditional vector graphics since all pixel which should be coloured must be surrounded by a path or be part of the border of such a path, while for diffusion curves this is not the case.

After the original formulation of diffusion curves several extensions have been proposed. Diffusion barriers which improve the usability for artist by not having to define a colour along every point along the curve was done by Bezzera et al. [1]. Besides, this they also proposed directional diffusion which guides diffusion into a certain direction, and different diffusion strengths along the curve.

The second extension of diffusion curves was Poisson vector graphics which introduced Poisson regions and Poisson curves which allow the user to control extrema. This allows for easy specular highlights and halos. Together with the previous extension it makes diffusion curves much easier to use, and consequently the images these extensions produce are a lot better. The image quality they provide however, comes at a cost in performance.

These performance issues arise from the fact that the PDE defining the image needs to be solved. While ordinary differential equation have relatively simple numerical approximation techniques such as the Euler method and the Runge-Kutta method, similar techniques do not exist for PDEs due to their nature of multiple, in general inseparable, variables depending on each other. This means more complicated methods are necessary. Such methods consist of methods based on the fast Fourier transform [13, p. 857][6], or the multigrid method [13, p. 871][9]. The multigrid method is the most prevalent solution to render diffusion curves as it has been used by Orzan et al. [12]. This method works by finding a very coarse solution first which it uses to accelerate the convergence of the solution at higher resolutions.

To alleviate the performance problem of solving PDEs Bowers, Leahey and Wang proposed a method which can be used to approximate the image without solving a differential equation [2]. In their approximation the diffusion curves are light sources and the light each pixel collects from these light sources is an approximation to the colour it is assigned by solving the Poisson equation. Their method was a success since it almost doubled the previous frames per second (FPS). However, it exhibits some artefacts which do not occur when directly solving the Poisson equation.

Building on this approach, Prévost, Jarosz and Sorkine-Hornung devised a method which uses a triangle mesh generated by a constrained Delaunay triangulation. This allows for only evaluating the colour on the vertices of this mesh and interpolating the colour.

The method of Bowers Leahey and Wang will also be used in this paper, but it will be sped up by using dedicated ray tracing hardware which has not been done before.

3 Diffusion curves and their approximation

Diffusion curves are defined in their simplest form as cubic Bézier curves with a colour on both sides. Cubic Bézier curves are piecewise polynomial splines defined using four control points per piece. The first and the last of these control points are interpolated and the other two control points guide where the curve should go. To find the curve defined by control points P_1, P_2, P_3 and P_4 these points need to be multiplied by the basis matrix for Bézier splines and the monomial basis vector up to and including degree 3 as shown in Equation 1.

$$f(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} \quad (1)$$

This function has the domain $u = [0, 1]$ and defines the entirety of the curve. To vary colour along the curve an arbitrary amount of colour control points with associated values for the curve parameter u are allowed. These colour control points are then linearly interpolated along u.

Rendering these curves requires solving the Poisson equation where the diffusion curves define the boundary and initial conditions. With these simple rules impressive images such as Figure 1 can be created.

This process is slow but, it can be sped up using approximations. The specific approximation used in this paper is one where the problem is transformed into a global illumination problem as formulated by Bowers, Leahey and Wang [2]. In this method the diffusion curves are light sources. The estimated light each pixel receives from these curves is our goal and can be found by solving a version of the rendering equation seen in Equation 2. Here $I(p)$ is the illumination of pixel p and the final colour of this pixel. the function $ch(p, \theta)$ returns the closest hit point for pixel p and ray angle θ and the R and W functions return the radiance/colour of the that

Figure 1: Image generated using method of Orzan et al. [12] which solves the Poisson equation. The image only contains diffusion curves which have only colour control points.

point and the normalized weight due to distance of the colour respectively.

$$I(p) = \int_0^{2\pi} R(ch(p,\theta))W(ch(p,\theta))d\theta \tag{2}$$

Solving this equation can not be done analytically, and we will approximate the solution by sampling it uniformly. This leads to Equation 3 where the integral is replaced by a summation and N is the number of samples that is used to estimate the colour of each pixel.

$$I(p) = (\sum_{i=0}^{N} R(ch(p, \frac{2\pi i}{N}))W(ch(p, \frac{2\pi i}{N})))/N \tag{3}$$

The uniform sampling will be done by ray tracing from each pixel and using the closest intersection with the diffusion curves as the point to gather the weighting and colour information from. After which this information can be displayed as the final image. Because this method is intuitive, easy to understand, leads to good quality images and can benefit greatly from hardware acceleration this method is chosen to be implemented in Optix.

4 Using Optix

Implementing the algorithm proposed by Bowers, Leahey and Wang [2] into Optix is not as straightforward as one might hope. Several difficulties come into play regarding ray collision with the curves, primitive types of the rays, and artefacts occurring.

Intersecting rays with Bézier splines is non-trivial for this reason Bower, Leahey and Wang used an increasingly fine linear approximation for the curves. However, A lot of research has been done about how to efficiently intersect cubic splines and Optix already implements such an algorithm for us. This algorithm is most probably based on work by Reshetov [15]. And takes a set of three-dimensional cubic B-spline curves as input and produces intersections at any curve and/or at the closest curve for a given ray. To leverage the speed of this algorithm the curves will not be approximated but will be intersected directly.

4.1 Intersecting curves

The algorithm implemented in Optix however has two mismatches with our input. The first of these mismatches is due to the dimensionality of the splines. Optix is build for three-dimensional ray tracing while diffusion graphics are two-dimensional. Since splines do not mix dimensions, the function in each dimension can be evaluated separately, this is easily fixed by setting the z coordinate of each control point to zero.

The second mismatch is somewhat more complex. Namely, it requires unclamped b-splines. This is a slightly different primitive than the Bézier splines which are the input of the algorithm of this paper. There are two ways to alleviate this problem.

The first option is the quick-and-dirty approach, where we take the control points for the Bézier curve and interpret them as b-spline control points. This can be a reasonable approximation since both types are cubic splines. However, these b-spline curves are unclamped which means they do not interpolate any of the control points i.e., they never intersect them. This was already foreseen by the Optix developers and in their programming guide [3] they propose to use phantom points, a reflection of the point preceding the end point through the end point of the curve. This means we get an extra b-spline consisting of the first phantom point and the first three points of the original curve if we want to clamp the start. Similarly, we can repeat this process at the end of the curve to clamp it there as well.

This method thus adds two extra primitives to each Bézier spline. Besides the performance hit that this has, the error is also relatively big as can be seen in Figure 2.

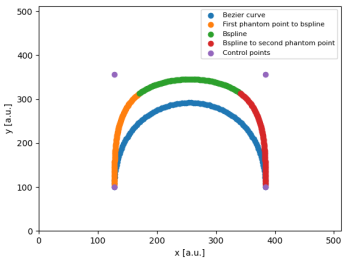


Figure 2: An example of a b-spline curve and a Bézier curve using the same control points. The green curve represents the b-spline curve and the orange and red parts are the curves added by adding phantom points to clamp the b-spline. The blue curve is the Bézier curve generated from the control points. The x and y-axis are in arbitrary units.

This leads to the second method which transforms the control points of the Bézier curve in such a way that they form the control points for a b-spline which is exactly the same as the Bézier curve. To achieve this we need to circumvent the multiplication with b-spline basis matrix and instead use the Bézier spline matrix. Since Optix has the monomial basis vector and the b-spline basis matrix pre-multiplied we need to transform our points using the inverse b-spline basis matrix and then multiply them by the Bézier spline basis matrix as shown in Equation 4. In this equation $f(u)$ is the function describing the Bézier-spline, M_b and M_b^{-1} are the b-spline basis matrix and its inverse respectively and M_{bez} is

the Bézier basis matrix. simplifying this equation would lead to Equation 1.

$$f(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} * M_b * (M_b^{-1} * M_{bez} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}) \quad (4)$$

Now the Bézier splines are transformed into b-splines, and we can use the algorithm Optix provides. This allows us to find the intersection but not yet the colour information.

4.2 Interpolating values

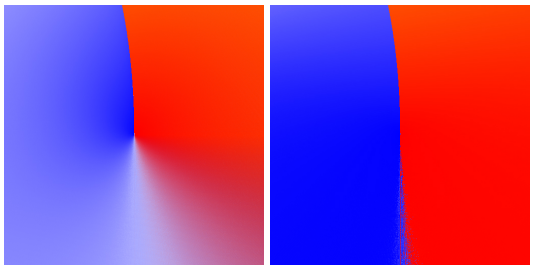
Now the intersection point is known some value needs to be returned. First it needs to be determined on which side the ray hits the curve since the curve has different colours on either side. Secondly the value needs to be interpolated to the intersection point and lastly this value needs to be returned with an appropriate weight.

Since Optix only provides the value of the curve-parameter at the point where an intersection occurred, we need to manually determine which side the ray has hit. This can be done by comparing the ray direction and the curve normal. To determine the curve normal we take the gradient of the curve which can be analytically calculated and rotate it. Determining the side on which the ray hits is now a simple case of checking whether the sign of the dot product between the normal and ray direction is positive or negative.

Another obstacle occurs due to only getting the curve parameter of the intersected spline, which is between 0 and 1. This is a problem since the colour properties are stored for the entire curve only and each spline in the curve adds 1 to the parameter used for interpolation. This can be fixed by maintaining a list, which maps each Bézier spline to the curve it is part of and a map for each curve in what order the splines are. These two maps allows for calculating of the total curve parameter and the curve properties can be deduced by linearly interpolating the values of the curve along the curve parameter.

The last problem after hitting a curve is determining the weight function $W()$ in Equation 3. This function should intuitively take the distance until the hit into account. Bowers, Leahey and Wang [2] propose $|x-p|^{-c}$ where $|x-p|$ is the distance between the intersection point x and the pixel p and c is a constant set for the entire rendering process, which determines the weight fall-off. This works well for dense graphics however for very sparse graphics this may lead to artefacts. For this reason, we also tested $w_c c_c^{|x-p|}$ where w_c and c_c are curve parameters. This lead to slightly better results for eliminating rays that have too much weight (Figure 3). However, due to the exponential nature w_c and c_c were very sensitive and a useful image would only appear with c_c very close to 1 (0.999) and w_c around 100000, was the minimum to get an image like Figure 4. This led to the choice of using a polynomial weight function like in the original method [2] but letting the degree be a curve parameter to allow for extra customization. The final weight function is thus $w_c|x-p|^{-c_c}$.

Another difference in the weight function compared to previous work is the ability to change the effect of distance. Previous work [1][2] only allowed the change of a constant multiplier. Now it is possible to create areas, which will have a high weight nearby but quickly drop off because another area, being less affected by distance, takes over. An example is shown in Figure 4.



(a) $W(x,p) = |x-p|^{-0.5}$ (b) $W(x,p) = 0.8^{|x-p|}$

(c) $W(x,p) = |x-p|^{-7}$

Figure 3: A comparison between three different weight functions $W(x,p)$ at the end of a diffusion curve. The rays to the right of the curve can never hit any red and no blue can be hit for the rays on the left. this means the colours will never mix, a method to fix this is shown in subsection 4.5. The rays underneath the curve will mostly hit a white area outside the view port causing a white area to appear where the rays hit neither the blue nor red side of the curve. This can be reduced by using weight function with very fast drop-off like an exponential, however a high degree polynomial works as well.

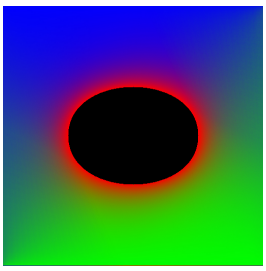


Figure 4: An example of different ways to use the weight function generated from weight_demo.xml . The red area decays quickly due to a high c_c but is still expressed due to a high w_c and thus has a small gradient with the green and blue areas. The green and the blue areas are a straight curve at the top and bottom respectively.They have a lower c_c and thus decays slower leading to a larger gradient they also vary w_c along the x-axis.

4.3 Gaussian blur

While the method discussed in the previous section already produces great images Orzan et al. also proposed interpolated Gaussian blur. This blur is defined in such a way that just like

the colours a sigma value for the Gaussian blur is defined along the diffusion curve using control points. This value is then diffused into the image to create a blur map. In this paper this blur map is created in the same manner as the colours are accumulated per pixel. The resulting blur map is then used to define a Gaussian kernel implemented in CUDA using a vertical and horizontal pass to blur the image. This can be used to recreate the effect of depth of field on a diffusion curve image. A comparison between no blur and blur can be seen in Figure 5.

Figure 5: A comparison of the lady_bug.xml [12] rendered with and without blur left and right respectively. The blur in the left image is reducing the edge sharpness only at the flowers. This gives it an effect like it is out of focus just like a real photograph would.

4.4 Portals and transparency

Bezzera et al. [1] proposed a way to connect two areas within the image such that the diffusion process can cross between them even though they are spatially separated. This feature is also present in our implementation. This is done using curves which have the connects attribute set and thus connect 2 or more curves. These portal curves transfer a ray from one curve to the connecting curve preserving its direction with regard to the curve normal. This allows for the creation of lenses guiding the diffusion process in a specific direction. For these curves the colours are still used, but instead of adding colour to the ray it uses the RGB values as a colour filter. This together with the fact that a portal can connect to itself, essentially letting the ray pass, allows for partially transparent diffusion curves, filtering out only a specific colour. A demonstration of these portals are shown in Figure 6.

4.5 Artefact reduction

Using ray tracing for rendering diffusion curves does not only have positives. It also introduces several artefacts. While these artefacts are mostly limited to areas with very sparse diffusion curves some are really obvious and grossly disturb the image. Two of the effects are a direct consequence of switching the implementation from solving the Poisson equation to ray tracing and the last is due to diffusion curves needing to have a width to be implemented in Optix.

The first defect is caused by ray tracing is shown in Figure 3, where it causes a light patch although this area should be purple. It is caused by the fact that the red and blue areas in this image can never mix since the rays cannot go around the corner and thus cannot have line of sight on both sides of the

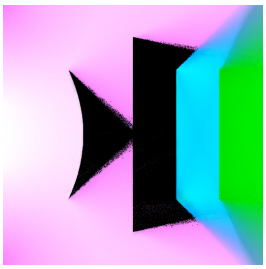


Figure 6: The PortalDemo.xml image rendered. In the middle is a diffusion barrier blocking the colour emitted by the diffusion curve on the far left. a portal connects the two sides and filters out the red light. The round left portal focusses the rays only on the white part of the curve. Finally, completely on the right there is a portal connecting to itself filtering the blue light out. Thus, the image has only one emitting diffusion curve but has wildly different colours.

curve. This artefact is thus only visible if there are no other curves nearby the endpoint of a curve. The problem only occurs in this situation because another curve can hide the fact that the colour transition is not really working. A possible solution to this artefact is to give the artist an option to add an endcap to a curve. This endcap will connect the left and right side in such a way that a ray has a greater area to hit and the rays originating from a pixel affected by this artefact can hit both a bit of the left side and a bit of the right side of the diffusion curve in question. This is achieved by creating a Bézier curve which starts and ends in the same point and the other two control points are on the top left and top right, creating a kind of drop shape as seen in Figure 7.

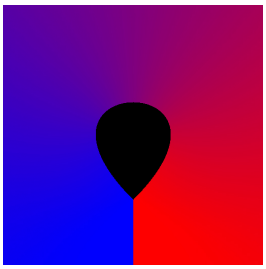


Figure 7: Image generated from endcap.xml showing an extra large endcap with a black inside to make the shape better visible.

The curve colour control points are chosen such that on the outside and inside this drop the colour interpolates in such a way that it smoothly transitions from the left colour of the curve to the right colour. this drop is then rotated to align with the tangent of the curve and translated to the correct spot at the endpoint of the curve. Creating a colour gradient connecting the left and right side of the curve as seen in Figure 8.

Another artefact is caused by the fact that Optix forces curves to have a width. This means if a pixel lays inside a curve it cannot hit that curve and consequently does not get the expected colour, but instead a radically different colour than either left or right of curve. Since pixels are typically 1000 times smaller than the curve width, we can vary the starting point inside the pixel such that the curve can be hit. Bowers, Leahey and Wang also implemented this but for dif-

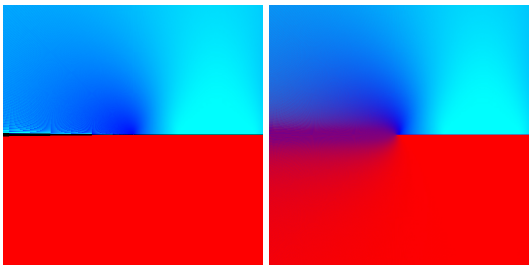


Figure 8: Comparison of without and with endcap on left and right respectively. The endcap improves the transition by making the colour transition smoother

ferent reasons it namely also provides some crude anti aliasing. The result of this technique is shown in Figure 9.

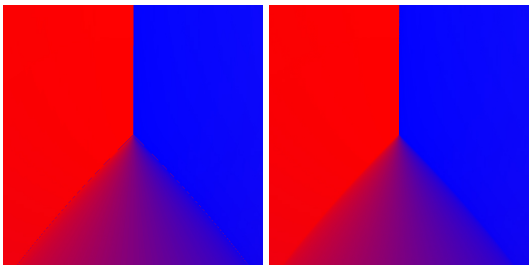


Figure 9: Comparison between an image without the randomization within the pixel on the left and with this feature on the right. We can clearly see some artefacts occur in the left figure, in the figure with random sampling inside the pixel this disappears.

Besides using this randomization technique for the edges the ray direction can also be randomized within its assigned angle. This allows circumvention of the image gradients which occur when only a few rays per pixel hit a curve which can occur at lower ray counts as shown in Figure 10.

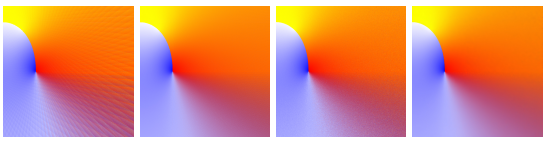


Figure 10: Comparison between two methods to reduce the effect of low ray count per pixel. From left to right we have an image generated with 128 rays per pixel, 512 rays per pixel, 128 rays per pixel and randomization of start location and angle within each pixel and the previous steps plus the Optix Denoiser. The endcaps were removed for this picture to give a stronger colour gradient to exaggerate the artefacts.

Although the discontinuities are gone after randomization, it does introduce a lot of noise in areas where only a few rays are hitting diffusion curves. To alleviate this the Optix denoiser is applied to the image. The Optix denoiser is an AI based denoiser originally meant to reduce the amount of samples required for path tracing 3d scenes. Having not enough samples in path tracing creates similar noise as we encounter. This means the results from this denoiser are quick and reliable for our purposes as well. The final image is shown in the right most image of Figure 10.

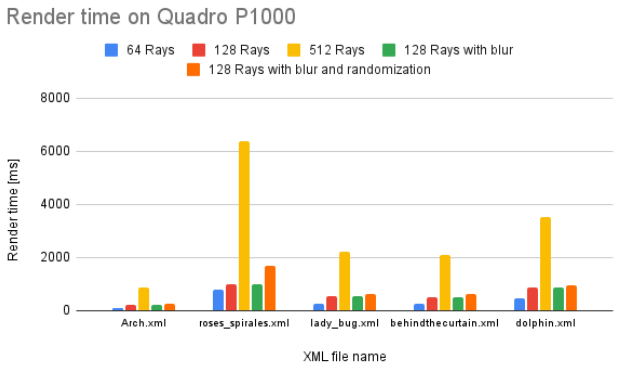
5 Runtime of the algorithm

To answer the research question it is important to get an indication of the runtime of the algorithm. To give an indication on the effects of the ray-tracing cores the renderer is tested on both a RTX 2060 and a Quadro P1000 card. The RTX card is supported by an Intel i5 10300H CPU at 2.5GHz and 16 GB RAM on Windows 10. The Quadro P1000 system has an Intel i7-8750H CPU at 2.2 GHz and 16 GB of RAM also on Windows 10. The latter components should however not be a bottleneck since the entire frame generation happens on the GPU. Several XML files, most from the original diffusion curve formulation [12], were chosen for this comparison. They range from 1 to 1500 diffusion curves per file and each curve consists of at least several splines. All were rendered in an 512 by 512 window with the picture centred. This means that some curves were not inside the view, however they were taken into consideration for the ray tracing. Besides the GPU split the results are also rendered with three settings for the amount of rays since some pictures do not suffer much from reducing the amount of rays due to a very dense curve layout. Lastly we evaluate performance with blur, without blur and randomization within each pixel together with denoising. The time required for portals remains untested since it does not contribute to the time if no portals are present and if they are present it will heavily depend on the size of and amount of portals. This separation is done since not all the previous work has all our features, and we wanted a comparison. The results of this experiment are shown in Figure 11 and the full results as table are shown in Appendix A Table 1 and Table 2.

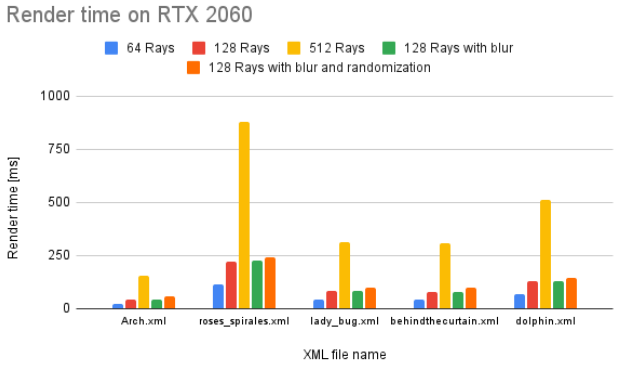
In Figure 11 we can see a clear difference between the RTX card which has dedicated ray-tracing cores and the Quadro P1000 which does not. This means simply porting the algorithm to Optix achieved an 8 times speed-up for systems which have RTX support, but the system is also viable with lower ray counts on systems that do not have this capability. We can also clearly see a linear relation between the amount of rays and the time required for rendering. Besides this exploiting the symmetry of the Gaussian kernel for a horizontal and vertical pass means that the time required for the Gaussian blur is virtually non-existent.

The setup time however is about four seconds the first time the program is run regardless of which image is used. This time is used for Optix to generate some cache relating to the module compilation, which is done at runtime. This lowers the setup time to between one and three seconds for the next run of the program. The exact times are shown in Appendix A which are taken after this initial setup run.

Because the Optix denoiser allows for much lower sample rates while retaining image quality, the limit of how much the sample rate can be reduced without noticeable artefacts was also tested. This showed that lowering the rays per pixel, to as low as 8 rays per pixel gives an image which, with the naked eye, is indistinguishable from one with 128 rays per pixel for the lady_bug.xml image. To investigate the differences further we took the RGB difference between 8, 16, 32 and 64 rays per pixel with the 128 ray per pixel variant the results



(a) Results for the rendertime per frame for the quadro p1000 system.



(b) Results for the rendertime per frame for the quadro RTX 2060 system.

Figure 11: Results showing the render time per frame for the Quadro P1000 and the RTX 2060 system, render time in milliseconds are on the y-axis. The results are clustered per XML file on the x-axis with each cluster consisting of a render with 64, 128 and 512 rays, 128 rays with blur and 128 rays with blur and randomization from left to right. The complexity of the XML files increases along the x-axis with 1, 20, 71,131 and 1521 diffusion curves respectively. Mind the 8 times difference on the y-axis between Figure 11a and Figure 11b. The roses spirales XML is much more difficult since it has more splines per curve and densely covers the entire screen. All files except Arch.xml are taken from work by Orzan et al. [12].

are shown in Appendix A, Table 3. This showed that the error was mostly along the edges, possibly because the Optix denoiser seems to create some visible artefacts around sharp edges.

6 Responsible Research

In this specific subfield of research there are not many ethical issues to consider. While there are certainly ethical concerns about how end-users could use the software created in this research, the extra concerns for this research are minimal. While all software used is freely available Optix only works on NVIDIA GPUs and thus not all users can run the software.

In contrast, reproducibility is a much bigger issue. Besides the fact that code-sharing is easy and only under specific circumstances impossible due to copyright issues, it does not happen enough in computer science as a whole and vector graphics is certainly not leading the charge on this issue. Out of all the citations in this paper only two shared their implementations [9][12]. For both of these it was a big success since the multigrid method from Real-Time Gradient-Domain Painting [9] is used in the solver of Orzan et al. [12]. The code publication of Orzan et al. made their diffusion curve primitives and art generated during their research the de facto standard for representation and test images.

Besides code-sharing hardware is also a contributor to lack of reproducibility. With the unbreakable connection between computer graphics and the graphics card, and the uncountable amount of different types of graphics cards, exact reproduction of results is nearly impossible even if the original code is used. What makes this even worse is the reliance on render time in ms as the main quantitative evaluation. Since GPUs are increasing in processing power and specialization quickly, this metric will increase over time without the code being changed.

While the second issue is an inherent problem of the field and this research also suffers from it, it can be mitigated by releasing the code. this allows future researchers to rerun the code on their own setup to make a fair comparison.

A second way to try and mitigate the second issue is to run the code on multiple devices and gather performance data for more hardware in this way. This has also been done for this project and led, although expected, to wildly differing results.

Since there is no specific copyright licence required for this project the source code can freely be made available online under an Apache V2.0 licence where specific files are under the same licence but published in other work. This means that any future work can re-use this code for their project. This makes sure the first issue is not applicable to this project. The source code is made available in the following GitHub repo : https://github.com/MikaZeilstra/RaytracingDiffusionCurves.

7 Discussion

The measured runtime of the results shows that the algorithm is very fast compared to previous work. Extra effort was also put into trying to hide the artefacts caused by ray tracing. Comparison with previous work will thus focus on these two aspects.

7.1 Speed comparison

As seen in Figure 11 the algorithm performs at about 98 ms per frame or about 10 FPS for the lady_bug XML file. This file is comparable in number of diffusion curves to other artist drawn images at 72 curves in the image. Each of these curves consists of tens of splines. The 10 FPS reached at 128 rays per pixel. This ray count is chosen since it is the same Bowers, Leahey and Wang used [2] and typically sufficient for artist created images. The 10 FPS reached with these settings includes blur and pixel and ray randomization. This is an almost 4 times increase in speed over the high quality implementation used by Bowers, Leahey and Wang [2].

Another algorithm we can compare against is the work of Prévost, Jarosz and Sorkine-Hornung [14]. Their algorithm had a total render time per frame of 74 ms for the

lady_bug.xml image at 1024 by 1024 resolution. This is really impressive and to compare against this higher resolution an extra test was ran with 8 rays per pixel. This slightly reduces the quality near the edges and can introduce some inconsistencies between subsequent frames. This test showed that for the setup with 8 rays and all the other features the image can be rendered in 74 ms for the lady_bug.xml and thus tying this the previous work.

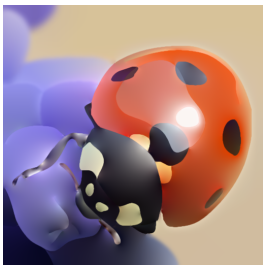
This new-found speed allows rendering of images with significantly more curves such as the dolphin image. This means we could use diffusion curves as some kind of compression.

Besides the previous point it also allows interaction with the final product of an image without delay. Interaction means that artists get better tools to work on diffusion curve images and thus better art can be created.

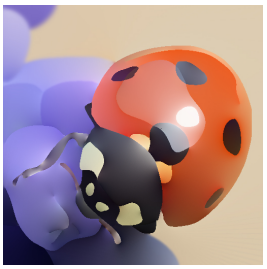
7.2 Image quality

But not only the speed of the algorithm is important the quality of the produced image is important as well. While the images look visually similar, there are some obvious and some more subtle difference that are revealed in the absolute difference between the images as seen in Figure 12.

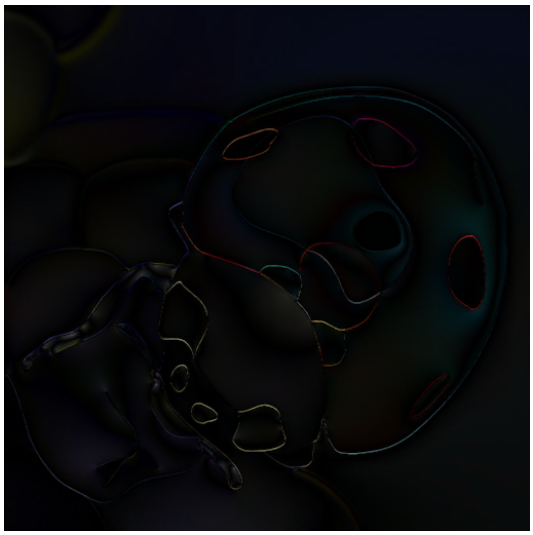
At first glance we can see that the image generated using the Poisson method is slightly darker than the other image and that some gradients are slightly different. This can all be tweaked by optimizing the weighting parameters of each curve to get a closer resemblance to the preferred look. But when we take a closer look at the absolute difference image in Figure 12c it also becomes clear that the blur and the edges are different. The difference in the blur is due to the implementation of the blur filter, which is different since the implementation needed to be guessed. Differences along edges are a recurring effect also observed by Bowers, Leahey and Wang [12].



(a) lady_bug.xml rendered using the method of orzan et .al [12]



(b) lady_bug.xml rendered using the proposed method



(c) the absolute difference between the images

Figure 12: Comparison of lady_bug.xml [12] rendered using the method of Orzan et al. [12] and the proposed method. The rendered images are shown on top and the absolute difference is shown in the large image.

8 Conclusions and Future Work

With the main research question in mind : Can the method of Bowers, Leahey and Wang to approximate the final product of diffusion curves be used in combination with the dedicated ray tracing hardware to achieve real time rendering of diffusion curve based graphics? it can be concluded that this is most definitely possible since the application reaches 10 FPS on typical artist created images. The use of ray tracing hardware provided an 8 times speed-up over the same implementation without this specialized hardware. However, the achieved rate of 10 FPS is still on the slower side but reaches interactability without giving up a lot of quality.

The speed-up also increases the effectiveness also in other ways. One of these ways is in allowing for more diffusion curves in the image while maintaining acceptable performance. This allows photo-realistic images to be shown and possibly compressed by using methods which allow raster images to be converted to diffusion curve images.

Besides this there are still some aspects left for future work. Namely, the intersection algorithm, currently the Optix default b-spline intersection algorithm is used, but this requires some workarounds and works in three dimensions which is not really necessary. Optix allows for custom intersection methods and in the future this should definitely be used meaning future version do not need to give the curves width and achieve extra speed-up by dropping support for 3 dimensions.

Another possibility is now that the viability of Optix is shown for diffusion curve images, is to also implement an improved version of the work by Prévost, Jarosz and Sorkine-Hornung [14]. Their algorithm calculates the colour of the image at a lot less points taking sparsity of the curves into account and thus a lot less rays are being traced. This makes their method as fast as the implementation in this paper, However their main bottleneck is still ray-tracing as it takes up about 50% of the time required. Thus using Optix can drastically improve the runtime of their algorithm.

References

[1] Hedlena Bezerra, Elmar Eisemann, Doug DeCarlo, and Joëlle Thollot. Diffusion constraints for vector graphics. In *NPAR 2010: Proceedings of the 8th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '10, page 35–42, New York, NY, USA, 2010. Association for Computing Machinery.

[2] John C. Bowers, Jonathan Leahey, and Rui Wang. A ray tracing approach to diffusion curves. In *Proceedings of the Twenty-Second Eurographics Conference on Rendering*, EGSR '11, page 1345–1352, Goslar, DEU, 2011. Eurographics Association.

[3] Nvidea Corporation. Nvidia optix 7.3 – programming guide, Apr 2021.

[4] Fei Hou, Qian Sun, Zheng Fang, Yong-Jin Liu, Shi-Min Hu, Hong Qin, Aimin Hao, and Ying He. Poisson vector graphics (pvg). *IEEE Transactions on Visualization and Computer Graphics*, 26(2):1361–1371, 2020.

[5] Stefan Jeschke, David Cline, and Peter Wonka. Rendering surface details with diffusion curves. *Transaction on Graphics (Siggraph Asia 2009)*, 28(5):1–8, December 2009.

[6] Jose L. Jodra, Ibai Gurrutxaga, Javier Muguerza, and Ainhoa Yera. Solving poisson's equation using fft in a gpu cluster. *Journal of Parallel and Distributed Computing*, 102:28–36, 2017.

[7] Mark J. Kilgard and Jeff Bolz. Gpu-accelerated path rendering. *ACM Trans. Graph.*, 31(6), November 2012.

[8] Rui Li, Qiming Hou, and Kun Zhou. Efficient gpu path rendering using scanline rasterization. *ACM Trans. Graph.*, 35(6), November 2016.

[9] James McCann and Nancy S. Pollard. Real-time gradient-domain painting. *ACM Trans. Graph.*, 27(3):1–7, August 2008.

[10] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. *ACM Trans. Graph.*, 27(5), December 2008.

[11] NVIDIA. Nvidia turing gpu architecture. Technical report, NVIDIA, 2018.

[12] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion curves: A vector representation for smooth-shaded images. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, volume 27, 2008.

[13] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, hardcover edition, 10 1992.

[14] Romain Prévost, Wojciech Jarosz, and Olga Sorkine-Hornung. A vectorial framework for ray traced diffusion curves. *Computer Graphics Forum*, 34(1):253–264, 2015.

[15] Alexander Reshetov. Exploiting budan-fourier and vincent's theorems for ray tracing 3d bézier curves. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Xin Sun, Guofu Xie, Yue Dong, Stephen Lin, Weiwei Xu, Wencheng Wang, Xin Tong, and Baining Guo. Diffusion curve textures for resolution independent texture mapping. *ACM Transactions on Graphics - TOG*, 31, 07 2012.

[17] Eric Willigers, Amelia Bellamy-Royds, Chris Lilley, Dirk Schulze, David Storey, and Bogdan Brinza. Scalable vector graphics (SVG) 2. Candidate recommendation, W3C, October 2018. https://www.w3.org/TR/2018/CR-SVG2-20181004/.

A Results Tables

Name	Frames	frametime	setup
Arch			
64	291	118,8	3111
128	86	227	1131
512	58	878,2	1099
128-blur	59	223	1117
128-AA	74	274,8	2560
Roses spirales			
64	42	799,5	1077
128	27	1002	2214
512	27	6387	1105
128-blur	23	1005	1116
128-AA	28	1688	1340
lady_bug			
64	892	282,9	1151
128	226	560,4	1148
512	53	2223	1080
128-blur	97	558,4	1094
128-AA	99	630	1333
behind The curtain			
64	104	268,2	1077
128	63	525,5	2214
512	22	2085	1105
128-blur	90	527,6	1100
128-AA	91	630,86	1305
dolphin			
64	393	447	1219
128	104	882	1187
512	43	3524	1111
128-blur	41	890,4	1175
128AA	67	946,2	1327

Table 1: Render time on the Quadro P1000 system. The name of each image, the amount of rays and possible extra features is specified in the first column. The amount of frames rendered and the average time to render each frame for all XML variants is shown in the second and last column respectively. In the last column the setup time or time before the first frame is rendered is shown.

Name	Frames	frametime	setup
Arch			
64	381	21,71	2540
128	325	40,97	2590
512	184	156,9	2567
128-blur	337	44,85	2997
128-AA	312	57,23	3056
Roses spirales			
64	259	113,3	2591
128	637	223,7	2508
512	29	878,2	2532
128-blur	128	224,3	2516
128-AA	94	241,8	2127
lady_bug			
64	274	42,18	2599
128	132	81,74	2529
512	69	315,6	2511
128-blur	105	82,35	2547
128-AA	165	98,7	2552
behind The curtain			
64	126	41,52	2364
128	155	79,34	2341
512	218	307,8	2521
128-blur	150	81,06	2525
128-AA	125	99,87	2576
dolphin			
64	177	66,32	2403
128	260	128,7	2311
512	73	510,7	2319
128-blur	112	130,3	2566
128AA	163	146,2	2557

Table 2: Render time on the RTX 2060 system. The name of each image, the amount of rays and possible extra features is specified in the first column. The amount of frames rendered and the average time to render each frame for all XML variants is shown in the second and last column respectively. In the last column the setup time or time before the first frame is rendered is shown.

Number of rays	Max difference	MSE	Absolute Difference
64	66	1,4403	455321
32	63	1,6843	513393
16	54	2,0469	583851
8	57	2,6783	693640

Table 3: A comparison of several different quality of renderings of the lady_bug.xml with a version rendered with 128 rays per pixel. The first column contains the amount of rays used per pixel, the second column displays the maximum difference between the image and the 128 version, the third column shows the mean squared error (MSE) and the last column the absolute error for the 512x512, 8 bit, 3 color channel image.