FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Shape Interpolation Using Diffusion Isosurfaces

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Florian Spechtenhauser
Matrikelnummer 0826226

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dr. Stefan Jeschke

Wien, 04.02.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Shape Interpolation Using Diffusion Isosurfaces

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Florian Spechtenhauser
Registration Number 0826226

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dr. Stefan Jeschke

Vienna, 04.02.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Florian Spechtenhauser
Neuwaldeggerstraße 18-18a, 1170 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

I present a diffusion based shape interpolation method which is applicable to 2D and 3D surfaces. As input, 2D shapes are represented as diffusion curves, the 3D shapes are simply 3D meshes. The algorithm generates an exact Voronoi diagram of two surfaces along with a distance map, both are stored as textures for further manipulation and lookup. The Voronoi diagram is used as starting point for the iterative color diffusion. After the diffusion step, an isovalue can be applied to the resulting texture. By varying the isovalue, different intermediate surfaces between the two input surfaces arise. In 2D, Diffusion Curves [3] are used as input, whereas in 3D, textured surface meshes as used. This shape interpolation method is applicable to every kind of shape that can be represented by diffusion curves or 3D surface meshes.

Contents

CHAPTER 1

Introduction

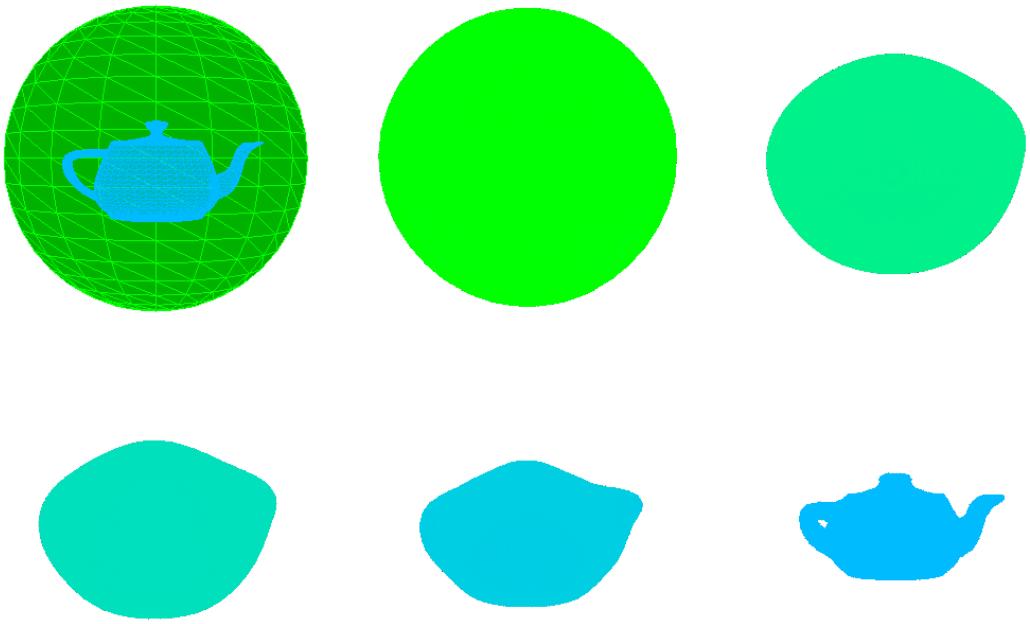


Figure 1.1: 3D shape interpolation sequence with different isovalues of the isosurface. Surfaces are a sphere colored in green and the classic CG-teapot in blue (top left). The resolution of the 3D texture is 256^3. The isovalues are: (top middle) 0.0001, (top right) 0.25, (bottom left) 0.5, (bottom middle) 0.75 and (bottom right) 1.0.

In shape interpolation, intermediate surfaces between two input surfaces are constructed. These intermediate surfaces are geometrically close to both input surfaces. Depending on a specified weight (in my case the isosurface value), the intermediate surface can be closer to one surface than to the other. The sequence of the resulting intermediate shapes can also be transformed into an animation, which, when played, shows a deformation from one surface to the other, called shape morphing. Shape interpolation can be applied to objects in any dimension, but the most common cases are the 2D and 3D shape interpolation. Figure 1.1 shows a 3D shape transformation sequence using diffusion isosurfaces.

Shape interpolation can be useful in medicine, computer aided design and special effects creation [4]. An example for a useful shape interpolation in 2D can be found in medicine. Usually, image techniques collect data of a patient's internal anatomy in slices. As the sampling often has a low resolution in the third dimension, only few slices are generated and so an interpolation has to be made. In each of these slices, the contours of the organs are created. Then, intermediate shapes are obtained by interpolating the contours of the actual slices by applying a 2D shape interpolation method. In computer aided design, shape interpolation methods can be useful for creating smooth surfaces. An example is the creation of a joint between two metal parts with different cross-sections. The connecting surface has to be smooth, so that cracks are not formed easily. The intermediate surface between these two metal parts can be obtained using shape interpolation. A third application for shape interpolation can be found in film industry. Here, for the sake of special effects, animated shape interpolations are used to morph between two objects.

I introduce shape interpolation in 2D and 3D. As input, I use diffusion curves [3] to define shapes in 2D, in 3D the shapes are defined as 3D meshes. Depending on the dimension, I either get a 2D or 3D texture as output, which represents an intermediate shape.
The first step in my approach is to generate a Voronoi diagram of the two surfaces, which results in a 2D or 3D texture, depending on the dimension of the input shapes. From this texture as starting point, a smooth spatial function is defined by applying a diffusion algorithm. After this step, isovalues are applied to obtain intermediate surfaces.
The problem I want to solve is not only to interpolate shapes in their forms, but also in their colors. To solve this, the color information needs to be saved in the Voronoi diagram and in the texture after applying the diffusion algorithm. Since we need the RGB channels of the textures to store and interpolate the color, the information to interpolate between the shapes is stored in the alpha value of the textures. After the intermediate shape is retrieved from the diffused texture by applying an isovalue to the interpolated alpha values, the color of the resulting shape is obtained from the RGB channels of the diffused texture.

CHAPTER 2

Related Work

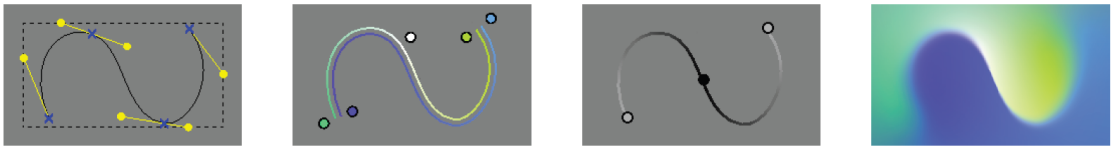
There exist various methods for 2D and 3D shape interpolation. Turk and O'Brien [4] introduced a shape transformation method using variational implicit functions, by casting the transformation between two N-dimensional objects as a scattered data interpolation problem in N+1 dimensions. The shape transformation between two 2D shapes takes place within two planes, one for each shape, which are placed parallel to one another in 3D. The slices between these two planes in the 3D space represent intermediate shapes, which are created with a variational interpolation technique. A similar approach can be used for 3D shapes by solving a 4D interpolation problem. My implementation has the same motivation as the shape interpolation method introduced by Turk and O'Brien [4] and is applicable to the interpolation of 2D and 3D shapes. The input of my 2D shape interpolation method, diffusion curves are used [3]. For the Voronoi diagram generation I used the approaches of Hoff et al. [1].

CHAPTER 3

2D Shape Interpolation

In the 2D shape interpolation, diffusion curves as defined by Orzan et al. [3] are used as input. From these, 2D shapes are generated. By diffusing these shapes, a smooth spatial function is defined. Intermediate shapes are then obtained by applying isovalues to the function. For the 2D implementation I used the application source of Jeschke et al. [2], which has shapes defined by diffusion curves as input and solves the diffusion in a user defined number of steps. However, for my algorithm, I have two sets of diffusion curves, each set represents a shape. The colors of one shape are then set to black, the colors of the other shape are set to white, so the resulting diffusion between these two shapes results in a greyscale image (Figure 3.6). The isovalue then represents a grey value which divides the image in two parts, black for the values darker than the isovalue and white for the others.

3.1 Diffusion Curves



(a) (b) (c) (d)

Figure 3.1: A Diffusion curve as defined by Orzan et al. [3]. (a) shows the geometric curve described by a Bézier spline, (b) shows the defined colors at each side of the curve, (c) shows the blur control points and (d) the diffused image.

For the 2D diffusion algorithm, diffusion curves as presented by Orzan et al. [3] are used. A diffusion curve is a vector based primitive for creating smooth-shaded images. Orzan et al. define it as a curve in the image space, which partitions the space through which it is drawn,

defining colors on each side of the curve. Diffusing these colors creates a smooth-shaded image. The diffusion curves are defined such as colors can vary smoothly along a curve, which make them very flexible for defining an image. Figure 3.1 shows a diffusion curve as presented by Orzan et al. [3].

The diffusion curve is a geometric curve defined as a cubic Bézier spline. Such a Bézier spline is specified by a set of control points. As additional information, two sets of color control points are added, one set for each side of the curve. As last addition a set of blur control points is added, which control the smoothness of the transition between the two sides of the curve. This color and blur attributes are then interpolated between the control points. Orzan et al. [3] use linear interpolation and colors in RGB space, as they can easily be mapped to an efficient GPU implementation.

The diffusion curves only define rules on how the rest of the canvas is filled. As next step we need to fill the canvas by generating a Voronoi diagram of the diffusion curves. In this step the canvas is filled with the colors of the diffusion curves which are nearest to the current pixel. To get a smooth image, the resulting texture is diffused.

The following paragraphs describe the generation of the Voronoi diagram (3.2) and the diffused images (3.3) in more detail. Paragraph 3.4 describes how the interpolated shapes are obtained from such diffused images, paragraphs 3.5 and 3.6 show some results and the performance respectively.

3.2 Voronoi Diagram Generation in 2D

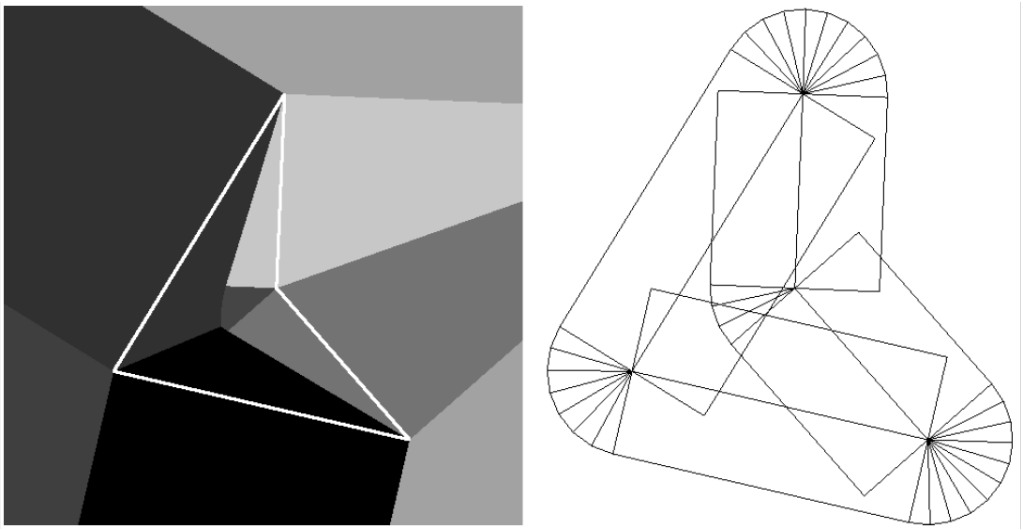


Figure 3.2: A Voronoi diagram and the corresponding distance mesh, generated by Hoff et al. [1]

A Voronoi diagram is defined as follows: Each pixel in the image has the color of the nearest primitive of this pixel. In our case the primitives are diffusion curves which are transformed into

linear segments. The pixels of the Voronoi diagram image get the color of the closest diffusion curve segment.

For the Voronoi diagram generation I used the algorithm of Hoff et al. [1]. This algorithm works in the following way: For each segment of the tessellated curve a mesh of the distance function of the segment is created (Figure 3.2). These *distance meshes* of each segment are stretched to overlap the whole region of influence of the segment in the output texture. The Voronoi regions then emerge automatically by using the z-buffer comparison, as the distance of the distance meshes are written into the z-component of the pixels.

Distance meshes represent the region of influence of a specific segment of the curve. The generation of the distance mesh depends on the type of the primitive. In the case of a tessellated diffusion curve there have to be handled two types of primitives: line segments and points.

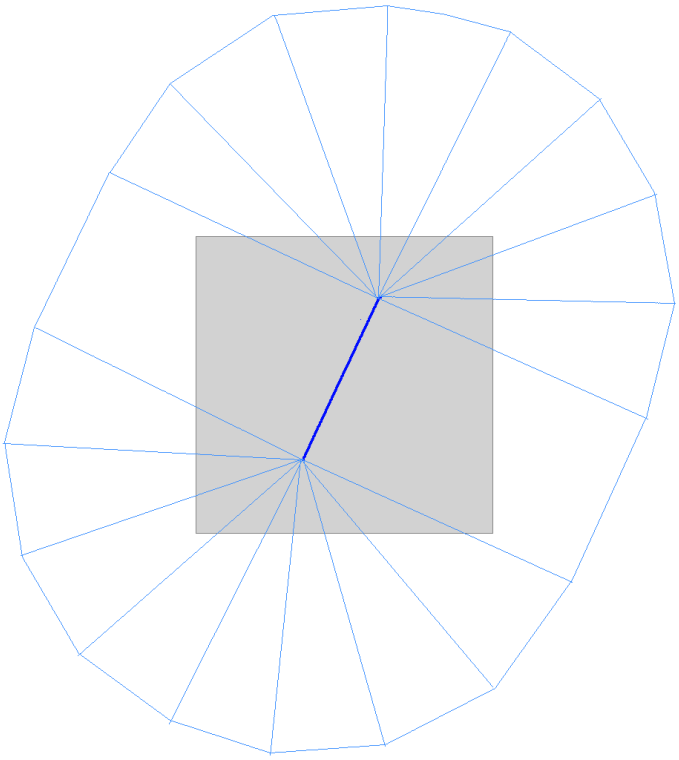


Figure 3.3: This picture shows how the distance mesh of a line segment primitive is generated. The distance mesh has to cover the whole texture, as the line segment primitive influences all points of the output texture.

The distance mesh of a line segment is generated very easily as it is composed of two quadrilaterals, one for each side of the line. This mesh, that represents the distance function as well as the region of influence of the line segment, has to be expanded so that the region of influence possibly covers the whole output texture (Figure 3.3). The distance function works as follows: pixels that lie directly on the line segment have the distance 0, the distance values of the pixels

increase as the pixel's distance to the line segment increases. The distance of the mesh has to be calculated only at the vertices of the mesh since the distance function is linear and therefore the distance of the pixels inside the mesh is interpolated by the shader. Since the mesh and the distance function for a line segment are exact, the distance error of this part of the diffusion curve contains only the error that has developed from the tessellation and rasterization. Figure 3.4a shows the distance function of a line segment.

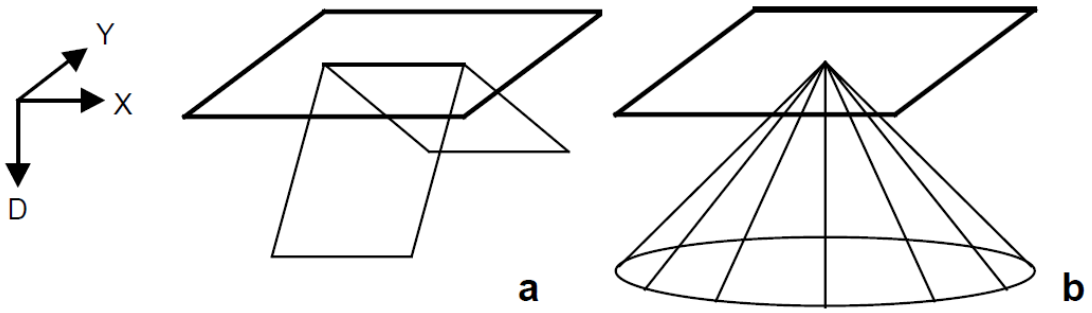


Figure 3.4: The distance functions of a line segment (a) and a point segment (b) of the 2D Voronoi diagram generation algorithm.

The other type of segment is the point segment. The distance mesh generation differs, if the point is an endpoint of the curve, or if it is a junction point between two line segments. Generally, the shape of the distance function of a point is a cone (Figure 3.4b), which is mapped onto the 2D plane. The resulting distance mesh is a sphere, the center of it consists of the point segment with distance 0, the distance values increase as the distance of the pixel to the point segment increases. However, for the Voronoi generation, the whole distance mesh of the point is not needed. In the case of an endpoint of a curve only half of the distance mesh is needed, as the other half is already covered by the distance mesh of the line segment that is connected to the point. In the case of a junction point even less is needed. Here, the size of the distance mesh depends on the angle between the two line segments that are connected to this point. Nevertheless, as in the line segment case, the distance mesh has to be extended to cover the whole output texture. Figure 3.2 shows a Voronoi diagram with the corresponding distance mesh.

If all the distance meshes of the segments of all curves are rendered, the Voronoi diagram is generated by z-buffer comparison. Along with this Voronoi diagram texture, a distance texture is generated, containing the distance of each pixel to its nearest segment. The resulting Voronoi diagram texture along with the distance texture serves as input for the diffusion algorithm.

3.3 2D Diffusion

Figure 3.5: A diffusion image generated by the algorithm of Jeschke et al. [2]. The first image shows the Voronoi diagram of the diffusion curves as initial guess, the other image show the diffused image after 1, 2, 8 and 400 steps.

After this step, Jeschke et al. [2] solve the diffusion in iterations. In one iteration, for each pixel, the color of the pixel is calculated from the pixels that lie in the pixel's *region of influence*. The size of the region of influence is determined by the distance texture that was created at the Voronoi diagram generation step. This ensures that the region of influence can't overlap a diffusion curve and therefore a color from the other side of a curve cannot flow to the other side. The current pixel simply would be calculated from the average of all pixels in the region of influence, however, Jeschke et al. [2] use only four pixels from the outer border of the region of influence. These pixels all lie at the main axes of the current pixel, each the maximum possible distance away. Only using these four pixels has the advantage that the algorithm converges much faster with few iterations. Another nice advantage is that this algorithm has a much higher performance. With more and more iterations of this algorithm, the diffusion gets smoother, but already at 8 iterations the algorithm shows a smooth-shaded image.

Figure 3.5 shows an image generated by the diffusion algorithm of Jeschke et al. [2], with the Voronoi diagram as initial guess and the diffused image after 1, 2, 8 and 400 steps.

3.4 2D Shape Extraction

From the resulting smooth spatial function we can extract the interpolated shapes by applying an isovalue to this function. The isovalue is used as a threshold, which, applied to the diffusion texture, results in an intermediate shape between the two input shapes.

Using the implementation of Jeschke et al. [2], which generated smooth-shaded diffusion images from a set of diffusion curves as input, I first extended it so that it can display and diffuse two sets of diffusion curves (Figure 3.6a). After this step, the colors of the first set of diffusion curves are set to black, the colors of the other one to white. Diffusing this combined set of curves results in a greyscale image (Figure 3.6b). By defining an isovalue, the isosurface can be retrieved. This isovalue is defined between 0 and 1, and represents a greyvalue. Applying this isovalue to the greyscale image renders all colors that are darker than the isovalue in black, all

lighter grey values as the isovalue are rendered white. This results in an isosurface image as in Figure 3.6c.

By varying the isovalue, shape interpolation can be achieved in a very simple way. If the defined isovalue is exactly 0 or 1, the image results in the original input surfaces. Any values between these two values are variations between the two input surfaces.

One of the advantages of this application is that all the operations can be done in real-time. The two surfaces can be moved or scaled at any time, the diffusion and the isosurface are updated and rendered in real-time.

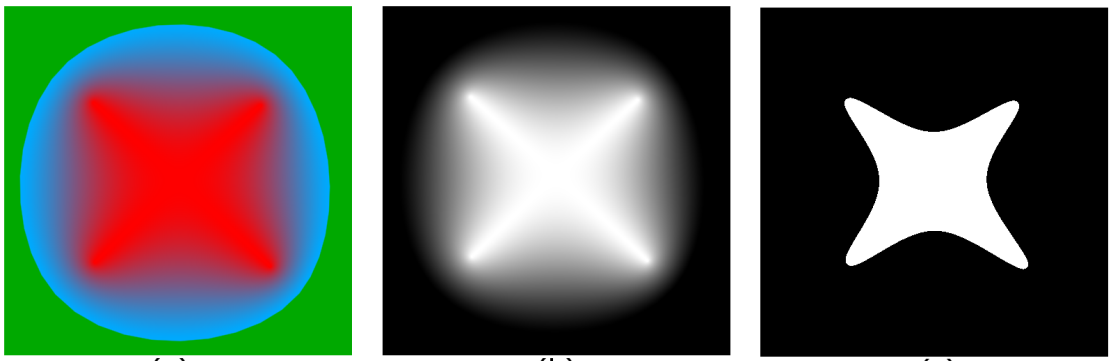


Figure 3.6: Two sets of diffusion curves diffused by the algorithm of Jeschke et al. [2]. The first set is the sphere, with two colors: green for the outer side of the set, blue for the inner side. The second set is a red cross that is placed inside the first set. (a) Shows the diffused image after 20 diffusion steps. (b) Shows the diffused image, when the colors of the two sets are set to black and white respectively. (c) Shows the image with an isovalue applied.

3.5 2D Results

For the algorithm to work, the one of the two surfaces needs to be placed inside the other. Figure 3.7 and 3.8 show results of the algorithm in this case. We can observe, that a change in the isovalue near 0 or 1 results in a higher change of the intermediate surface than a change in the isovalue between 0.25 and 0.75. We can see this behaviour in Figure 3.7. This is not as much of a problem, but Figure 3.8 shows a problem with much more importance. Due to the form of the thin blue area (Figure 3.8), the color of the other surface (in this case yellow) will diffuse to it only in an extremely high number of diffusion steps, which is not reasonable. Because of this (with a reasonable number of diffusion steps), the thin blue region will always have the same color. The result is that this region will not be included in the shape interpolation.

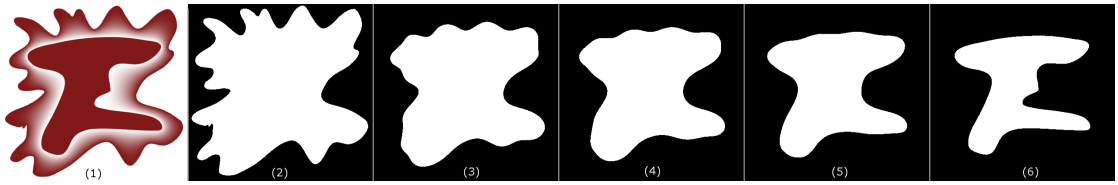


Figure 3.7: 2D shape interpolation of two surfaces with 40 diffusion steps. (1) shows the diffused surfaces, one is placed inside the other. The pictures show interpolated surfaces with isovalues of 0.001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

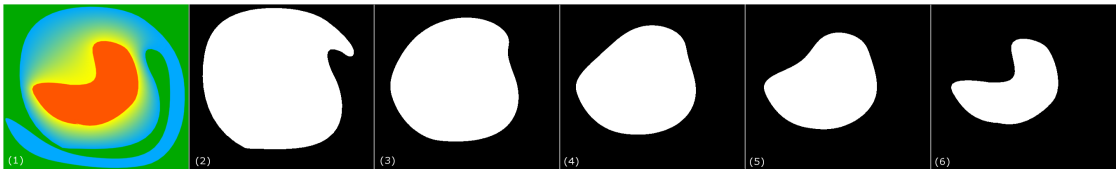


Figure 3.8: 2D shape interpolation of two surfaces with 40 diffusion steps. (1) shows the diffused surfaces, one is placed inside the other. The pictures show interpolated surfaces with isovalues of 0.001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

Figure 3.9 and 3.10 include other examples of the algorithm to show that the algorithm doesn't work well for shape interpolation if one of the surfaces isn't placed inside the other. In Figure 3.9 we can see that this case just leads to a weighted Voronoi diagram, Figure 3.10 shows no really useful result at all.

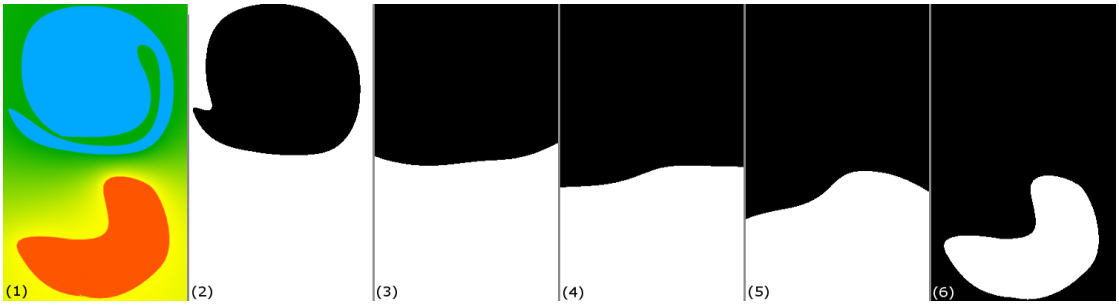


Figure 3.9: 2D shape interpolation of two surfaces with 40 diffusion steps. (1) shows the diffused surfaces, one is placed below the other. The pictures show interpolated surfaces with isovalues of 0.001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

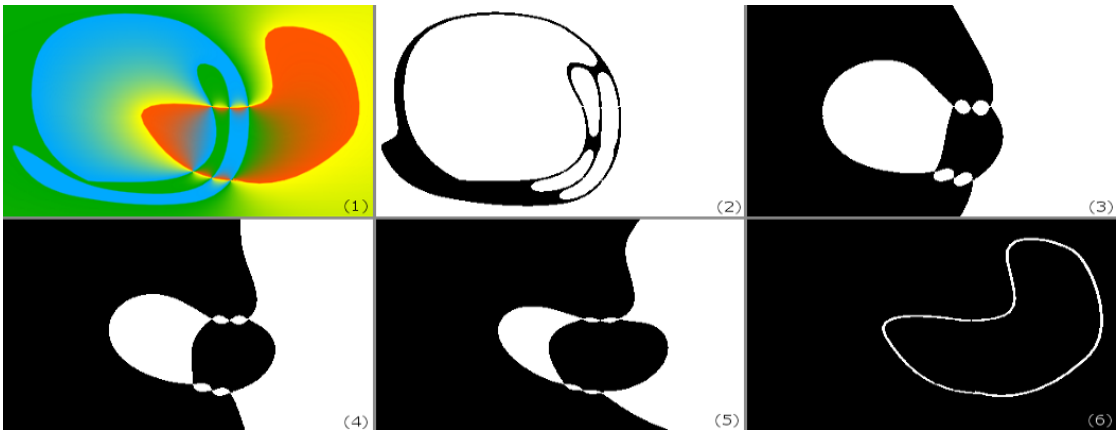


Figure 3.10: 2D shape interpolation of two surfaces with 40 diffusion steps. (1) shows the diffused surfaces, one is placed half inside and half outside the other. The pictures show interpolated surfaces with isovalues of 0.001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

3.6 2D Performance

In the 2D implementation, all rendering and calculation can be done in real time. My testing system consists of an Intel Core i5-2500 CPU with 3.30 GHz, 8 GB of RAM and a NVIDIA GeForce GTX 570.

I tested the frame rate of the diffusion of the two sets of diffusion curves in Figure 3.6 at a resolution of 800x800 pixels. The higher the number of diffusion steps get, the lower the frame rate drops. But even if the number of diffusion steps is set to a maximum of 400, the diffusion can be done in real time, however only with an average of 9-10 FPS. Table 3.1 shows the performance according to the number of diffusion steps.

In practice, more than 40 diffusion steps are not reasonable, since a higher number of steps don't significantly improve the smoothness of the image. Therefore the algorithm has a good performance with less than 40 steps. Depending on the number of diffusion curves, frame rates

diffusion steps	frames per second
0	670 fps
8	290 fps
40	85 fps
100	37 fps
200	19 fps
300	12 fps
400	9 fps

Table 3.1: Performance of the diffusion of the sets of diffusion curves from Figure 7, depending on the number of steps in the diffusion algorithm. Resolution is 800x800.

above 60 - 80 frames per second are easily achieved.

As in the 2D case only 2D textures are needed, the amount of video memory needed is rather low. The only factor for the amount of video memory needed is the window size, as the textures for the algorithm are in the same size as the window. It stays at a constant level if the size of the window stays the same size during runtime.

CHAPTER 4

3D Shape Interpolation

Using the 2D implementation as an overall guideline for the steps (Voronoi generation, diffusion, isosurface visualization), I began to extend the idea to the 3D case.
The idea in the 3D case is similar to the 2D case. The input for the 3D implementation consists of two textured 3D surfaces. Again, we need to define a smooth spatial function by diffusing the two input shapes in 3D space. By applying an isovalue to the function, we can obtain the interpolated surfaces.
Again, we first need to generate a Voronoi diagram. Then, the diffusion algorithm is applied to the resulting 3D texture of the Voronoi diagram generation, and as last step the isovalue is applied to the diffusion texture.
Paragraphs 4.1-4.3 describe the algorithm in the 3D case in more detail, paragraphs 4.4 and 4.5 discuss the results.

4.1 3D Voronoi Diagram Generation

As in the 2D Voronoi diagram generation I began with the approach of Hoff et al. [1], but with a slight modification. Instead of computing the distance meshes of the primitives in the geometry shader, as Hoff et al. [1] propose, I only generate the meshes of the regions of influence of the primitives in the geometry shader and map them to the slices of the 3D texture, the distance then is calculated in the pixel shader. This leads to a decrease in performance, but with the advantage the resulting Voronoi diagram is exact, with no approximations as in the approach of Hoff et al. [1]. In their approach the distance meshes of some primitives are only piecewise linear approximations of the real distance function.

The Voronoi diagram in three dimensions is computed as a sequence of 2D slices. For each slice and each primitive of a surface a distance mesh of the primitive has to be computed and mapped to each slice. As in the 2D case, the distance meshes differ with the type of underlying primitive. In the 3D case three types of primitives have to be distinguished: polygonal primitives, line segment primitives and point primitives. Each of these tree primitives has a different region

of influence to a slice and a different kind of distance mesh. By combining the regions of influence of all three primitives, the whole slice is covered.

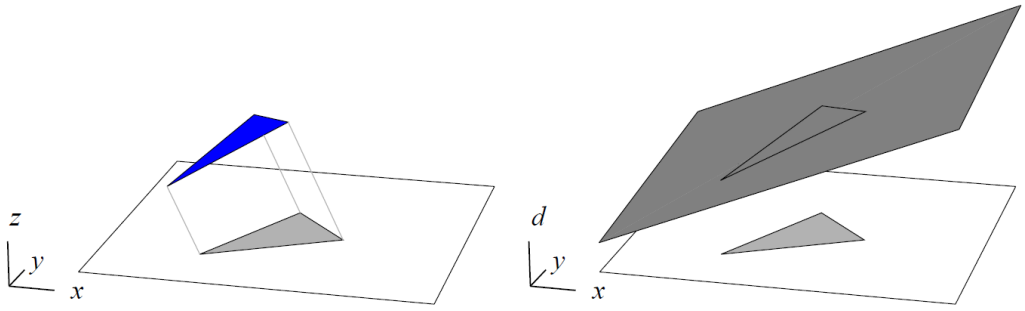


Figure 4.1: A polygonal primitive. The left image shows the region of influence of the polygon, the right image shows the linear distance function of the polygon with respect to the slice.

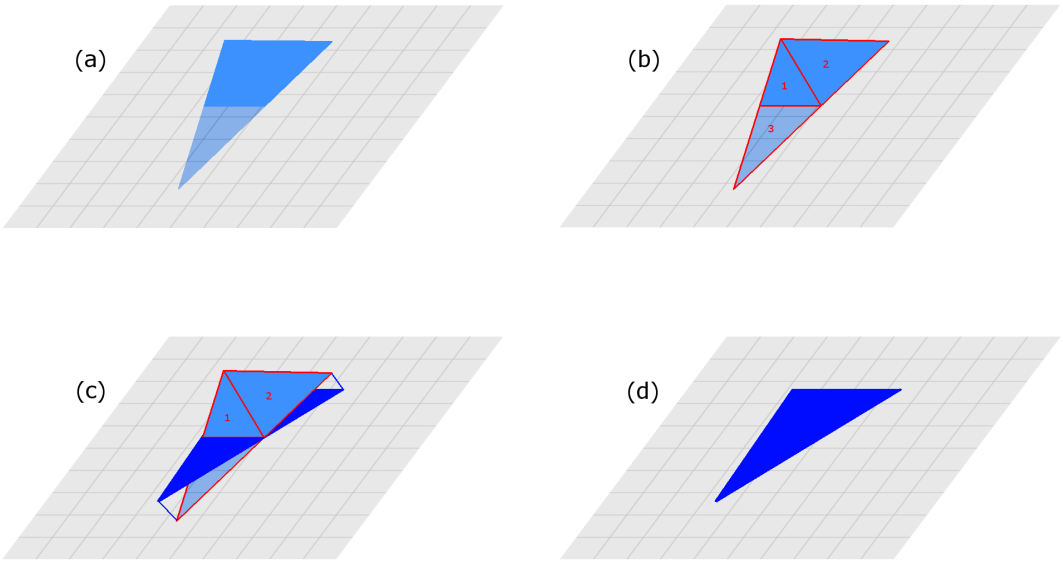


Figure 4.2: In this image a polygon (in this case a triangle) is divided by the slice. (a) Shows the triangle with one part above the slice, and one part below it. (b) Shows the subdivision of the triangle in three parts. (c) Shows the mapping of each of the newly created three triangles to the slice along their normal vectors. (d) Shows the mapped triangle on the slice.

At first, the meshes of the regions of influence of the polygonal primitives are mapped to each slice. In this case, the region of influence is the region formed by sweeping the polygon

orthogonally through space, which is shown in Figure 4.1 (left). This region is chosen so small, because all points outside this region are closer either to a line segment primitive or a point primitive. So for one slice, the polygon has only to be mapped to the slice along its normal vector. As I explained earlier, this mapping is done in the geometry shader. The distance between the region of influence on each slice and the polygonal primitive is then computed in the pixel shader. The distance function of this primitive is shown in Figure 4.1 (right). The only difficulty at the distance mesh generation of this primitive is when a slice divides the polygon in two parts. In this case, the triangle is divided into 3 new triangles, which then are projected to the slice along the normal vectors. This case is displayed in Figure 4.2.

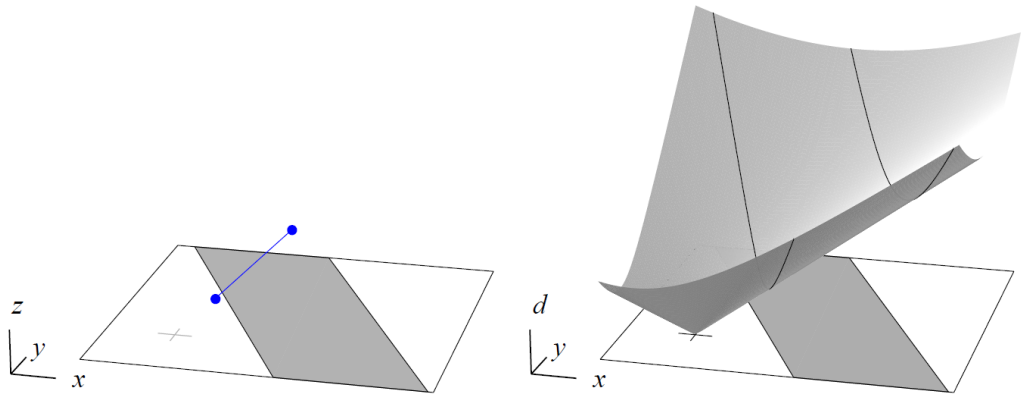


Figure 4.3: The region of influence of a line segment (left) and its distance function (right).

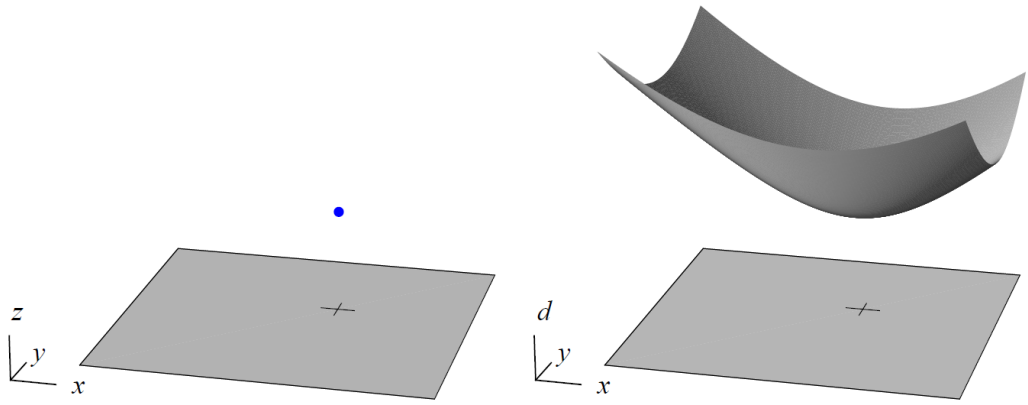


Figure 4.4: The region of influence of a point primitive (left) and its distance function (right).

The other two primitives are more complicated. The distance function of a line segment primitive is an elliptical cone with the apex at the intersection of the line with the slice (Figure 4.3). The eccentricity is determined by the angle of the line and the slice. But, as in the polygonal primitive case, there is the possibility to reduce the region of influence of the line

segment primitive in the slice. This region of influence lies between two parallel planes through the endpoints of the line, any point outside this region is closer to one of the endpoints of the primitive, which is then covered when we come to calculating the distance meshes of the point primitives. This leads to the projection as in Figure 4.3. I first project the line segment primitive to the slice along its normal vector that is pointing towards the slice. From these newly generated two endpoints I calculate the 4 points for a quad that covers the region of influence of the line segment primitive. These points are located far enough from the two projected points, so that the quad covers the whole region of influence to the slice. As always, the distance from the line segment to the region of influence on the slice is calculated in the pixel shader. In the case of a line segment, there also exists the case when a slice divides the primitive in two parts like in the polygonal primitive. In this case, the intersection point of the line segment and the slice has to be computed, and the meshes of the regions of influence of the newly created line segments have to be mapped to the slice.
The distance function and respectively the region of influence of the point primitive cover the whole slice. In this case, in the geometry shader, a quad has to be generated, that covers the whole slice. This is shown in Figure 4.4. After this step, the pixel shader computes the distance between each pixel of this mesh and the point primitive.
As in the 2D case, the z-buffer is used for distance comparison.

4.2 3D Diffusion

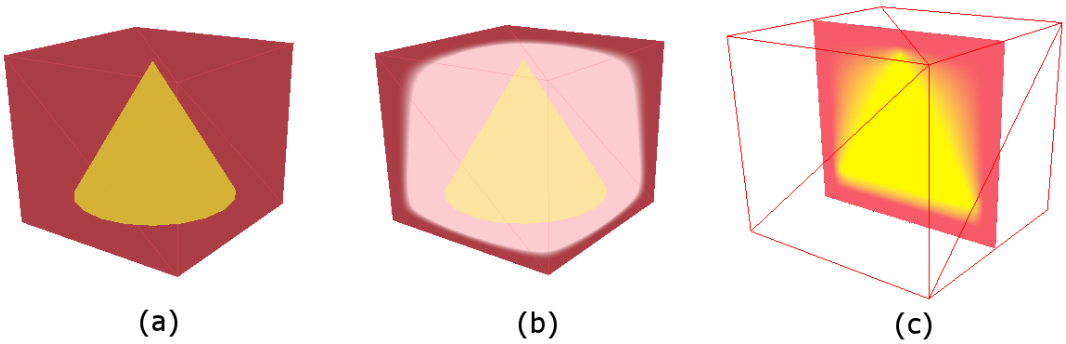


Figure 4.5: Example of a 3D diffusion with 8 steps. (a) Shows the two surfaces, a red cube with a yellow cone inside. (b) Shows the surfaces with the 3D diffusion texture. (c) Shows one slice of the 3D diffusion texture.

In the 3D case, the diffusion algorithm of 2D is simply extended by one dimension. In the 2D diffusion algorithm, for each pixel in the image, a point was sampled for each main direction. This resulted in 4 points that influenced the current pixel. The distance from these points to the pixel to be calculated was retrieved from the distance texture, which is the distance of the closest diffusion curve to the pixel. This ensured that colors at the other side of the diffusion curve could not flow to the other side.

Now, in the 3D case, the same approach is used. But as there are three dimensions, 6 points are now sampled. The influencing points now would lie inside a sphere with the distance of the distance map as radius. As in the 2D case, a sampling of only the points in the main axes leads to a fast convergence and an increase in performance. The color of the resulting pixel is simply calculated as the average of the colors of the pixels in the 6 main axes. Each iteration of this algorithm leads to a smoother diffusion of the colors, but as in the 2D case, this algorithm converges quickly and leads to a smooth representation after just a few steps.
Figure 4.5 shows an example of a 3D diffusion.

4.3 3D Shape Extraction

As in the 2D case, we defined a smooth spatial function by applying the diffusion algorithm to the Voronoi diagram. To extract the intermediate shapes, again we apply an isovalue to this function. The isovalue again is a threshold, which is applied to the 3D diffusion texture.
For the isosurface visualization of the 2D case, I set the color of one set of diffusion curves to white, the colors of the other set to black. Using this approach, the isovalue is just defined as a grey value. In the 3D case, I tried a different approach. In the Voronoi diagram I use the alpha value of the 3D textures for storing isosurface information. I set the alpha value of all pixels, which are nearer to the first surface to 1.0, the other to 0.0. In the diffusion algorithm, this alpha value is diffused in the same way as the color values of the texture, therefore all pixels that lie between the two surfaces get alpha values between 0.0 and 1.0. Because of this, the isosurface then can easily be retrieved from the color diffusion texture. In the user interface of my application, I added an additional option to switch between the color representation of the isosurface (the isosurface gets the colors of the diffusion texture) and a black/white representation of the isosurface.
Figure 4.6 shows colored isosurfaces between a red cube and a yellow cone with different isovalues. In this case, the closer the isovalue is to 0, the isosurface resembles more the cube and on the other hand it resembles more the cone if it is closer to 1.

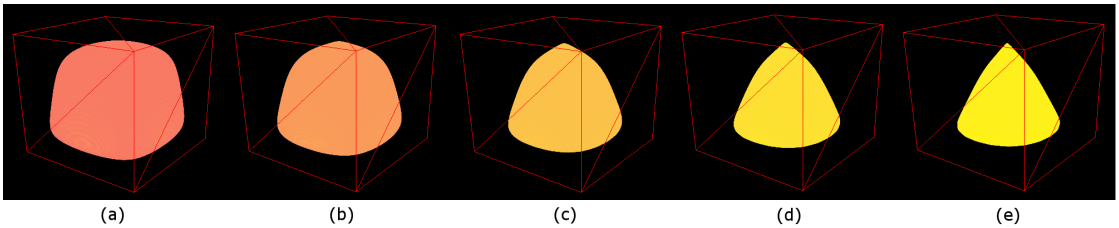


Figure 4.6: Colored isosurfaces of the example of Figure 4.5, with isovalue (a) 0.1, (b) 0.25, (c) 0.5, (d) 0.75 and (e) 0.9

4.4 3D Results

The algorithm works best if one of the two input meshes is placed inside the other. Examples can be found in the teaser image and in Figures 4.7 and 4.8. As we can see in the examples, the changes in the intermediate surfaces are a bit higher in the isosurface values near 0 and 1 as in the values between 0.25 and 0.75. As it should be, the intermediate surface with the isovalue 1 is almost the same as the input surface (with a slight difference due to the representation as volume). However, the intermediate surface does not behave like that at an isosurface value near 0, especially if the object has thin structures far from the other object. Examples can be found in Figure 4.7 (the ears of the bunny) and Figure 4.8 (the nozzle of the teapot).

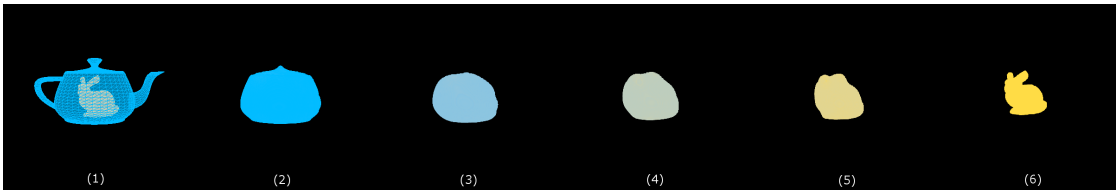


Figure 4.7: Shape interpolation between the bunny and the teapot mesh with 8 diffusion steps. The bunny is placed inside the teapot (1). The pictures show the isosurface with isovalues 0.0001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

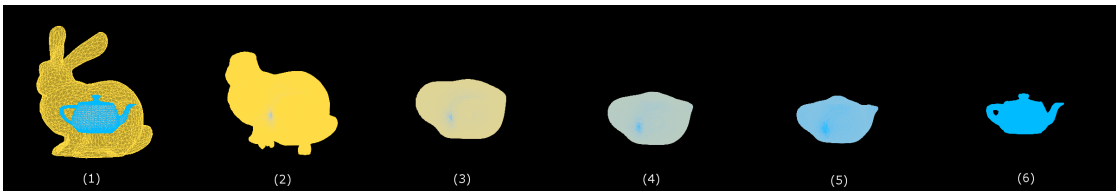


Figure 4.8: Shape interpolation between the bunny and the teapot mesh with 8 diffusion steps. The teapot is placed inside the bunny (1). The pictures show the isosurface with isovalues 0.0001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

If the objects aren't placed inside each other, but aside of the other (Figure 4.9), the algorithm behaves a bit different. The intermediate surface at the isovalue 1 looks like in the previous examples (Figure 4.7 and 4.8), however, at isovalue 0 the intermediate surface is inverted. The reason of this is that in my application I assumed that one of the object lies inside the other. If isovalues between 0 and 1 are applied, the intermediate surfaces look like a weighted Voronoi diagram.

For the algorithm to work as expected, one object has to be placed inside the other and the borders of the objects mustn't overlap. The result of an overlap between two input surfaces is shown in Figure 4.10.

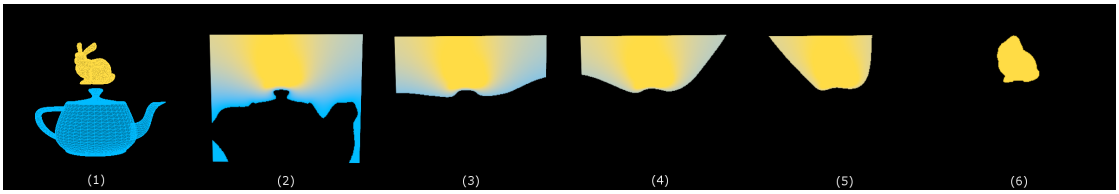


Figure 4.9: Shape interpolation between the bunny and the teapot mesh with 8 diffusion steps. The bunny is positioned above the teapot (1). The pictures show a slice of the isosurface volume with isovalues 0.0001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

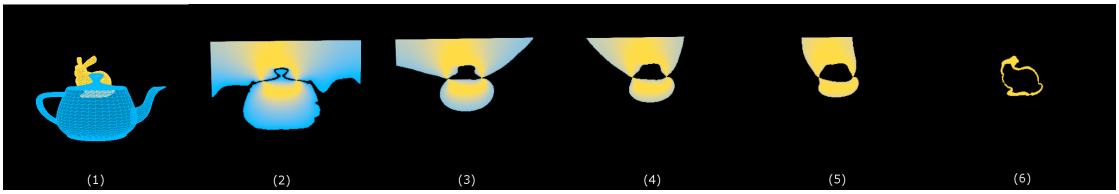


Figure 4.10: Shape interpolation between the bunny and the teapot mesh with 8 diffusion steps. The bunny is positioned half outside the teapot and half inside it (1). The pictures show a slice of the isosurface volume with isovalues 0.0001 (2), 0.25 (3), 0.5 (4), 0.75 (5) and 1.0 (6).

4.5 3D Performance

In contrast to the 2D application, all the rendering in the 3D application can't be done in real time. Due to the fact that I calculate an exact Voronoi diagram in the way as described before, the rendering of the Voronoi diagram takes some time to complete. The diffusion is also more expensive in 3D. Therefore, the rendering of the Voronoi and diffusion texture is only done if the user presses the Diffuse!-button. The volume rendering of course is then real-time. I tested the performance with the surfaces from Figure 4.6. Table 4.1 shows the performance of the Voronoi diagram generation with different texture sizes, Table 4.2 shows the performance of the diffusion algorithm with a texture size of 256 and a different number of diffusion steps.

texture size	Voronoi generation
128	less than 1 sec
256	1.5 sec
512	10.5 sec
1024	1 min 20 sec

Table 4.1: Performance of the voronoi diagram generation of the surfaces in Figure 17.

As seen in Table 4.1, the rendering time of the Voronoi diagram rapidly increases with the texture size. As we calculate the distance between a pixel and its corresponding primitive in the pixel shader, the size of a texture and therefore the size of a slice directly contributes to the increase of rendering time. The more pixels a slice has, the more distances between the

diffusion steps	diffusion generation
4	0.3 sec
8	0.6 sec
16	1.3 sec

Table 4.2: Performance of the diffusion algorithm with a texture size of 256 and different diffusion steps.

pixels and primitives have to be calculated. And this has to be done for every slice and every primitive. Additionally as the texture size grows, also the number of slices is increased, and this also contributes to the high performance loss at high texture sizes.
The rendering time of the diffusion also depends on the texture size, but not as much as the Voronoi diagram generation. The performance of the diffusion algorithm also depends on the number of diffusion steps as seen in Table 4.2. As the number of diffusion steps increases, the rendering time increases linearly.

The rendering time of the Voronoi diagram is not only depending on the output texture size, but also on the number of vertices of the surface meshes. To show this, I compared the example of Figure 4.6 and the example as of Figure 1.1. Table 4 shows the results of this comparison.

texture size	Figure 4.6	Figure 1.1
128	less than 1 sec	17 sec
256	1.5 sec	2 min 41 sec

Table 4.3: Performance of the Voronoi diagram generation of the surfaces in Figure 4.6 in comparison with the surfaces used in Figure 1.1.

We don't even need to compare the rendering times of the Voronoi diagrams of texture sizes higher than 256 to see that the rendering of the Voronoi diagram also greatly depends on the number of vertices of the two input surface meshes. Additionally we can see that an increase in the texture size increases the rendering time even more when there are a high number of vertices. However, the diffusion algorithm is only dependent on the texture size, an increase of the number of vertices has no effect on the diffusion algorithm.

Due to the number of 3D textures needed in the algorithm, the amount of video memory consumption is very high. The amount of video memory needed increases rapidly with an increase in the texture size. If rather small texture sizes are used (about 128), the algorithm does not require much memory. For example, with a texture size of 128, about 150 MB of video memory is used when the algorithm is applied to the surfaces in Figure 1.1. But as the texture size increases, the amount of video memory needed increases heavily. Already at a texture size of 256, about 800 MB of video memory is needed.

CHAPTER 5

Framework

5.1 2D Framework

The 2D framework is based on DirectX 10 with the DXUT10 API. For shader management, the Effects framework is used. In contrast to DirectX11, where the Effects framework was outsourced from the DirectX11 API, the Effects framework for DirectX 10 is included in its API.

Implementation Details

As basis for my implementation I started from the implementation of Jeschke et al. [2], which loads a set of diffusion curves and applies the diffusion. I extended this implementation so that two sets of diffusion curves are loaded at startup. To utilize as much from the existing implementation as possible, I simply combined these two sets of diffusion curves to one set, with the only difference that they can independently be moved and scaled.

Additionally, for the sake of isosurface generation, I added the possibility to set the color of one set to white and the other set to black. After applying the diffusion to this scene, the isosurface can easily be retrieved with a user-defined isovalue.

User interface

Figure 5.1 shows the user interface of the 2D framework. The diffusion of two sets of diffusion curves is displayed. It provides the possibility to move or scale the two sets independently, a slider to control the diffusion steps, and an option to display the two sets in black and white respectively. If the black and white option is enabled, the option to show the isosurface can be enabled. A slider controls the isovalue of the isosurface to be displayed.

5.2 3D Framework

For the 3D implementation I used DirectX 11 with the DXUT11 API along with HLSL as shader language. To this configuration I also added the Effects11 framework, which is also shipped with the DirectX 11 SDK. This framework simplifies the shader management, so that you only need to retrieve the technique and passes from the HLSL shader.
For model loading, I used the Open Asset Import Library (Assimp), texture loading is done with the FreeImage library.

Implementation Details

As a first step, two meshes are loaded, the framework supports textured meshes. After this step, the Voronoi diagram of these two meshes is generated as a 3D texture of a user defined size. The algorithm calculates a bounding box around the two meshes, and then renders the distance meshes of each primitive to the slices of the 3D Voronoi texture through an orthogonal projection. The mapping of the distance meshes to the slices is done in the geometry shader, because new geometry is generated. The distance is then calculated in the pixel shader. The Voronoi diagram generation is done in three passes, one for each kind of primitive of the mesh.
In the first pass, all triangles of the meshes are mapped to each slice of the texture along their surface normal. As the 3D texture size is not always uniform because of the proportion of the bounding box, the aspect of the 3D texture has also to be taken into account when mapping a triangle to a slice. Another case that has to be taken into account is that a slice can divide a triangle in two parts. In my implementation I divide the triangle into three new triangles (Figure 4.2), each of those is then mapped to the slice. The distance of each pixel of the mapped distance mesh is then calculated in the pixel shader.
In the second pass, all edges of the meshes are mapped to each slice. This is not so easy, because the calculation of the normal that maps the edge to the slice is not as trivial as in the triangle pass. The normal plane at each point of the edge has to be found, which serves for creating the distance mesh of the edge. Again, the distance is then calculated in the pixel shader.
In the third pass, the distance meshes of the vertices of the two input meshes have to be generated. As a point primitive's region of influence covers the whole slice, the distance mesh is simply a quad with the slice's size. Also the distance calculation in the pixel shader is trivial, as for each pixel the distance to the original vertex has to be calculated.

Along with the generation of the 3D Voronoi texture, a distance 3D texture has to be created, which is needed by the diffusion algorithm. The distance 3D texture simply contains the distance of each pixel in the 3D Voronoi texture to the closest primitive.

The next step is the generation of the 3D diffusion texture. As in the 2D case, this is done in a user-defined number of diffusion steps. As starting point of the algorithm serve the two resulting 3D textures of the Voronoi generation. For the first diffusion step, for each pixel in the newly created 3D diffusion texture, a pixel for each of the six main directions is sampled from the Voronoi texture, each of these sampled pixels is displaced by the distance in the distance texture. For the next diffusion step, the resulting 3D texture is then used as starting point, the distance texture stays the same. This is repeated until the number of diffusion steps is reached.

In the 3D diffusion texture I also store the isosurface information in the alpha channel of the texture.

The framework also includes a volume renderer, which I will not describe in detail. It uses a simple ray casting algorithm. The user now has the option to visualize the 3D diffusion texture or only one specified slice of the texture. He can also choose to visualize the isosurface, which can easily be generated, because of the existing information in the alpha channel of the 3D diffusion texture. The isosurface is also represented as a 3D texture, it has either a white color, or (if the user wants to display a colored isosurface) the color of the corresponding pixels in the diffusion texture. The user then has also the option to display the whole isosurface texture, or only one slice of it.

The user has also the possibility to save the currently displayed volume to a .dds file.

User interface

The user interface of the 3D implementation is shown in Figure 5.2. The user has the possibility to load two surfaces, between which the shape interpolation is applied. Methods for rotating, scaling and moving the surfaces are provided, along with a method to rotate the camera. A slider is used to change the maximum texture resolution, the actual resolution of the texture size in x-, y- and z-dimension is obtained by the proportion of the bounding box. Three checkboxes can be toggled to hide or show the surfaces, the volume and the bounding box. The Diffuse!-button triggers the Voronoi and diffusion texture-generation. For the diffusion, a slider controls the number of diffusion-steps. In the volume renderer, it can be switched between the rendering of all slices of the current displayed 3D texture, or only one slice (which can be selected with a slider). Two checkboxes control the visibility of the isosurface. It can also be switched between black/white- and the color-representation of the isosurface. The isovalue is controlled by a slider. The Save Volume...-button is used to save the currently displayed 3D texture.

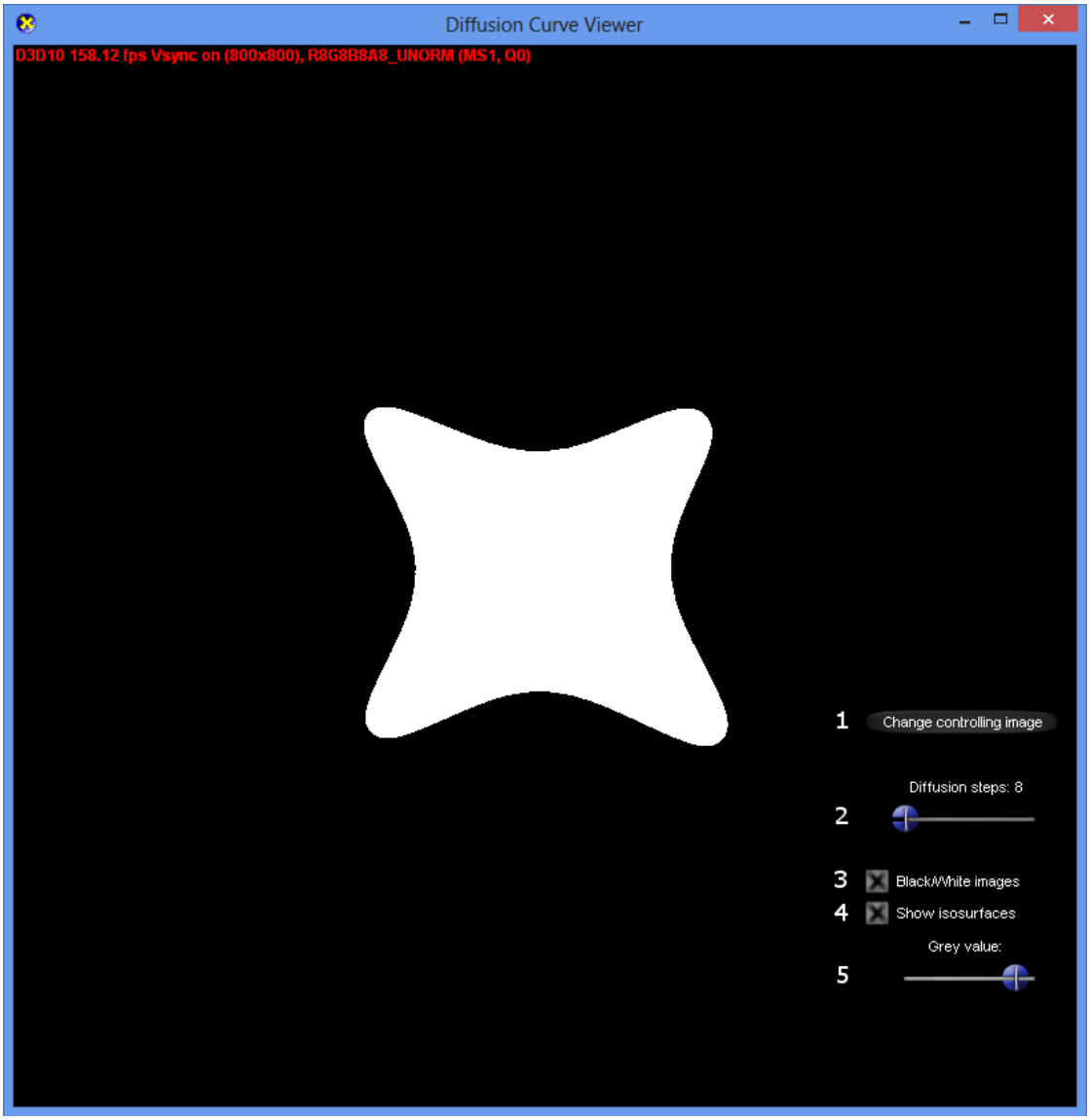


Figure 5.1: The user interface of the 2D framework. The first button (1) changes which set of diffusion curves the user can move or scale. The diffusion steps slider (2) controls the amount of diffusion steps for the diffusion algorithm. Its range is from 0 to 400 steps. The first check button (3) controls if the original colors of the diffusion curves are used, or if the colors are set to black respectively white. The second check button (4) controls if the isosurface is displayed or not. This option only appears if the first check button (3) is checked. The second slider (5) sets the grey value for isosurface retrieval. Its range goes from 0 to 1.

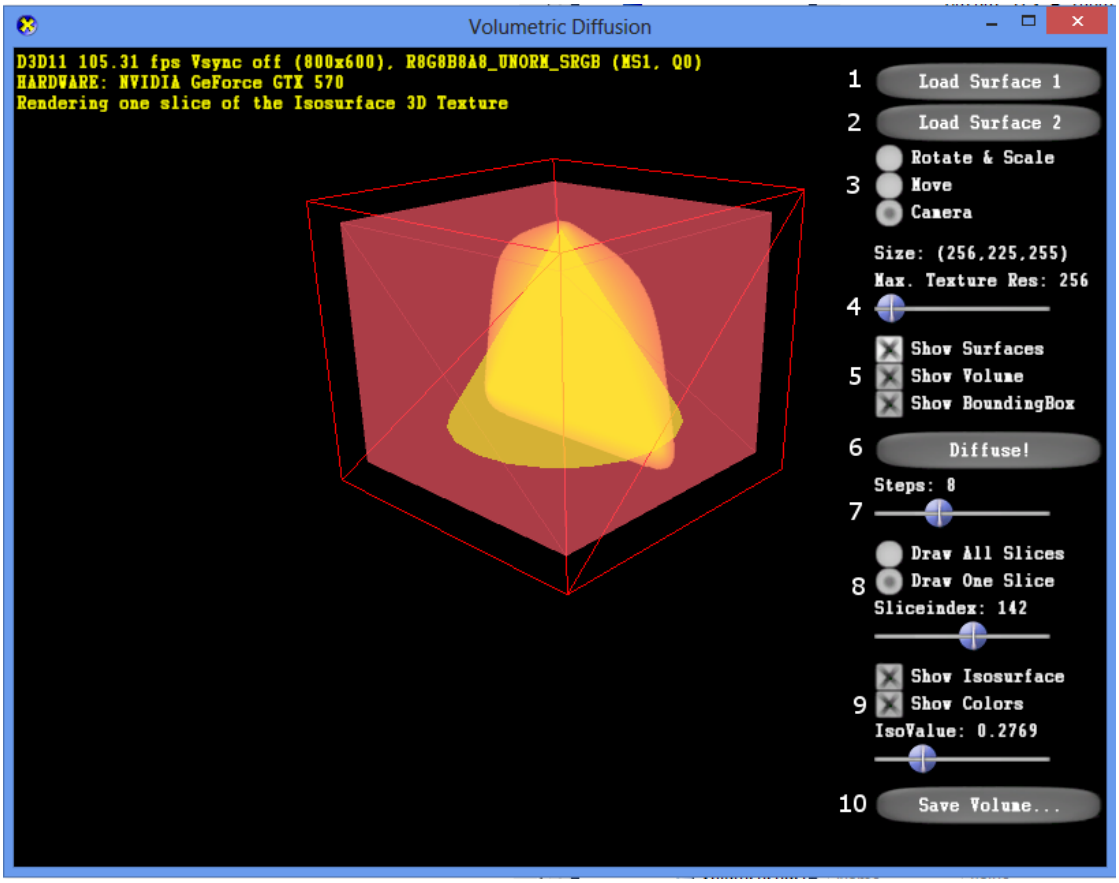


Figure 5.2: The user interface of the 3D application. With the first two buttons (1+2) the user can change both surfaces. The radio buttons below (3) are used to control the scene. The first surface is rotated, scaled and moved with the left mouse button, the second surface with the right mouse button. The first slider (4) is used to specify the resulting texture resolution. The check buttons in 5 control if the surfaces, the volume and the bounding box is displayed. By pressing the Diffuse!-button (6), the Voronoi diagram is generated and the diffusion algorithm is applied to the current scene. The slider in 7 controls the diffusion steps, the next set of controls (8) manage if the whole output texture is displayed or only one specified slice. The control set of 9 manage the rendering of the isosurface. With the last button (10) the currently displayed volume can be saved to the hard drive as .dds file.

CHAPTER 6

Conclusion

This bachelor thesis presented a shape interpolation method using diffusion isosurfaces. It is easily applicable to 2D and 3D, the isosurface of the diffusion texture can be retrieved in real time in both cases, therefore a smooth shape interpolation can be rendered in real-time. In 2D, even the Voronoi diagram generation and the diffusion algorithm can be done in real-time, in 3D the Voronoi and diffusion step is way more complex and expensive. As the Voronoi diagram and the diffusion texture only has to be renewed when a surface is displaced, it's not so important if those steps cannot be done in real time, as usually the shape interpolation is applied to two static surfaces that have a fix position in space.
All kinds of shapes that can be represented through diffusion curves or 3D surface meshes can be transformed with this shape interpolation method.

Bibliography

[1] K.E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999.

[2] S. Jeschke, D. Cline, and P. Wonka. A gpu laplacian solver for diffusion curves and poisson image editing. In *ACM Transactions on Graphics (TOG)*, volume 28, page 116. ACM, 2009.

[3] A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin. Diffusion curves: a vector representation for smooth-shaded images. In *ACM Transactions on Graphics (TOG)*, volume 27, page 92. ACM, 2008.

[4] G. Turk and J.F. O'Brien. Shape transformation using variational implicit functions. In *ACM SIGGRAPH 2005 Courses*, page 13. ACM, 2005.