Faculty of Mathematics, Physics
and Computer Science
COMENIUS UNIVERSITY, Bratislava

MASTER THESIS

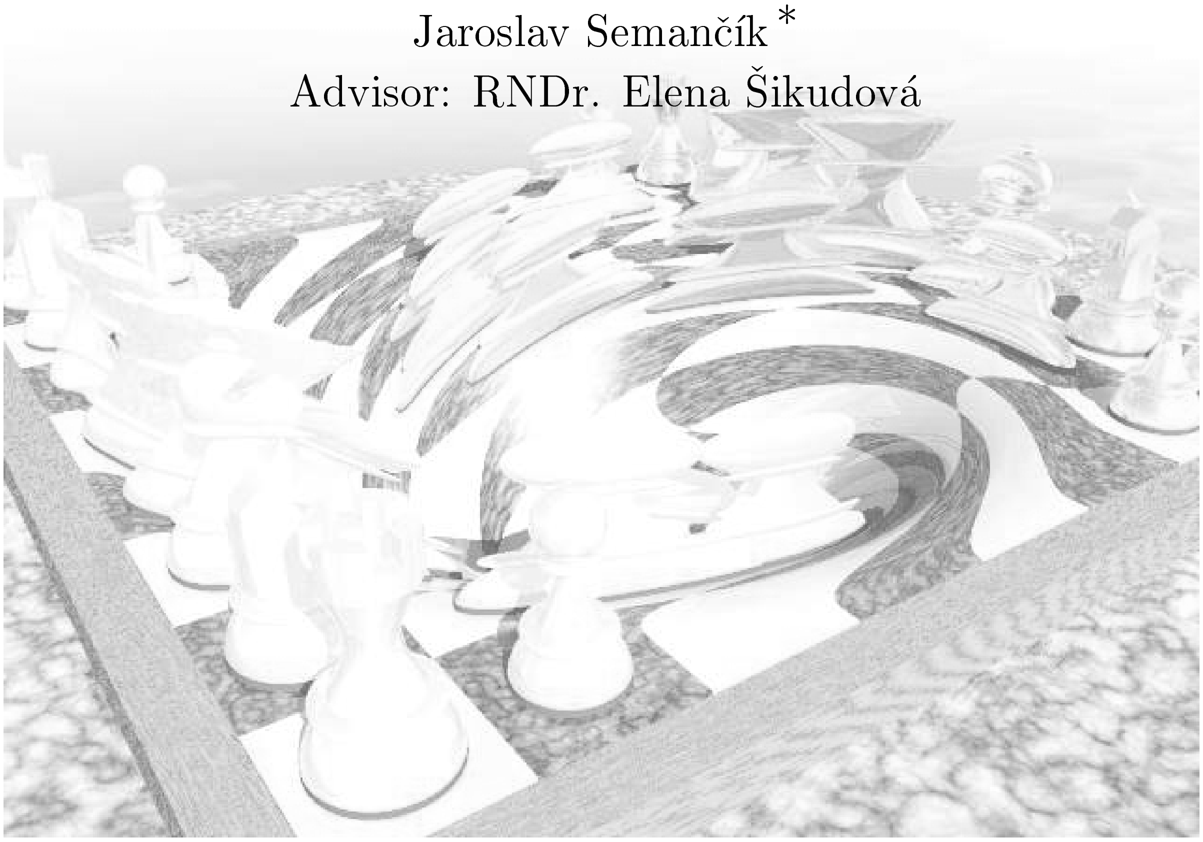
Ray Tracing Non-Linear Spacewarps

April 2001 Jaroslav Semančík

M A S T E R   T H E S I S

Ray Tracing Non-Linear Spacewarps

Jaroslav Semančík *
Advisor: RNDr. Elena Šikudová



Department of Computer Graphics and Image Processing
Faculty of Mathematics, Physics and Computer Science
Comenius University, Bratislava

* jaroslav_semancik@yahoo.com

Dedicated to my mother

Thanks to my advisor RNDr. Elena Šikudová for the patience and careful reading of the text.

Special thanks to Karol Datko and Matej Rehák for the hardware support.

Hereby I declare that this thesis was elaborated individually by myself only with use of the listed bibliography.

Ray Tracing Non-Linear Spacewarps

Abstract. The thesis deals with visualizing scenes in the warped space (or global deformations of the scenes) based on the ray tracing method. Rather than deforming objects in the scene directly, the inverse spacewarp is applied to the sight rays transforming them into curves. Rendering with the curved rays evokes then illusion of the warped space. Proposed method keeps all qualities of ray tracing as the most popular method for photorealistic rendering and presents an unified solution for deformation of solids in either surface, analytic and volume representations. Algorithms for tracing the curved rays are investigated and illustrated in an experimental implementation.

Keywords: spacewarp, deformation, visualization, ray tracing, curved ray

Zobrazovanie Nelineárnych Deformácií Priestoru

Abstrakt. Práca sa zaoberá zobrazovaním scén v deformovanom priestore (resp. globálnou deformáciou scény) metódou sledovania lúča. Namiesto deformovania objektov v scéne je však na sledované lúče aplikovaná inverzná deformácia, ktorá spôsobí ich zakrivenie. Sledovaním krivkových lúčov vzniká ilúzia deformovaného priestoru. Táto metóda si zachováva všetky kvality algoritmu sledovania lúča ako najpopulárnejšej metódy fotorealistického zobrazovania a predstavuje jednotné riešenie deformovania telies v povrchovej, analytickej i objemovej reprezentácii. Skúmajú sa algoritmy na sledovanie krivkových lúčov a sú ilustrované v experimentálnej implementácii metódy.

Kľúčové pojmy: deformácia priestoru, vizualizácia, sledovanie lúča, zakrivené lúče

Contents

Chapter 1
Introduction

Since the first origins of computer graphics men try the best to portray the real world. From the first wireframe models through simple local shading to present-day photorealistic rendering. Most of the fundamental problems of modeling and rendering three-dimensional scenes like hidden surface removal, realistic illumination or texture mapping has been satisfactory resolved through the years and an unskilled eye can hardly recognize if a picture has been taken from a real scene or generated by computer. Today's research on the field of rendering is highly specialized; even more sophisticated illumination models are being developed and simulation of various natural phenomena (like vegetation, ocean waves or fallen snow) is a permanent topic of the last world's top conferences on computer graphics. An immense amount of human endeavor is devoted to imitating the real world and the nature on computer screen. It is like a child trying to mimic the things he can see around. But when the child grows up, he finds its own way of self-expression. So what about letting the real world be real and create an original one(s) in infinite cyberspace? The worlds with their specific rules and laws, worlds never seen before. It's a challenge to human creativity, imagination and fantasy. Will they be nicer?

Some first signs of thinking this way have already appeared as *non-photorealistic rendering*, extending computer graphics by an artistic dimension. And this thesis is an another contribution, allowing an author to define its own shape of the space in a virtual world.

1.1 Motivation

In the beginning there was some considering about nonlinear perspective and unusual projections of space. The very first motivation for this thesis were M. C. Escher's[1] graphics. Well known Dutch artist–mathematician could be regarded as a forerunner of modern computer graphics. Long ago computer graphics era amazed his drawings by absolute realism and faithfulness. The litograph *Three spheres II* depicting three spheres made of glass, mirror and opaque material are really "photorealistically rendered", with proper reflections, refraction and caustics. There is reflected, working artist, the drawing on the table, whole room and even the trees beyond the window, on the mirror surface of the middle sphere. It really reminds an image demonstrating ray tracing capabilities.

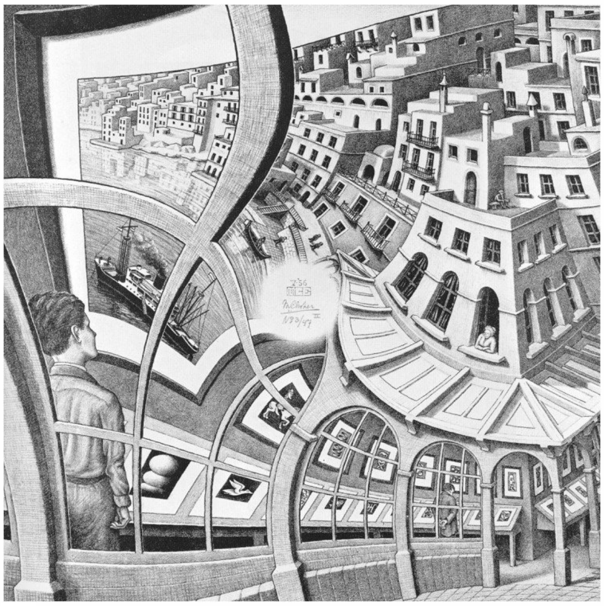


Figure 1.1. *Print Gallery*, M. C. Escher 1956, lithograph

[1] Maurits Cornelis Escher *1898 – †1972

Escher was, however, not only the master of his craft. Though mathematician amateur, with no higher education in mathematics, he has showed very good understanding of the spirit of mathematics and extraordinary ability to depict its beauty. The greatest part of his work illustrates the wide range of mathematical ideas and principles. Just the most fascinating pictures are the ones dedicated to geometry of space and its structure.[2] Besides the illustrations of various non-Euclidean spaces and "living" tesselations of the plane, there are variety of impossible constructions achieved by violating the logic of space. Searching three-dimensionality in such a two-dimensional picture, where some spatial relations among the objects are broken, confuses man's mind and results in visual paradoxes or illusions.

Maybe the most related Escher's picture to this thesis is the litograph *Print Gallery* (see figure 1.1). Here a young man in an art gallery is looking at a print of seaside town with a shop along the docks, and in the shop is an art gallery with a young man looking at a print of a seaside town... but wait! What's happened? Space is deformed here, turned somehow back into itself, so that the young man is both inside the picture and outside of it simultaneously. The secret of how the picture is constructed can be revealed by examining the grid-paper sketches, the artist made in preparation for the litograph, shown in figure 1.2. Note how the scale of the grids grows continuously in a clockwise direction and a *singularity* in the middle – the hole where this deformation of space is not continuous. To obscure this imperfection, Escher put his trademark initials smack over it.

Figure 1.2. Construction of the *Print Gallery* picture. Escher's preparatory sketches.

More about the art of M. C. Escher can be found on the WWW. Among the dozens of picture galleries especially remarkable is an essay [PR], where a mathematical background of Escher's work is considered in a very readable way.

Another inspiration to the thesis was *image warping*. Image warping is a computer graphics technique of intentional deforming or distorting an image. A transformation (usually non-linear) is applied to the picture and all pixels are mapped to a new position in resulting image.[3] Fairly interesting effects can be achieved, such as emphasizing particular features or illusion of deformed space in a picture (see figure 1.3). The illusion is, however,

[2] A branch of mathematics called *topology* studies this matter today.

[3] In fact, *inverse mapping* is used in practice. In this way an original to every pixel of resulting image is localized in original image. Using *forward mapping*, i.e. transforming all pixels to a new location in resulting image, "cracks" may occur in the places where no pixel is mapped onto.

not so convincing and in a careful look may fail. The reason is two-dimensionality of the image to be deformed (warped image is actually closer to a reflection in curved mirror as to a portrayal of space deformation). We never succeed in simulating such a deformation, which shows the backface parts of solids in the picture simply for the fact, that there is absolutely no information about appearance of the faces turned away from the viewer, contained in the image. All in all, we can achieve just some modification of the shape; spatial relations among the objects (e.g. mutual overlapping) and the visibility of their faces will never change. But what about doing something alike in 3D?

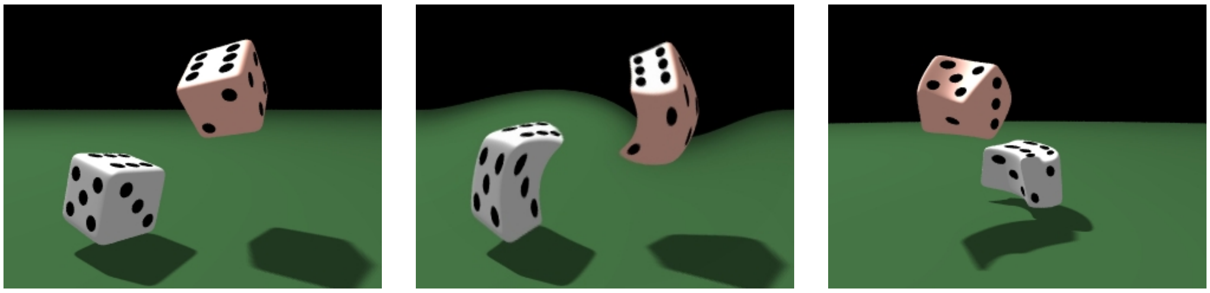


Figure 1.3. The difference between *image warp* and *space warp*. On the left, the original picture of two dice is shown (rendered in POV-Ray). In the middle, *image warp* (a whirl in this instance) was applied to the picture. Compare with the right where a *space warp* had taken effect and the scene deformed accordingly was rendered. Note new faces, turned away in the left picture, are visible now.

Finally, deciding was the following idea. Rendering a 3D-scene on the screen we may regard the screen as a "window" to a virtual world and for every pixel realize, what is visible "through" it. Simple and pretty intuitive way to achieve that is to shoot a ray from the eye through each pixel and when the ray hits an object in the scene, paint corresponding pixel by the color of the object hit first. It is very simplified principle of *ray tracing* – widely used method for photorealistic rendering, it will be explained thoroughly in chapter 2. But what happens if those rays will not be straight, but curved? The space of virtual world "beyond the screen" should appear warped or distorted some way. But is that true? And how it works? This thesis have seen light of the day to explore it.

1.2 Objectives

As mentioned before we will deal here with three-dimensional analogy of image warping; let's name it *space warping*. Given an arbitrary deformation[4] of three-dimensional space, our goal will be visualizing a scene in the deformed space. We focus on non-linear deformations.

Example of such deformation is a "black hole" attracting its surroundings, with influence decreasing with the distance. Thus the closer is an object to the black hole, the more warped is its shape towards the black hole center. But note that we are not talking about real black holes in the universe where the gravity forces nearby objects to move towards the center. Objects do not move here. By attracting is meant just the deformation of space, the difference from common Euclidean space. It is strong around the black hole center, while farther areas of space are almost unchanged.

[4] *Deformation* or *warp* will be exactly defined in section 3.1. Simply said, it is a transformation of the space (i.e. bijective mapping) satisfying a couple of requirements.

Rendering of the scene will be based on a modified ray tracing method – the scene itself remains intact, the sight rays are curved instead. And curved rays evoke an illusion of warped space. We are saying "illusion", but whether the space is really warped or not is just a matter of interpretation. Ray tracing hasn't been chosen on a whim. Besides the flexibility and remarkable qualities of that method in photorealism, an alternative and unconventional approach is introduced. And finally, as we will see later, it better corresponds to the term "spacewarp".

A part of the work is an experimental implementation of proposed method to confirm the theoretical conclusions. The implementation is supposed just to illustrate capabilities of the method, to deal with related problems and introduce a new ray tracing feature. We will not put a lot of effort to build a commercially successful, comfortable and robust application with plenty of packages, tutorial program and customer support. We insist, however, on an "open" implementation, where a spacewarp is arbitrary and user defined. The main goal of this thesis will be exploring and experimenting.

1.3 State of the Art

Deformations are not a new idea in computer graphics. Coming out from the needs of practice, deformations are widely used in modeling for quite along period of time. Gradually some basic stones – elementary and intuitive deformations were formed, such as *tappering* (scaling a cross-section along an axis by an amount dependent on the position on that axis), *twisting* (rotating a cross-section along an axis) and *bending* (like bending of a bar or sheet metal). Those deformations are mathematically well described (see [Barr84]) and often composed hierarchically to get more complex ones.

The more sophisticated deformation – *free-form deformation* represents a very flexible and powerful tool for solid modeling. A free-form deformation is determined by a three-dimensional grid of control points surrounding a solid to be deformed. A volume surrounded by the grid is deformed according to position of individual control points and spline interpolation between them. Designer moves the control points by hand until the desired result is achieved.

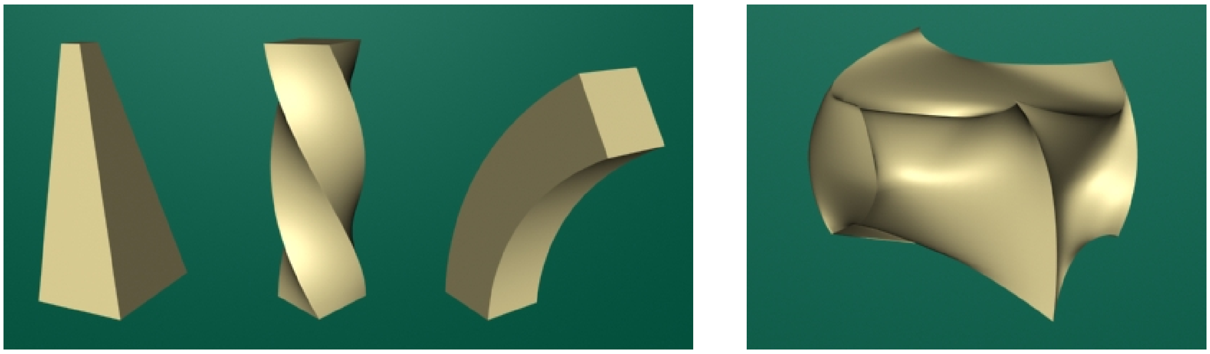


Figure 1.4. Three standard deformations – *taper*, *twist* and *bend* applied to the same bar are in the left image. The right image shows result of *free-form* deformation of a box.

The deformations mentioned above are usually used as local, i.e. they are assigned to the particular solid as an object modifier. However, global usage in larger volume of space is principally not excluded.

Even the term "spacewarp" is not unknown in the world of computer graphics. For instance, spacewarp can be found in popular modelling&rendering program 3D Studio MAX, as a global modifier. An artist may choose from built-in types of spacewarps and adjust its parameters (like strength, etc.) due to his heart. Next it is necessary to bind the spacewarp with all objects to be affected. Thus the spacewarp is not circumscribed to a particular volume of space, but to a set of individual objects regardless of they position. Using the terminology from Chapter 6 we would say, the spacewarp is *object-local*.

Common Approach vs. Inverse Approach

And what about implementation of the deformations outlined above? Common and straightforward method is *tesselation* (usually triangulation) of the solid and applying warping function to every vertex. Warped solid is then formed by the same tesselation on the new vertices. Just its geometry has changed, topology is the same. Modeling programs (like 3D Studio) often work with the uniform polygonal representation only. It means that all objects are approximated by a polygonal mesh. In this case just the transformation of all vertices is needed. The warped solid is regular polygonal mesh like anyone else and may be rendered using any method. This approach has a couple of virtues and several drawbacks.

The approach described in previous paragraph is characterized by uniformity and consistency. All the solids to be deformed are handled equally, and once a solid is warped, it is treated just in the same way as other solids. The scene may be rendered then very rapidly using any rendering method and even a dedicated hardware like graphics accelerator. Very strong power, especially in real-time animation, right? Moreover deformed solids may be transformed further, rotated, moved… But stop! A spacewarp is fixed in space and relative to world coordinates, hence a solid if moved, rotated, scaled or transformed in another way, as objects in animation usually do, its counterpart in warped space changes the shape as well, and so the solid must be warped each frame again. Situation is more kindly if the scene is still and only the viewpoint moves. It's quite a frequent case, for instance if looking for appropriate view to the scene. In that case it is sufficient to warp the scene just once.

On the other hand there are several drawbacks, all rooted in handling polygonal representation only and tesselation necessity. The simple geometric shapes like a sphere, box or a cylinder, all must be approximated by a set of faces (usually triangles). Tesselation is a kind of *sampling* (measuring a value in some discrete steps) in general – sampling of the solid's surface in this case. And like often in sampling , *aliasing* may arise, resulting in known "jagged" edges (see figure 1.5) or *artefacts* (undesired elements). To avoid aliasing, the methods of *antialiasing* are used, based on *subsampling*, i.e. sampling with higher density. Whole surface may be sampled denser (*uniform sampling*) or only the parts, where it's needed (*adaptive sampling*). To conclude and apply the theory above to our case, the most suitable is adaptive tesselation dependent on local curvature of the surface. The tesselation is sparse where the surface is nearly flat, while more detailed in the highly curved areas.

The most striking example of inefficiency of this approach is the case of volumetric or CSG representation (see next chapter), neither allowing to be deformed directly. The former used in medical applications, can be converted to a polygonal surface (e.g. by *Marching cubes* algorithm). The latter one heavily used in CAD/CAM and engineering, ray traced very elegantly and precisely, must be triangulated as well. In both cases a huge

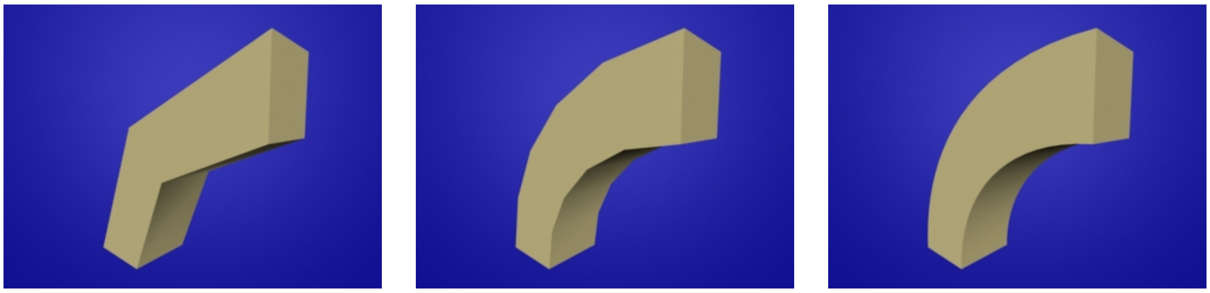


Figure 1.5. Bending a box in 3D Studio MAX with varying degree of tesselation. From left to right the box is vertically divided into 2, 5 and 20 segments respectively.

amount of primitives takes longer time to render and requires great amount of memory, especially for auxiliary data structures designated to speed up rendering, like bounding hierarchies or spatial subdivision trees.

In conclusion to described approach note that warping takes place in modeling stage in fact, *before* rendering. It is easy to implement - just apply the spacewarping function to coordinates of every individual vertex and render then. A goal of this thesis should be introducing original and unconventional idea – deforming viewing rays instead of the scene. To put some more effort to develop a method based on the basic ray tracing and realizing the spacewarp just in *render-time* without modifying the scene, to eliminate drawbacks treated above. As we will derive later, result of warping the scene and inverse warping of the viewing rays is the same. That idea is unfortunately not new. I was very disappointed discovering that the “inverse approach” had appeared already in several works.

Related Papers

The idea of inverse warping of the sight rays instead of warping the objects was the first time introduced in [Kaji83] and [Wijk84] applied to swept surfaces, and also mentioned in [Barr84] as a chance to render deformed solids. It was just four years after ray tracing was born. The outlined idea was later expanded in [Barr86], extending the set of “ray-traceable” surfaces to all the differentiable ones. A smooth surface is treated as a deformation of a flat sheet, the ray-surface intersection problem is converted to a new coordinate system in which the surface is flat and the rays are bent. An intersection searching is turned into finding a zero difference vector between a parametric surface point and a ray point for unknown surface parameters u, v and ray parameter t. Setting an initial difference vector in *difference* space, a differential equation is derived, which solution is a curve in *parameter* space joining a (u, v, t)-triple corresponding to the initial difference vector and (u_0, v_0, t_0)-triple corresponding to the zero difference vector, which is the intersection point. The initial value problem is solved, yielding the solution curve and the values of u_0, v_0 and t_0 determining the intersection position are found if they exist, i.e. if an intersection of the ray and surface exist. Starting values are best obtained from preprocessed surface divided into small portions with bounding box each. Intersecting a ray with that hierarchy of bounding boxes first, eliminates the rays completely missing the surface and gives a good estimate of starting values to the hitting ones. Computation is performed only to a needed degree of accuracy (a fraction of pixel).

In [Gröl95] the term “non-linear ray tracing” is introduced. Several sources of non-linearity – gravity centers, gravity lines and chaotic dynamical systems may force bending of the ray, or an explicit parametric curve for the ray may be given. Two representations of curved rays are considered – an iterative one and a hierarchical tree of curve portions. For the purpose of intersecting with object, the curve is adaptively approximated by a sequence of line segments (i.e. the length of the segment depends on local curvature of the curve). Algorithms handling two acceleration techniques – space subdivision and bounding volume hierarchies combined with the two curve representations are developed and implemented. However, only the primary rays are warped in that implementation; reflected and shadow rays are traced straight, which does not correspond to physical reality.

Ray tracing with curved rays is considered shortly in [Neyr95]. Rays are traced in a structured space (e.g. voxels or space subdivision grids), while focusing on the local illumination evaluation and its anisotropic case.

Finally, a series of papers [KuYa95, 96, 97] deals with deformations of surface and volumetric models in a very clever way. A new modeling operator – *deflector* is introduced. Deflectors acts like local deformations positioned in space and may be concatenated together, thus pushing the modelling process towards sculpting. There are several types of deflectors – translate, rotate, scale and discontinuous deflector, the last one designated to look into volumetric models producing cuts and cracks through the space. Deflector is actually a local transformation satisfying some requirements. Two different methods are proposed to render a scene affected by deflectors, both deforming just in render-time without modifying the whole model first. While ray tracing, sight rays passing through a deflector's area of influence are bent giving the impression of deformed surface. The rays are transformed in the direction opposite to that of desired visual effect. Due to the fact that the deflector is local, only a segment of the ray inside a deflector's area of influence is curved, yielding an effective implementation. Another way is "hardware-assisted" – employing a commercially available hardware mapping three-dimensional rasters (volumes) onto polygons. By rendering many parallel polygons, slicing the volume and perpendicular to the view direction, one generates a view of rectangular volume data set. Rendering these polygons from back to front and blending them into the frame buffer generates a correct image of the volume. Here the deformation takes place within a different mapping of the volume onto individual polygons. In this way, although at a price of lower quality, one can come close to interactive frame rate.

Our philosophy and primary intention in the following chapters will be "rendering spacewarps" in the sense of the word, i.e. rendering scenes in deformed, non-Euclidean space. Such spacewarps usually affects the scene as a whole, though local spacewarps and deformations of individual solids are not out of question and will be treated in later chapters.

Chapter 2
Fundamentals

In this chapter some mathematical basics and computer graphics methods are introduced, essential for understanding following parts. We explain briefly various representations of the objects used with ray tracing and the ray tracing algorithm itself. More thorough reading about those can be found in any better book about computer graphics principles, like [Fole95] or [Žára98].

2.1 Notation and Definitions

First of all we present typographic conventions and notation used henceforward in the text and we add also some fundamental definitions to avoid later misunderstandings.

The right-handed coordinate system is used (see figure 2.1). The set of real numbers is denoted by blackboard $\mathbb{R}$, and an n-dimensional Euclidean space as $\mathbb{R}^n$, particularly $\mathbb{R}^2$ stands for a plane and $\mathbb{R}^3$ for common three-dimensional space. A direct product $R_1 \times R_2 \times \ldots \times R_n$ of the sets $R_1, \ldots, R_n$ we shorten to $\prod_{i=1}^{n} R_i$.

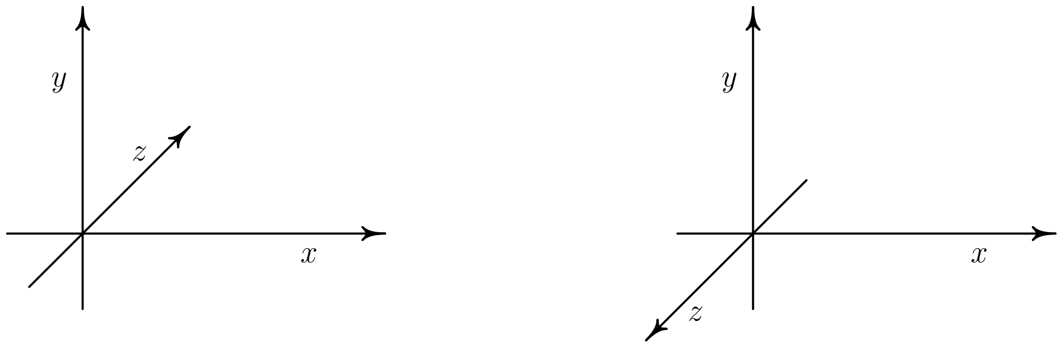


Figure 2.1. Left-handed coordinate system (left image) vs. right-handed coordinate system (right image). You can model the axes x, y, z of the left-handed coordinate system by the thumb, index finger and middle finger of your left hand respectively. For the right-handed coordinate system it is possible only with your right hand.

Both points and vectors are treated as n-tuplets and we will make no difference between them. An n-tuplet is interpreted as a point or vector due to the context. They are typeset boldface, e.g. $\mathbf{p}$, $\mathbf{v}$, their components are denoted by the same letter indexed from one and written in a *column-major form*, e.g.

$$\mathbf{p} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}, \qquad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}.$$

However, within a paragraph we write $\mathbf{p} = (p_1, p_2, p_3)^T$ in sake of saving the space. The *origin* point is denoted $\mathbf{0} = (0, 0, 0)^T$. The *length* of the vector $\mathbf{v}$ is signed $\|\mathbf{v}\|$ and defined through the dot product as

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}}, \qquad \mathbf{v} \in \mathbb{R}^n.$$

Rewritten to individual components it yields

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \ldots + v_n^2}, \quad \mathbf{v} \in \mathbb{R}^n, \quad \text{especially } \|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2} \quad \text{in } \mathbb{R}^3.$$

Boldface are also typeset functions returning a multicomponent vector, e.g. $\mathbf{f} : \mathbb{R} \to \mathbb{R}^3$, $\mathbf{w} : \mathbb{R}^3 \to \mathbb{R}^3$, on the other hand $g : \mathbb{R} \to \mathbb{R}$, $h : \mathbb{R}^3 \to \mathbb{R}$ are in standard mathematical italics. Moreover implicit functions are written capital, e.g. $F : \mathbb{R}^3 \to \mathbb{R}$, $F(\mathbf{p}) = 0$. Finally matrices are typeset as boldface capitals, e.g. $\mathbf{M}$, $\mathbf{Q}$.

Sometimes, especially with matrix forms of geometric transformations, it is beneficial to add one additional coordinate called *weigth*, getting *homogeneous coordinates* of a point or vector.

Definition 2.1. *Homogeneous coordinates* of a point (or vector) $\mathbf{p} = (p_1, p_2, p_3)^T$ is any $(x, y, z, w)^T$ that fulfills $p_1 = \frac{x}{w}$, $p_2 = \frac{y}{w}$, $p_3 = \frac{z}{w}$.

For instance $(4, 2, 6, 2)^T$ and $(6, 3, 9, 3)^T$ are both homogeneous coordinates of the point $(2, 1, 3)^T$. However, in practice is the weight component usually equal to 1.

Another specific type of coordinates are *barycentric coordinates*. In this case a point is expressed as a linear combination of several fixed points.

Definition 2.2. *Barycentric coordinates* of the point $\mathbf{p} \in \mathbb{R}^n$ with respect to given points $\mathbf{p}_1, \ldots, \mathbf{p}_k \in \mathbb{R}^n$ is a k-tuplet $(c_1, \ldots, c_k)$ of real numbers such that $\mathbf{p} = c_1\mathbf{p}_1 + \ldots + c_k\mathbf{p}_k$ and $c_1 + \ldots + c_k = 1$.

Barycentric combinations of two points forms a line, for three independent points it is a plane and in case of four independent points the whole three-dimensional space.

Definition 2.3. Let $\mathbf{f} : A \to B$. We say that the function $\mathbf{f}$ is *injective* (or *1–1*) if

$$\forall \mathbf{x}, \mathbf{y} \in A : \mathbf{x} \neq \mathbf{y} \Rightarrow \mathbf{f}(\mathbf{x}) \neq \mathbf{f}(\mathbf{y}).$$

We say that the function $\mathbf{f}$ is *surjective* (or *onto*) if

$$\forall \mathbf{y} \in B\, \exists \mathbf{x} \in A : \mathbf{y} = \mathbf{f}(\mathbf{x}).$$

When function is either injective and surjective we say it is *bijective.*

The consequence is that a bijective function unambiguously pairs members of the set A with members of the set B. Also an inverse function to a given function function exist if and only if it's a bijection.

Definition 2.4. Let $\mathbf{f} : A \to B$ and $\mathbf{p} \in A$. Function $\mathbf{f}$ is *continuous in the point* $\mathbf{p}$ if for arbitrary small $\varepsilon > 0$ there exist a $\delta > 0$ that

$$\forall \mathbf{q} \in A : \|\mathbf{q} - \mathbf{p}\| < \delta \Rightarrow \|\mathbf{f}(\mathbf{q}) - \mathbf{f}(\mathbf{p})\| < \epsilon.$$

We say that function $\mathbf{f}$ is *continuous* if it is continuous in every point $\mathbf{p} \in A$.

Translated freely, a function is continuous if a neighborhood of a point is mapped to a neighborhood of the point's image. Or in another way, getting closer to a point, the corresponding images are getting closer to the image of the point too.

A special class of functions are *linear mappings*.

Definition 2.5. A function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, is a *linear function* (or more often a *linear mapping*) if it fulfills

$$\forall \mathbf{p}_1, \ldots, \mathbf{p}_k \in \mathbb{R}^n,\ c_1, \ldots, c_k \in \mathbb{R} : \mathbf{f}(c_1\mathbf{p}_1 + \ldots + c_k\mathbf{p}_k) = c_1\mathbf{f}(\mathbf{p}_1) + \ldots + c_k\mathbf{f}(\mathbf{p}_k).$$

So linear mapping preserves linear combinations of points (and vectors). Any linear mapping $\mathbf{f}$ can be given by its matrix $\mathbf{M}$ as $\mathbf{f}(\mathbf{p}) = \mathbf{M}\mathbf{p}$. Composition of linear mappings $\mathbf{f}_1 \circ \ldots \circ \mathbf{f}_n$ with matrices $\mathbf{M}_1, \ldots, \mathbf{M}_n$ is a linear mapping given by the matrix $\mathbf{M}_n\mathbf{M}_{n-1} \ldots \mathbf{M}_1$. Inverse mapping to a linear mapping is a linear mapping again, given by the inverse matrix. Note that common geometric transformations like scaling, translation or rotation, their inverses and compositions are all linear mappings.

2.2 Representations of Geometric Objects

To work with geometric primitives like a line, ray, triangle or sphere and to evaluate spatial relations (e.g. intersection) among them, it will be necessary to express those object in an exact mathematical manner. It is usually accomplished by a formula or a system of formulas, occasionally it can be a procedure (e.g. fractal shapes) or some other mechanism. There are two useful ways with different philosophy to express an object analytically – *parametric* and *implicit*. We call them *representations*.

Parametric Representation

Example 2.1. Let's start by a simple illustrative example. A line in three-dimensional space is given by a point $\mathbf{p}$ to pass through and a direction vector $\mathbf{v}$ as shown in figure 2.2. Points of the line may be obtained by adding various multiples of direction vector $\mathbf{v}$ to the point $\mathbf{p}$. We express it

$$\mathbf{l}(t) = \mathbf{p} + t\mathbf{v}, \qquad t \in \mathbb{R}. \tag{2.1}$$

Note that $\mathbf{l}(t)$ is a point of the line corresponding to a t-multiple of $\mathbf{v}$ added to $\mathbf{p}$. Any point of the line is a t-multiple of $\mathbf{v}$ added to $\mathbf{p}$ for some parameter t, hence it can be expressed by equation (2.1). Moreover, no other point outside the line can arise from equation (2.1). Substituting only those $t \geq 0$, we get a ray in direction of $\mathbf{v}$. And $0 \leq t \leq 1$ results in a line segment with the length of $\mathbf{v}$.

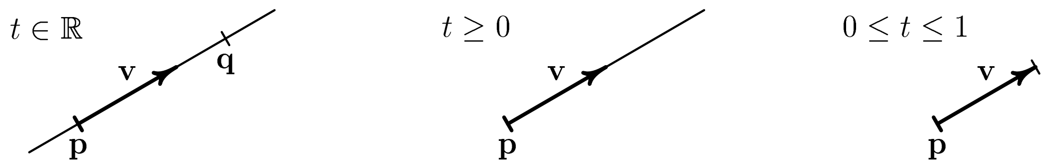


Figure 2.2. A line given by a point $\mathbf{p}$ and direction vector $\mathbf{v}$. Full line (left) is generated by parametric representation (2.1) for all $t \in \mathbb{R}$, a ray (middle) is generated for $t \geq 0$ and finally a line segment (right) for $0 \leq t \leq 1$. Note the point $\mathbf{q}$ in the left image as a result of equation (2.1) for $t = 1.5$, $\mathbf{q} = \mathbf{p} + 1.5\mathbf{v}$.

In general, any object in 3D space is formed by a set of points, it is actually a subset of all points from the space. The task of *parametric representation* is to generate *all* the points forming the object, but *no one else*. A formula with one or several parameters represents the object. Substituting parameters in the formula by a particular values results in a point of the object. Substituting parameters by all combinations of values from the domains given for each individual parameter, the set of corresponding points forms the whole object. Number of parameters implies "dimensionality" of resulting object. In previous example an one-dimensional line corresponds to the formula with one parameter. In a very formal way, parametric representation of d-dimensional object in 3D space may be written as

$$\mathbf{f}_d : \prod_{i=1}^{d} R_i \rightarrow \mathbb{R}^3 \tag{2.2}$$

where $d \in \{0, 1, 2, 3\}$ and R_i are domains (i.e. sets of permissible values) for individual parameters.[1] In a careful look one may notice, that the parametric representation is actually a function from parameter space $\prod_{i=1}^{d} R_i$ to object space $\mathbb{R}^3$, assigning to any parameter vector $\mathbf{t} \in \prod_{i=1}^{d} R_i$ a corresponding point $\mathbf{f}(\mathbf{t}) \in \mathbb{R}^3$ of the object.

Example 2.2. Perhaps the most trivial solid is a sphere; in spite of that (or just due to this fact) it is frequently used in scenes. Parametric representation of a sphere with a center point $\mathbf{c}$ and radius r can be written

$$\mathbf{s}(\phi, \theta) = \mathbf{c} + r \begin{pmatrix} \cos\phi \sin\theta \\ \sin\phi \sin\theta \\ \cos\theta \end{pmatrix}, \qquad \phi \in \langle 0; 2\pi \rangle,\ \theta \in \langle 0; \pi \rangle. \tag{2.3}$$

The angles ϕ and θ determines a "longitude" and a "latitude" respectively of a corresponding point on the sphere surface. Note that $\mathbf{s}$ is a bivariate function – an instance of the general form (2.2) of parametric representation for $d = 2$ and $R_1 = \langle 0; 2\pi \rangle$, $R_2 = \langle 0; \pi \rangle$. Why only two parameters whereas the sphere is three-dimensional? Formula (2.3) generates only the points on the surface of the sphere, which is two-dimensional. No point from sphere interior can be obtained. To get not hollow but solid sphere the third parameter is needed for varying radius

$$\mathbf{s}'(\phi, \theta, t) = \mathbf{c} + t \begin{pmatrix} \cos\phi \sin\theta \\ \sin\phi \sin\theta \\ \cos\theta \end{pmatrix}, \qquad \phi \in \langle 0; 2\pi \rangle,\ \theta \in \langle 0; \pi \rangle,\ t \in \langle 0; r \rangle. \tag{2.4}$$

Implicit representation

Another philosophy to parametric "generate all points" representation is a question "Which points form the object and which points do not?" Or in other words "Given a point in space, is this a point of the object or not?" *Implicit representation* is a formalization of the above question. It is a function

$$F : \mathbb{R}^3 \rightarrow \mathbb{R} \tag{2.5}$$

assigning to every point $\mathbf{p}$ of the space a scalar value $F(\mathbf{p})$. An object with implicit representation (2.5) is then a set of all those points $\mathbf{p}$ where $F(\mathbf{p}) = 0$.

[1] Note that in 3-dimensional space cannot exist more than 3-dimensional object, simply for the fact, that an object is a subset of the space. Also note, that 0-dimensional object is a point with no parameter in its parametric representation.

Example 2.3. Consider our old friend – the sphere with radius r and center point $\mathbf{c}$. The sphere surface is a set of the points equidistant from the center point at distance of r, i.e. set of all points $\mathbf{p}$ satisfying $\|\mathbf{p} - \mathbf{c}\| = r$. A function assigning the value 0 just to those points is straightforward

$$S(\mathbf{p}) = r - \|\mathbf{p} - \mathbf{c}\| \tag{2.6}$$

or rewritten to individual components

$$S\begin{pmatrix} x \\ y \\ z \end{pmatrix} = r^2 - (x - c_1)^2 - (y - c_2)^2 - (z - c_3)^2. \tag{2.7}$$

The sphere surface is then given by implicit formula $S(\mathbf{p}) = 0$. It is a hollow sphere like in previous case, with infinitesimally thin shell; an implicit formula for solid sphere can be written

$$S'(\mathbf{p}) = \begin{cases} 0, & \|\mathbf{p} - \mathbf{c}\| < r \\ r - \|\mathbf{p} - \mathbf{c}\|, & \text{else.} \end{cases} \tag{2.8}$$

Furthermore, note that an implicit function F, divides the space into three parts. The points $\mathbf{p}$ where $F(\mathbf{p}) > 0$, the points where $F(\mathbf{p}) = 0$ and finally those where $F(\mathbf{p}) < 0$. The second part is the object itself. In accordance with intuitive understanding (check the sphere surface above) we may define then an *interior* of the object as the part where $F(\mathbf{p}) > 0$ and an *exterior* as the part where $F(\mathbf{p}) < 0$.[2] In case of unbounded, open object like a plane or paraboloid, its interior is infinite and unbounded as well. Usually the object separates the interior part from the exterior part, but if implicit function is extremely wild (and it may be principally any function), the result can be a cloud of individual points or anything, where the terms "interior" and "exterior" loose the sense. In commonly used, reasonable objects, however, the interior and exterior parts corresponds to our intuitive understanding and even more may be deduced from the value of $F(\mathbf{p})$. The greater is an absolute value of $F(\mathbf{p})$, the farther is the point from the object. Moreover in our sphere surface example, the value of $|F(\mathbf{p})|$ is directly the distance to the surface, but it is not applicable in general.

Each of the two described analytic representations – *parametric* telling "*how* to generate points of the object" and *implicit* deciding "*whether* a point belongs to the object or not" has its own advantages and disadvantages over the other. Parametric representation is suitable for systematic generating of points on the object, e.g. in creating a polygonal model by tesselation. On the other hand for a given point it can say nothing about the point's position with respect to the object, e.g. if the point belongs to the object, is it inside, outside or how far from the surface. That is just the matter of implicit representation. Though not providing instructions for object generation (from that the term "implicit" – as "non-explicit", indirect), it answers the spatial relations above. Implicit solid may be rendered by ray tracing (see next section), deriving its intersection with a parametric ray. Conversion from parametric to implicit representation and vice versa is almost impossible task. For majority of commonly used geometric primitives both parametric and implicit expression have been derived independently (recall again our example

[2] In our example the solid sphere has no interior, since $S'(\mathbf{p}) \leq 0$ for every $\mathbf{p}$.

of a sphere). There are, however, ones with only the parametric expression known (e.g. Bezier patches) and ones with only the implicit expression, like blobs.[3]

Constructive Solid Geometry (CSG)

Artificial, human-made objects are usually characteristic by straight edges and regular structure, hence they can be well mathematically described. However, one global formula would be still too much complex. One may notice that human-made objects are built from a couple of primitive elements (most often boxes, cylinders) using operations like joining two components or cutting a part off. *Constructive solid geometry (CSG)* is based on building the object from geometric primitives, like box, sphere or cylinder by three set operations – *union*, *intersection* and *difference*. For instance a hole may be drilled into a box by subtracting a cylinder or a lens is obtained as an intersection of two eccentric spheres. A solid in CSG is represented by a tree with geometric primitives in its leaves and set operations and spatial transformations in inner nodes. The root of the tree corresponds to complete solid.

For CSG representation, reflecting a manufacturing process of the solid is the main application area engineering and CAD/CAM modeling. CSG solids are very precisely rendered by ray tracing (see in next section about ray tracing algorithm).

Volumetric Representation

For highly complex shapes, natural objects and especially data scanned from the real world an analytic formula is not at our disposal. The only way to store it is the *volumetric representation*. The idea is simple: Surround the object by a three-dimensional grid and for each node of the grid set if it is inside or outside the object. One simple way to visualize such model is to display a small cube at position of each inside node. The denser is the grid, the more faithful is the approximation. Much more clever is to store in each node not just in/out, but the distance to the surface or "local density" (or another value) of represented object, assuming the density outside the object as zero. Surface of the object is then reconstructed by interpolation between adjacent node values.

For reasonably faithful model the grid often requires hundreds of nodes on each axis, resulting in megabytes of data. To reduce the amount of data an *octree* structure has been designated. Starting with the whole box enclosing the object, divide it into eight ($2 \times 2 \times 2$) boxes with half-size edges. Determine which boxes are full – filled by the matter of the object, void – where no part of the object interferes, and mixed – containing both the matter and empty space. Set the type of each box, and all mixed boxes are recursively subdivided further, yielding a tree structure, until some maximum depth is reached. The tree represents the object much more effectively than a grid storing the uniform areas of the space by a single large node, while subdividing finer around the surface.

Volumetric representation is used primarily in medical applications where a scanned 3D data are common.[4] An object in volumetric representation can be converted to a triangular mesh, mostly by *Marching cubes* algorithm (see [Žára98]) or rendered directly by ray tracing.

[3] Blobs are smooth surfaces specially suitable for modeling organic shapes. Blob usually looks like several drops of water blended together. Given several points in space and corresponding intensity values, imagine those points as sources of magnetic field. Anywhere in the space the total intensity of the field is the sum of all component's contributions. The blob surface is a particular level of the field – all points where the intensity of the field is some fixed, given value. See figures 7.1 and 7.2 for blob examples.

[4] Maybe from that the most often example for volumetric representation is a human head or skull.

2.3 Ray Tracing

Ray tracing is the most popular method for generating realistic images by computer and since its introducing in 1980 a considerable amount of work has been devoted to enhance the basic algorithm. The result is a robust and versatile method of *photorealistic rendering* with hardly anything beyond its power. As the term says, images produced by ray tracing may be difficult to distinguish from photographs of the real world. Since ray tracing will be an essential algorithm in rendering spacewarps, we explain the basics.

All what we see is provided by the light falling into our eye. A little bit simplified, the light rays are emitted from a light source, they are absorbed and reflected by the surfaces, even transmitted through the translucent solids, until the lucky ones ends up in someone's eye and their less lucky friends are reflected away to the deep space. Besides the fact, that only the small part of those rays contributes to the final image, tracing the infinite number of all those rays would be impossible anyway. We are able, however, to deal with only the rays ending up in observer's eye by tracing the ray *backward*. Given a *viewpoint* (a virtual camera position) in space, a direction and a view angle, the window can be derived the observer is looking through and showed on the screen. To get an image corresponding to observer's view for each pixel of the screen answer the question "What is the light information carried by the ray incoming through the pixel?" and set it to the respective color. To answer that, shoot a ray through the pixel from the viewpoint and trace it backward, reflect by surfaces and transmit through translucent solids, just as the light do, but in the opposite direction.

Basic Algorithm

The basic algorithm for ray tracing is presented below. For full understanding it is crucial to realize that the real light doesn't consist of isolated rays being reflected. A single ray hitting a surface is distributed to a number of rays leaving the surface in all directions according to optical properties of the surface. And likewise, a single ray leaving the surface is composed by contributions of the rays incoming from all directions, reflected in direction of the leaving ray. Those laws are described in general by the global illumination model. The ray tracing algorithm considers only the most important contributions – from direct light sources and the two rays following the laws of reflection and refraction. The following procedure is called with depth $D = 0$ for each ray shot from the viewpoint.

TraceRay(ray R, recursion depth D)

1. Find the nearest intersection I of the ray R with objects in the scene
2. If there is no intersection, assign to ray R the background color and return
3. Shoot a ray from intersection point I to every light source and evaluate the light's contribution to illumination of I
4. If recursion depth D hasn't reached a given maximum depth yet, then
   a) create reflected ray R_R and *TraceRay*(R_R, $D + 1$)
   b) create refracted ray R_T and *TraceRay*(R_T, $D + 1$)
5. Sum R_R, R_T and all light's contributions and the resulting color assign to ray R

An example of tracing one particular ray is shown in the figure 2.2.

Ray tracing algorithm is very simple and elegant, nevertheless really powerful. The most important problems of realistic image synthesis are solved – hidden surface removal, shading and texturing, casting of shadows, reflections on glossy surfaces and

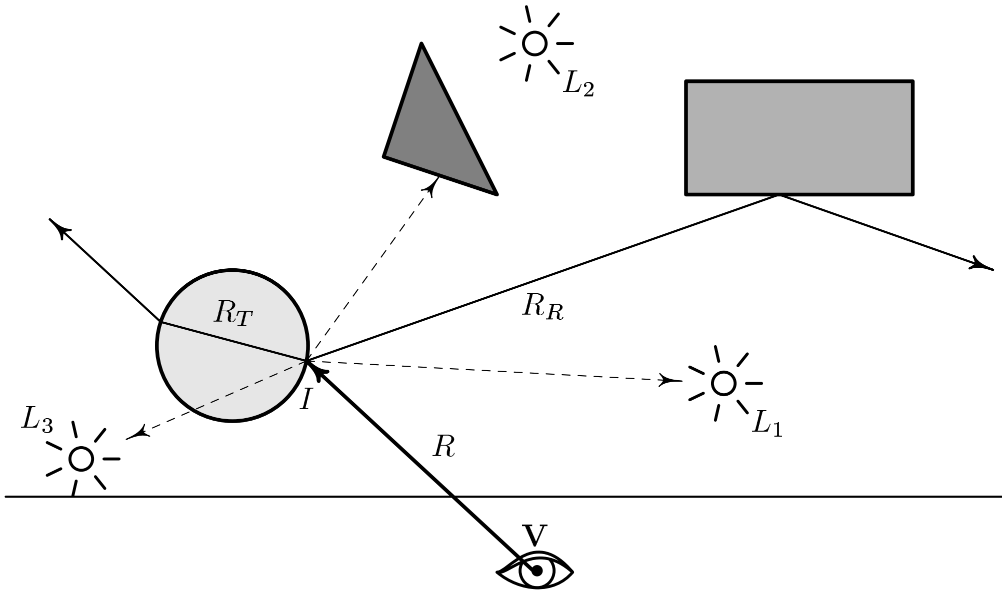


Figure 2.2. Tracing one particular ray. The ray shot from the viewpoint hit the sphere surface first at the point I. Three rays to the individual light sources are shot then and amount of incoming light is evaluated. The light from L_1 is in full intensity, the light from L_2 is blocked by an opaque solid, i.e. I lies in the solid's shadow, and finally the light from L_3 is attenuated by the translucent interior of the sphere. Moreover, a reflected ray R_R is traced further and since the sphere is translucent a refracted ray R_T as well.

even the light refraction at interface of different environments. On the other hand, ray tracing cannot handle indirect diffuse illumination.[5] Invaluable feature of ray tracing is handling a variety of representations of the solids, moreover all at once in the scene. The only requirement is to be able to evaluate intersection of the ray with represented object. For analytic representations the intersection is derived from formulas. In CSG shapes the intersections with all primitive components are evaluated first. With the intervals, where the ray is inside individual components, the same set operations are performed then as with the components during shape creation. Finally, the first point of the nearest resulted interval is the nearest intersection with CSG shape. In case of volumetric representation, the ray is traced through a 3D grid.

Speed-up Techniques

However, the price is paid for the quality. The time in this case. The algorithm as described above may consume hours to render an image of a more complex scene. Some methods have been developed, which speed up the basic algorithm by several orders, nevertheless the ray tracing is still far from being real-time. The greatest part of the time is spent by looking for intersections in case of the basic algorithm, since it tests each ray against each object. For that reason the major performance leap is achieved by more effective intersection testing. A *preprocessing* step is added before actual rendering, when auxiliary data structures are built holding some spatial relations in the scene. Utilizing these data structures, rendering is significantly faster, and overall performance (preprocessing + rendering time) benefit is achieved.

One way to more efficient ray-object intersections is introducing the bounding volumes. Bounding volume is a simple geometric primitive (sphere or axis-aligned bounding

[5] Indirect illumination is the incoming light reflected by the other surfaces.

box, abbr. AABB) enclosing a more complex object. A ray is intersected with the bounding volume first very quickly and only if the ray hits the bounding volume, intersection with enclosed object is performed. Since the most of the rays misses the object, costly ray-object intersection is frequently substituted by faster ray-bounding volume intersection test and overall performance raises. Natural extension of this technique is enclosing a cluster of several nearby bounding volumes into a larger bounding volume and check it for intersection first. Following the idea results in a *hierarchy* or a tree of bounding volumes with original objects at its leaves. Searching for the nearest intersection starts in the root (where a bounding volume enclosing entire scene is stored) and traverses down the tree. If there is no intersection with a node the whole subtree may be omitted from further calculations.

Another similar way of doing that is *space subdivision*. An uniform grid or an octree is constructed with a list of references in each leaf or cell. Every object at least partially contained in the cell is referenced. During tracing a ray, the cells intersected by the ray are inspected in order as the ray hits them, like in line drawing algorithm. Different alternative to an octree is a *BSP-tree* (*Binary Space Partitioning*), dividing the space into two half spaces by a plane in each node. All the techniques reviewed above provide the greatest benefit in case of the scenes with a high number of objects and balanced trees. As the most effective is usually considered the BSP-tree.

Studying algorithm *RayTrace*, one may notice that the greatest part of all traced rays are the ones shot towards the light sources in step 3, to evaluate contributions to illumination. This leads to the following idea called *light buffer*. Enclose each light source by a unit cube and project every object onto the cube surface. To each cube face create then a list of references to the objects projected on the face. This is realized in preprocessing step. When tracing a ray pointed to the light source, the face of the cube intersected by incoming ray is determined. The only objects potentially blocking the light are those referenced in the list of the face. The ray is needed to test for intersections only with the objects in the list.

Lastly, aim our attention to primary rays, shot from the viewpoint. Many of them doesn't hit any object, but they are lengthy tested in a hierarchical structures described above. The key is again a projection. Project the bounding box of every object onto the screen in preprocessing step, yielding a set of rectangles and in a special auxiliary buffer mark every pixel within any rectangle. Thus the rays completely missing any object are immediately recognized and the remaining ones are tested only with those objects corresponding to intersected rectangles. This technique is known as *vista buffer*. It is not as effective as it could seem (except the scenes with large background areas), since the number of primary rays is relatively small.

Beside the acceleration techniques a number of impressive special effects in the ray tracing field has been developed through the years – soft shadows (shadows without sharp edges) like produced by an area light, substituting the area light by a grid of spotlights; focal blur (parts of the scene in a different depth as the virtual camera is focused are blurred); atmospheric effects (like rainbow) or simulating media like fog, smoke or a flame. Most of them are, however, very time consuming. Ray tracing still remains one of the most grateful time-consumers ever.

Chapter 3
Mathematical model

Now, we know enough about the mathematical and algorithmic background, necessary to understand the principle of our method for rendering spacewarps. Following two chapters cover the core of the thesis, derivation of the idea and its realization.

3.1 Spacewarp Definition

Until now when saying "spacewarp" we relied on intuition and an informal understanding of the term. Here we define it exactly, in accordance with one's intuitive expectations.

Definition 3.1. The *Spacewarp* (or *deformation of the space*) is a bijective and continuous function $\mathbf{w} : \mathbb{R}^3 \to \mathbb{R}^3$.

So the spacewarp takes a point $\mathbf{p}$ in the space and maps it to an another point $\mathbf{w}(\mathbf{p})$. Every point is mapped to an another point. Mapping is bijective, so there is an unambiguous pairing original-image of all points in the space. Bijectivity of the function $\mathbf{w}$ guarantees the inverse function existence. To avoid spacewarps like a "chaotic mix" and to make ray tracing feasible we require the continuity of $\mathbf{w}$ (see definition 2.4).

If all points of the solid are transformed by the spacewarp $\mathbf{w}$, a new solid is obtained, different in shape. However, for the continuity of $\mathbf{w}$ topological qualities (like number of components and holes) are preserved in the image of the solid. We will consider the deformed solid as the original solid in a *warped space*. Meaning of the point's coordinates is different there. A point $\mathbf{p}$ with coordinates $(p_1, p_2, p_3)^T$ is in common Euclidean space a linear combination $\mathbf{p} = p_1\mathbf{i} + p_2\mathbf{j} + p_3\mathbf{k}$ of the orthonormal vectors $\mathbf{i} = (1, 0, 0)^T$, $\mathbf{j} = (0, 1, 0)^T$, $\mathbf{k} = (0, 0, 1)^T$ of the base. Applying spacewarp $\mathbf{w}$, the point $\mathbf{p}$ is transformed to $\mathbf{w}(\mathbf{p})$, thus the same coordinates $(p_1, p_2, p_3)^T$ determines a different point $\mathbf{w}(\mathbf{p})$ instead of $\mathbf{p}$. Since the mapping from coordinates to the point is different from linear combination of the base vectors, we say the space is *warped*.[1]

To conclude it, there are two spaces – *common* and *warped* differing by interpretation of coordinates. A given coordinates $(x, y, z)^T$ determines a point $\mathbf{p} = (x, y, z)^T$ in *common space*, while in *warped space* result is the point $\mathbf{w}((x, y, z)^T) = \mathbf{w}(\mathbf{p})$. The whole solid may be interpreted as we know it in original space or deformed in the warped space. The solid itself "knows" nothing about the space it lies inside. Or in other words, a solid is given by coordinates of its vertices (or by some representation in general) and there is no information about its "true" shape. And the coordinates may be interpreted in infinitely many ways through various spacewarps, yielding a different appearance of the same solid in various warped spaces. Finally, even the common space can be considered as the space warped by the identity spacewarp.

Example 3.1. We give an example of a simple spacewarp in plane $\mathbb{R}^2$ (a "planewarp"?). It is similar to a spacewarp in $\mathbb{R}^3$, but easier to illustrate. Let $\mathbf{w}$ be a function defined

[1] Note that $\mathbf{w}(\mathbf{p})$ and the whole deformed solid lies actually still in Euclidean space $\mathbb{R}^3$ (the range of values of spacewarp $\mathbf{w}$ is $\mathbb{R}^3$). There is an orthonormal base in this space and rectangular coordinates of $\mathbf{p}$ in warped space are given by $\mathbf{w}(\mathbf{p})$.

as follows:

$$\mathbf{w} : \mathbb{R}^2 \to \mathbb{R}^2, \qquad \mathbf{w}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y + \frac{1}{2}x^2 \end{pmatrix}. \tag{3.1}$$

It's easy to verify that $\mathbf{w}$ is bijective and continuous, hence it is a spacewarp in terms of definition 3.1. Corresponding warped space is shown in figure 3.1 with circle as a sample shape deformed accordingly. Parabolic spacewarp $\mathbf{w}$ can be straightforwardly extended to three dimensions

$$\mathbf{w}' : \mathbb{R}^3 \to \mathbb{R}^3, \qquad \mathbf{w}'\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y + \frac{1}{2}x^2 \\ z \end{pmatrix}. \tag{3.2}$$

The result is shown in figure 3.2.

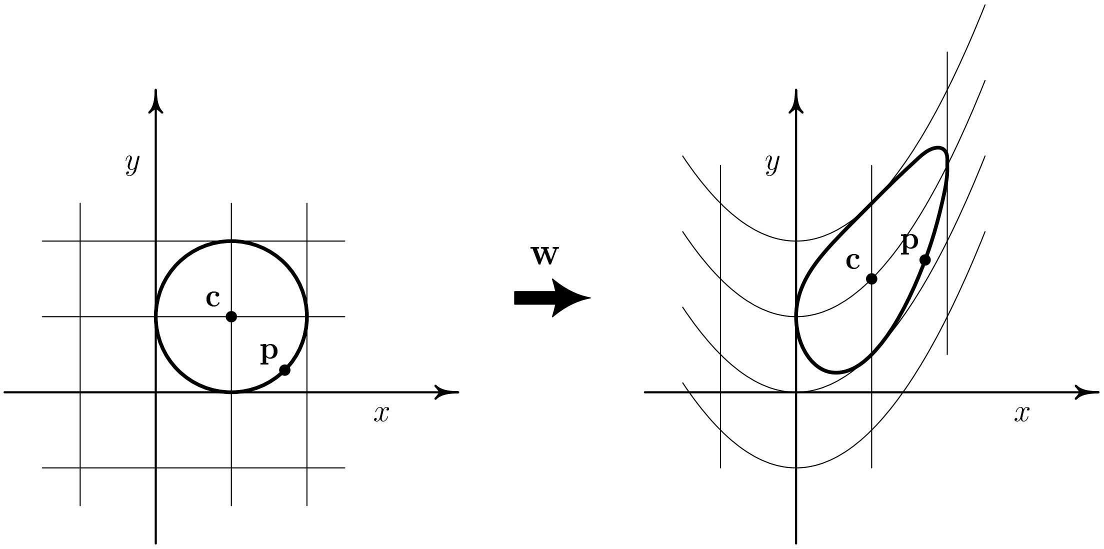


Figure 3.1. A simple spacewarp given by formula (3.1). Common space on the left vs. warped space on the right. The gridlines parallel to coordinate axes are shown to visualize the warp. Note that the points with the same y-coordinate forms a parabola in warped space. Compare the deformed shape of the circle and a different location of the same point $\mathbf{p}$ in common and warped space.

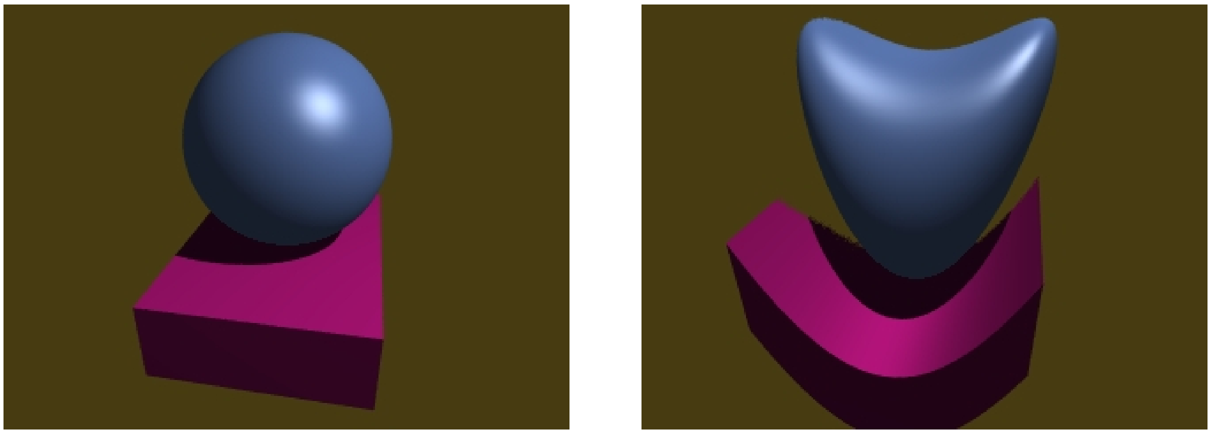


Figure 3.2. Space deformed by 3-dimensional parabolic spacewarp given by formula (3.2). Compare the sphere and the cube in common space (left) and in the warped space (right).

Example 3.2. Another example of a three-dimensional spacewarp is the black hole already mentioned. The black hole spacewarp attracts the space around, towards the black hole center. And its influence diminishes in the distance. Let $\mathbf{b} : \mathbb{R}^3 \to \mathbb{R}^3$ be a black hole spacewarp with center at origin $\mathbf{0}$. Any point $\mathbf{p}$ is attracted directly towards the black hole center – it means that the spatial angle of $\mathbf{p}$ remains unchanged or more technically, point $\mathbf{b}(\mathbf{p})$ lies on the ray from origin to $\mathbf{p}$, but closer to the origin as the $\mathbf{p}$ is. Then the spacewarp $\mathbf{b}$ is in the form

$$\mathbf{b}(\mathbf{p}) = f(\|\mathbf{p}\|) \cdot \mathbf{p} \tag{3.3}$$

where f is a function controlling the influence in dependence on the distance of $\mathbf{p}$ from the center. One suitable function is $f(x) = (1 - e^{-x})$ restricted to $\mathbb{R}^+$ (since the distance is always non-negative). Graph of the function $f(x)$ is shown in figure 3.3. It starts with a zero value and continually approaches to one. So the points nearby origin are affected strongly, while the distant ones are almost unchanged. The complete spacewarp looks then

$$\mathbf{b}(\mathbf{p}) = (1 - e^{\|\mathbf{p}\|}) \cdot \mathbf{p} \tag{3.4}$$

or rewritten to individual coordinates

$$\mathbf{b}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1 - e^{\sqrt{x^2+y^2+z^2}}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}. \tag{3.5}$$

A schematic illustration of that black hole spacewarp is in figure 3.3, for a 3D result see figure 3.4.

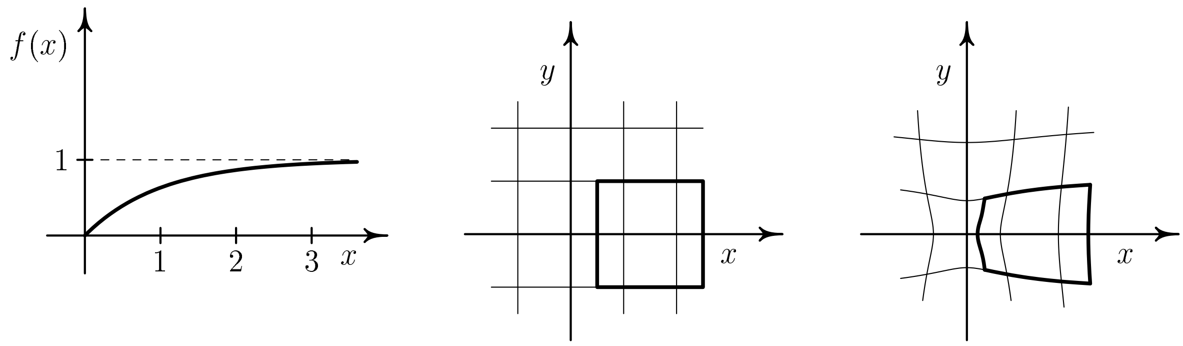


Figure 3.3. Graph of the black hole influence function $f(x) = (1 - e^{-x})$ is shown in the left image. Common space with a sample square in the middle and the space warped by a 2-dimensional analogy of the black hole spacewarp centered at origin and given by formula (3.4) is shown on the right.

Like any other function a spacewarp $\mathbf{w}$ may be *linear*, when $\mathbf{w}$ is a linear mapping (see definition 2.5). Linear mappings includes common geometric transformations like translation, scaling or rotation and their composition. If $\mathbf{w}$ is not a linear mapping, we say that spacewarp is *non-linear*. The spacewarps shown in the two examples above are both non-linear. Dealing with linear spacewarps is incomparably easier and formulas comprising a linear spacewarp are usually resolved analytically. We mention that case in the sections henceforward. On the other hand, non-linear spacewarps though requiring numerical methods and approximations to evaluate, produce much more interesting effects. That's the reason why we aim primarily on non-linear spacewarps in our work. And

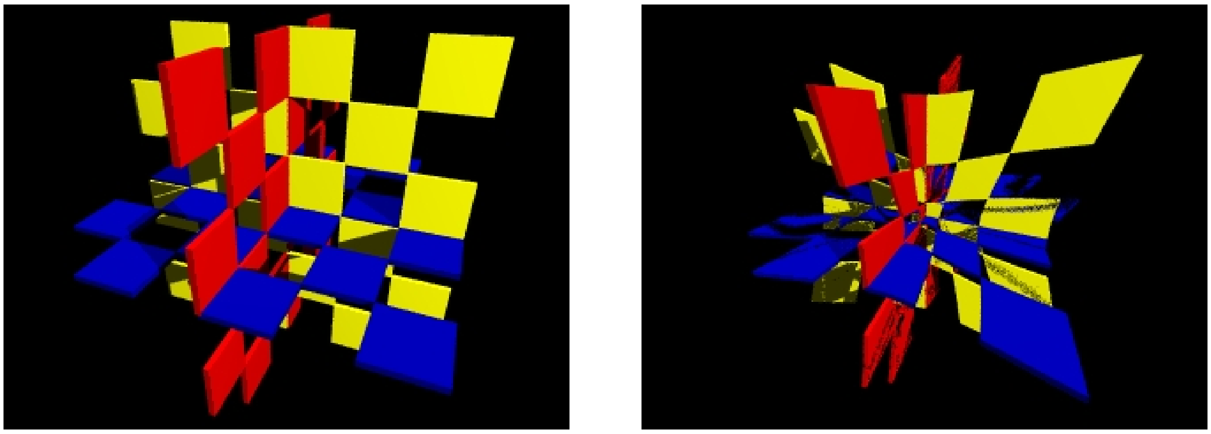


Figure 3.4. Space deformed by the black hole spacewarp centered at origin, given by formula (3.4). A tiling in common space (left) and its counterpart in warped space (right). Some undesired artifacts in the right image were caused by approximation of the inverse spacewarp.

now we at last understand the meaning of all terms from the thesis title – *Ray Tracing Non-Linear Spacewarps*.

3.2 Representations of Geometric Objects in Warped Space

If we wish to visualize objects in warped space, deformed in shape, we need to mathematically describe them first. We have shown some analytic representations in section 2.2. Here we present, how to transform them for the use in warped space, introduced in previous section.

Parametric representation

The generall form of parametric representation $\mathbf{f}$ is given by formula (2.2). For a parameter vector $\mathbf{t}$ it returns a point $\mathbf{p} = \mathbf{f}(\mathbf{t})$ of the represented object. In a space warped by a spacewarp $\mathbf{w}$ (hence a $\mathbf{w}$-warped space) coordinates of the point $\mathbf{p}$ are interpreted in a different way – as the point $\mathbf{w}(\mathbf{p})$. Assignments

$$\mathbf{f} : \mathbf{t} \mapsto \mathbf{f}(\mathbf{t}) =: \mathbf{p} \qquad \text{and} \qquad \mathbf{w} : \mathbf{p} \mapsto \mathbf{w}(\mathbf{p}) \tag{3.6}$$

may be clearly joined together, yielding

$$\mathbf{f} \circ \mathbf{w} : \mathbf{t} \mapsto \mathbf{w}(\mathbf{f}(\mathbf{t})). \tag{3.7}$$

Hence, given a d-dimensional object represented by $\mathbf{f}_d$ and parameter domains $R_1, \ldots, R_d$, its parametric representation in $\mathbf{w}$-warped space is a composition of $\mathbf{f}_d$ and $\mathbf{w}$

$$\mathbf{f}_d \circ \mathbf{w} : \prod_{i=1}^{d} R_i \to \mathbb{R}^3, \quad \text{mapping} \quad \mathbf{t} \mapsto \mathbf{w}(\mathbf{f}_d(\mathbf{t})). \tag{3.8}$$

Implicit representation

The implicit representation F of an object assigns 0 just to all points of the object. In $\mathbf{w}$-warped space points forming the object are transformed to a different location (see figure

3.5). What is the common attribute of all those points in warped space? The answer is: “They all are images of the points satisfying implicit equation F”, or in other words “Transformed by inverse spacewarp $\mathbf{w}^{-1}$ they all satisfy the implicit equation F”.[2] Now writing it formally, implicit equation of the deformed object is

$$F(\mathbf{w}^{-1}(\mathbf{p})) = 0. \tag{3.9}$$

All points satisfying equation (3.9) forms the object in warped space. Concluding it – implicit representation of the object defined by F in $\mathbf{w}$-warped space is given by

$$\mathbf{w}^{-1} \circ F : \mathbb{R}^3 \to \mathbb{R}, \quad \mathbf{p} \mapsto F(\mathbf{w}^{-1}(\mathbf{p})). \tag{3.10}$$

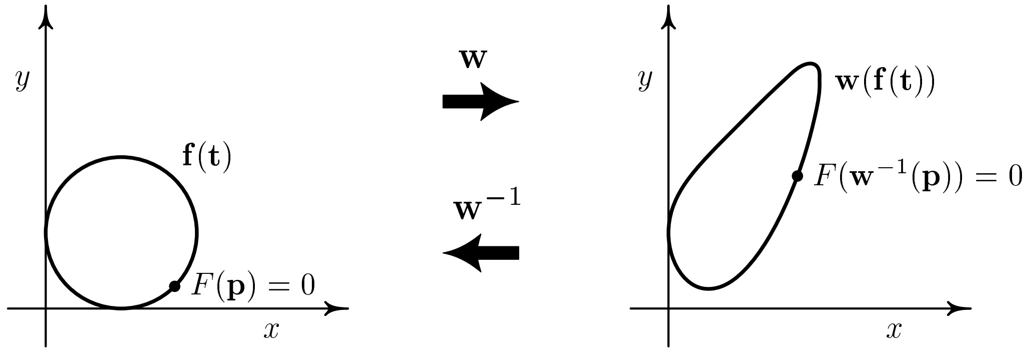


Figure 3.5. Object representation in common space (left) vs. warped space (right). A shape in warped space is obtained by applying spacewarp $\mathbf{w}$ to the original shape and so works also parametric representation $\mathbf{f}$, while in implicit equation for warped space points are transformed back by $\mathbf{w}^{-1}$ and substituted to original implicit equation F.

Other representations

Note that CSG representation has a topological character – it prescribes some set operations performed on geometric primitives expressed themselves analytically. Since a spacewarp doesn’t change topology but only the shape, the CSG representation in warped space is the same tree of set operations, only the analytic representations of geometric primitives in the leaves need to be modified as was explained in above paragraphs, to change their shape according to the specific spacewarp.

Volumetric representation can be regarded as an instance of implicit representation. It is a function assigning the stored values (e.g. density) to the points from discrete 3D grid and in the other points are values evaluated by interpolation (or extrapolation) the grid-points. Thus the volumetric representation turns to our known function $F : \mathbb{R}^3 \to \mathbb{R}$ and its behaviour in warped space is described by formula (3.10).

Analytic representations and linear spacewarps

As we have already mentioned, the life with only the linear spacewarps would be easier. In that case a specific analytic formula for $\mathbf{w}$-warped space can be often derived, avoiding to calculate $\mathbf{w}$ or $\mathbf{w}^{-1}$ within every evaluation of the formula.

A linear spacewarp being a linear mapping preserves linear combinations of vectors from its definition and hence also barycentric coordinates of a point (see definition 2.2).

[2] Existence of the inverse spacewarp $\mathbf{w}^{-1}$ is guaranteed by the bijectivity of $\mathbf{w}$.

Any plane P is determined by three non-colinear points $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$. A point $\mathbf{p}$ of the plane P can be written as a barycentric combination of the points $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$

$$\mathbf{p} = u\mathbf{p}_1 + v\mathbf{p}_2 + w\mathbf{p}_3, \quad u + v + w = 1. \tag{3.11}$$

That point is mapped to $\mathbf{w}(\mathbf{p})$ in the $\mathbf{w}$-warped space. From linearity of $\mathbf{w}$ we get

$$\mathbf{w}(\mathbf{p}) = \mathbf{w}(u\mathbf{p}_1 + v\mathbf{p}_2 + w\mathbf{p}_3) = u\mathbf{w}(\mathbf{p}_1) + v\mathbf{w}(\mathbf{p}_2) + w\mathbf{w}(\mathbf{p}_3) \tag{3.12}$$

which states that $\mathbf{w}(\mathbf{p})$ is a barycentric combination of $\mathbf{w}(\mathbf{p}_1)$, $\mathbf{w}(\mathbf{p}_2)$ and $\mathbf{w}(\mathbf{p}_3)$, hence it lies in the plane determined by them. It is true for any point $\mathbf{p}$ of the plane P. The consequence is that any object with plane faces (triangle, box, polygonal mesh) is in warped space again an object with plane faces determined by its transformed vertices (in case of linear spacewarp!).

Besides the planes and objects with planar faces, the most frequently used solids are *quadratic surfaces*. Sphere, ellipsoid, cylinder, cone, paraboloid, hyperboloid (and even the plane) – they all are quadratic surfaces. The general form of a quadratic surface is the implicit equation

$$ax^2 + by^2 + cz^2 + 2dxy + 2exz + 2fyz + 2gx + 2hy + 2iz + j = 0 \tag{3.13}$$

where $a, \ldots, j$ are constants determining the shape of quadratic surface. We show now that a quadratic surface in linearly warped space is again a quadratic surface and derive its formula.

The general form (3.13) can be rewritten to a matrix form using homogeneous coordinates (see definition 2.1)

$$(x, y, z, 1) \begin{pmatrix} a & d & e & g \\ d & b & f & h \\ e & f & c & i \\ g & h & i & j \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0 \tag{3.14}$$

or denoting the matrix of quadratic surface as $\mathbf{Q}$

$$\mathbf{p}^T \mathbf{Q}\, \mathbf{p} = 0. \tag{3.15}$$

We already know that in implicit representation for $\mathbf{w}$-warped space the point $\mathbf{p}$ needs to be substituted by the inverse warped point $\mathbf{w}^{-1}(\mathbf{p})$. In linear spacewarp $\mathbf{w}(\mathbf{p}) = \mathbf{M}\mathbf{p}$ for the matrix of linear mapping $\mathbf{M}$. Inverse $\mathbf{w}^{-1}(\mathbf{p}) = \mathbf{M}^{-1}\mathbf{p}$. Then the quadratic surface in warped space is given by

$$\begin{aligned} (\mathbf{w}^{-1}(\mathbf{p}))^T \mathbf{Q}\, \mathbf{w}^{-1}(\mathbf{p}) &= 0 \\ \mathbf{p}^T \mathbf{M}^{-1T} \mathbf{Q}\, \mathbf{M}^{-1} \mathbf{p} &= 0 \\ \mathbf{p}^T \mathbf{Q}' \mathbf{p} &= 0 \end{aligned} \tag{3.16}$$

which is a quadratic surface again, with matrix $\mathbf{Q}' = \mathbf{M}^{-1T} \mathbf{Q}\, \mathbf{M}^{-1}$.

3.3 Intersection of Ray with Object in Warped Space

The preparation is now at an end and the time arrived to visualize warped spaces by our highly praised ray tracing method.

Intersection in common space

The fundamental computation in ray tracing is finding an intersection of a ray with an object. It is usually derived from the parametric expression $\mathbf{r}(t)$ of the ray and the implicit representation F of the object. Intersection is such a point $\mathbf{p}$ common to both the ray and the object, hence $\mathbf{p} = \mathbf{r}(t)$ for some t and also $F(\mathbf{p}) = 0$. Substitution yields

$$F(\mathbf{r}(t)) = 0 \tag{3.17}$$

which is an equation of one unknown t. The smallest positive solution t_i of above equation is the parameter corresponding to the nearest intersection point. Its coordinates are obtained then as $\mathbf{r}(t_i)$. If there is no solution of above equation, the ray doesn't intersect the object.

Example 3.3. Remind our faithful friend – the sphere with center point $\mathbf{c}$ and radius r. Its implicit equation is

$$S(\mathbf{p}) = r - \|\mathbf{p} - \mathbf{c}\| = 0. \tag{3.18}$$

The intersection of a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ given by an origin point $\mathbf{o}$ and an unit direction vector $\mathbf{d}$ with the sphere surface is a solution of

$$\begin{aligned}
S(\mathbf{r}(t)) = r - \|\mathbf{r}(t) - \mathbf{c}\| &= 0 \\
\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\| - r &= 0 \\
(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - r^2 &= 0 \\
t^2(\mathbf{d} \cdot \mathbf{d}) + 2t(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 &= 0 \\
t^2 + 2t(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 &= 0.
\end{aligned} \tag{3.19}$$

The last step comes from the fact that $\|\mathbf{d}\| = 1$, hence $\mathbf{d} \cdot \mathbf{d} = 1$. Denoting $\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})$ as b and $(\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$ as c yields an ordinary quadratic equation

$$t^2 + 2bt + c = 0. \tag{3.20}$$

For its solutions t_1 and t_2 the intersection points of the ray and the sphere are $\mathbf{r}(t_1)$ and $\mathbf{r}(t_2)$; see the left image in figure 3.6 for illustration. Intersection with other quadrics like cylinder, cone or hyperboloid can be derived similarly.

Parametric representation of an object is not so friendly for deriving intersections. Intersection is the same point generated by both parametric ray $\mathbf{r}(t)$ and parametric object $\mathbf{f}(\mathbf{t})$, thus solving

$$\begin{aligned}
\mathbf{r}(t) &= \mathbf{f}(\mathbf{t}) \\
\mathbf{r}(t) - \mathbf{f}(\mathbf{t}) &= 0
\end{aligned} \tag{3.21}$$

for unknowns t and $\mathbf{t}$ we get parameters corresponding to an intersection point. Note that (3.21) is actually a system of three equations with several unknowns – t and those comprised in vector of parameters $\mathbf{t}$, which is obviously more difficult to solve than (3.17).

Intersection in warped space

Turn to the warped space now, say given by a spacewarp $\mathbf{w}$. Here an implicit equation of a deformed object has the form $F(\mathbf{w}^{-1}(\mathbf{p})) = 0$ as explained in previous section.

Intersection ray-object is such a point $\mathbf{r}(t)$ of the ray satisfying the implicit equation; we express it again by substitution $\mathbf{r}(t)$ to the implicit equation

$$F(\mathbf{w}^{-1}(\mathbf{r}(t))) = 0. \tag{3.22}$$

Its smallest solution t_i corresponds to the nearest intersection point $\mathbf{r}(t_i)$.

Example 3.4. Consider again the sphere with center point $\mathbf{c}$ and radius r – our eternal friend and the three-dimensional parabolic spacewarp $\mathbf{w}'$ from example 3.1, given by formula (3.2). Its inverse is

$$\mathbf{w}'^{-1}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y - \frac{1}{2}x^2 \\ z \end{pmatrix}. \tag{3.23}$$

Implicit equation of the sphere in $\mathbf{w}'$-warped space is $\mathbf{w}'^{-1} \circ S$, thus

$$\mathbf{w}'^{-1} \circ S(\mathbf{p}) = r - \|\mathbf{w}'^{-1}(\mathbf{p}) - \mathbf{c}\| = 0 \tag{3.24}$$

and rewritten to the coordinate components (see formula (2.7))

$$\mathbf{w}'^{-1} \circ S\begin{pmatrix} x \\ y \\ z \end{pmatrix} = r^2 - (x - c_1)^2 - (y - \frac{1}{2}x^2 - c_2)^2 - (z - c_3)^2 = 0. \tag{3.25}$$

Substitution of $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ for point $(x, y, z)^T$, thus $x = o_1 + td_1$, $y = o_2 + td_2$ and $z = o_3 + td_3$ yields a fourth order equation for t. Its solutions corresponds to intersection points of the ray and the warped sphere. See the right image in figure 3.6 and note that as many as four intersections are possible there.

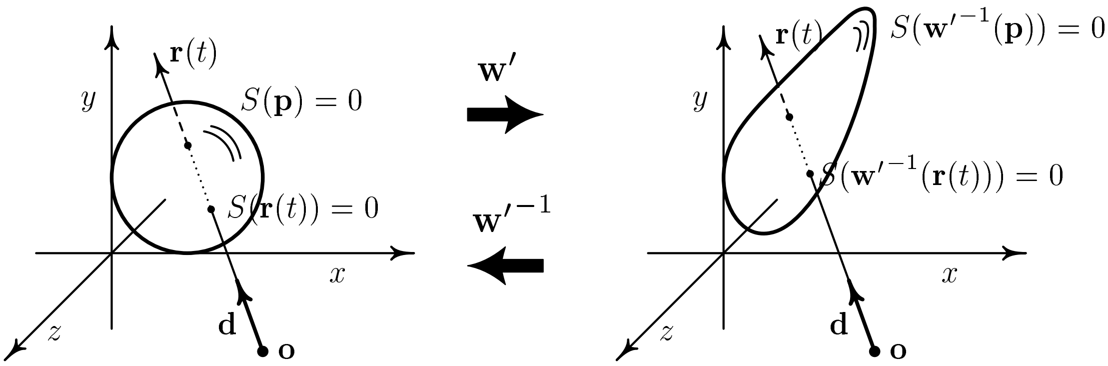


Figure 3.6. Illustration to examples 3.3 and 3.4. Intersection of a ray with a sphere in common space (left image) and a sphere in $\mathbf{w}'$-warped space (right image).

A careful reader may ask "Why we consider the straight rays while being in warped space?" A scene in warped space is an ordinary scene with deformed solids. We are free to visualize that scene using whichever method and we have chosen standard ray tracing as described in chapter 2.

By the way, ray tracing the deformed scene by the straight rays is inevitable. Consider an observer inside a scene and place it all to a warped space. Assuming that light rays follows the warp of the space, observer doesn't notice any difference, since all intersections of the rays and the scene are the same just transformed by the spacewarp. As we already discussed when introducing spacewarp in section 3.1, "one never knows if he is inside a warped space or not". However, then all the work about space-warping is pointless. To see the deformation of space, rays must be traced straight. A simple and imaginable physical interpretation justify that feature. The light rays since being massless remains unaffected by the spacewarp and hence straight, unlike the material solids in the scene. It may be a helpful imagination, but the primary reason for straight rays is the first one said.

Parametric representation in the warped space is given by formula (3.8). Like in Euclidean space, intersection being a common point to both ray and object is the same point generated by their parametric expressions. Thus the solutions of the following equation corresponds to intersection points of an object $\mathbf{f}$ and a ray $\mathbf{r}$ in $\mathbf{w}$-warped space

$$\begin{aligned} \mathbf{r}(t) &= \mathbf{w}(\mathbf{f}(\mathbf{t})) \\ \mathbf{r}(t) - \mathbf{w}(\mathbf{f}(\mathbf{t})) &= 0. \end{aligned} \tag{3.26}$$

Note that it's a system of three equations with several unknowns t and $\mathbf{t}$ again.

3.4 Illumination Calculations in Warped Space

Visualizing a scene by ray tracing algorithm does not end by finding a ray-object intersection. Illumination is evaluated for the point on the surface depending on the angle of incidence of incoming light. Moreover an reflected ray is generated following the law of reflection which states that the angle of incidence is equal to the angle of reflection. And possibly a refracted ray due to the law of refraction and material's index of refraction. For illumination evaluation as well as for determining directions of the generated secondary rays, the normal vector to the surface is needed.

The normal vector is calculated directly for the standard solids or as the cross product of two independent tangent vectors. Let $\mathbf{u_p}$ be a tangent vector to the surface in the point $\mathbf{p}$. As shown in [Barr84] corresponding tangent vector to the deformed surface in $\mathbf{w}$-warped space is obtained by

$$\mathbf{w}(\mathbf{u_p}) = \mathbf{J_p}\mathbf{u_p} \tag{3.27}$$

where $\mathbf{J_p}$ is the Jacobian matrix $\mathbf{J}$ of the spacewarp function $\mathbf{w}$ defined as

$$\mathbf{J} = \left(\frac{\partial \mathbf{w}}{\partial x}, \frac{\partial \mathbf{w}}{\partial y}, \frac{\partial \mathbf{w}}{\partial z}\right) \tag{3.28}$$

evaluated in the point $\mathbf{p}$. The cross product of two independent tangent vectors transformed by formula (3.27) yields the normal vector to the deformed surface.

The normal vector $\mathbf{n_p}$ to the deformed surface in the point $\mathbf{p}$ can be calculated also directly without the tangent vectors as follows:

$$\mathbf{w}(\mathbf{n_p}) = \det \mathbf{J_p}\mathbf{J_p}^{-1T}\mathbf{n_p}. \tag{3.29}$$

Here $\mathbf{J_p}$ stands again for Jacobian matrix of the spacewarp $\mathbf{w}$ evaluated in the point $\mathbf{p}$. The proof is presented in [Barr84]. Calculating $\det \mathbf{J_p}$ may be actually omitted, since only the direction of the normal vector is significant. Note that if the function $\mathbf{w}$ is not differentiable in the point $\mathbf{p}$, then $\mathbf{J_p}$ cannot be evaluated, but in this case an edge is passing through the point $\mathbf{p}$ and there is no clear normal vector defined either.

Example 3.5. Look at the figure 3.1 showing the 2-dimensional parabolic spacewarp $\mathbf{w}$ given by formula (3.1) and a circle deformed accordingly. We compute the normal vector in the warped space for the marked point $\mathbf{p}$. The Jacobian matrices of the function $\mathbf{w}$ and its inverse are[3]

$$\mathbf{J} = \begin{pmatrix} 1 & 0 \\ x & 1 \end{pmatrix} \qquad \mathbf{J}^{-1} = \begin{pmatrix} 1 & 0 \\ -x & 1 \end{pmatrix}. \tag{3.30}$$

Particular $\mathbf{J_p}$ and $\mathbf{J_p}^{-1}$ for the marked $\mathbf{p} = (1 + \frac{\sqrt{2}}{2}, 1 - \frac{\sqrt{2}}{2})^T$ are

$$\mathbf{J_p} = \begin{pmatrix} 1 & 0 \\ 1 + \frac{\sqrt{2}}{2} & 1 \end{pmatrix} \qquad \mathbf{J_p}^{-1} = \begin{pmatrix} 1 & 0 \\ -1 - \frac{\sqrt{2}}{2} & 1 \end{pmatrix} \tag{3.31}$$

and the unit normal vector $\mathbf{n_p}$ in the point $\mathbf{p}$ is $\mathbf{n_p} = (\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2})^T$. Then the normal vector in warped space is transformed to

$$\mathbf{w}(\mathbf{n_p}) = \det \mathbf{J_p} \mathbf{J_p}^{-1T} \mathbf{n_p} = 1 \cdot \begin{pmatrix} 1 & -1 - \frac{\sqrt{2}}{2} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} + \sqrt{2} \\ -\frac{\sqrt{2}}{2} \end{pmatrix}. \tag{3.32}$$

The result should be additionally adjusted to the unit length. Check figure 3.7 depicting the situation.

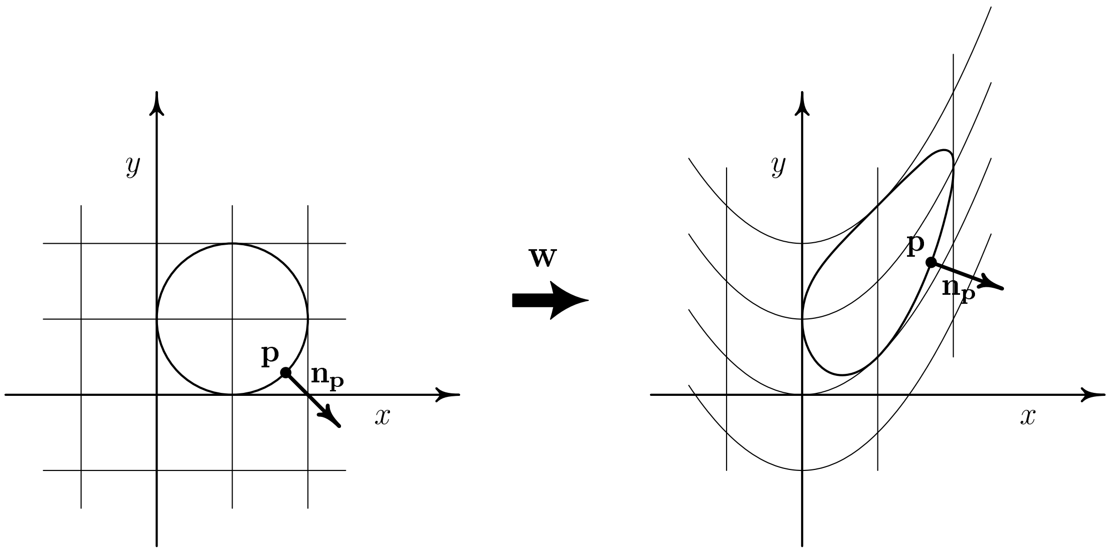


Figure 3.7. Illustration to example 3.5. Normal vector transformation from common to the warped space.

3.5 A Different View to Ray Tracing in Warped Space

The key turn of considering is following here. In section 3.3 we have derived equation $F(\mathbf{w}^{-1}(\mathbf{r}(t))) = 0$ for intersection of a straight ray and a deformed solid in the warped

[3] Note that $\mathbf{J}^{-1}(\mathbf{w}) = \mathbf{J}(\mathbf{w}^{-1})$.

space as substituting the points $\mathbf{r}(t)$ of the ray into the implicit equation $F(\mathbf{w}^{-1}(\mathbf{p})) = 0$ of deformed solid. That equation can be, however, interpreted also otherwise – as a substitution of the points $\mathbf{w}^{-1}(\mathbf{r}(t))$ into the implicit equation $F(\mathbf{p}) = 0$ of the original solid. Thus its solution is a point generated by the univariate parametric representation $\mathbf{p} = \mathbf{w}^{-1}(\mathbf{r}(t))$ satisfying $F(\mathbf{p}) = 0$, hence an intersection of the objects represented by parametric $\mathbf{w}^{-1}(\mathbf{r}(t))$ and implicit $F(\mathbf{p}) = 0$. So what object is produced by the parametric representation $\mathbf{w}^{-1}(\mathbf{r}(t))$? It clearly maps every point $\mathbf{r}(t)$ of the straight ray to its inverse under spacewarp $\mathbf{w}$, thus $\mathbf{w}^{-1}(\mathbf{r}(t))$ represents the curve which is the inverse image of the straight ray $\mathbf{r}(t)$ under $\mathbf{w}$. The curve is continuous (from continuity of $\mathbf{w}$) and transformed by spacewarp $\mathbf{w}$ it yields the straight ray $\mathbf{r}(t)$.

We have proved now the fact mentioned already in introduction, that intersection of the ray and a warped solid is the same as intersection of the inverse warped, curved ray and the original solid. Expanding the idea results in following method for rendering a scene in warped space. The whole scene remains as it is, unmodified and all rays become the curves made as inverse image of the straight rays under the spacewarp. Note that not only the rays have to be inverse warped, but all rendering agents as well, it means the viewpoint and projection plane, since they determine generated rays. See figure 3.8 for a visual image of the situation. Using this principle it's evident that the scene "suspect" nothing about any warping, absolutely no coordinates, vertices or representations of the solids change, it is just rendered by an unusual method with curved rays instead of straight. That's one of the reasons why we say "spacewarp" and not a "global deformation of the scene".

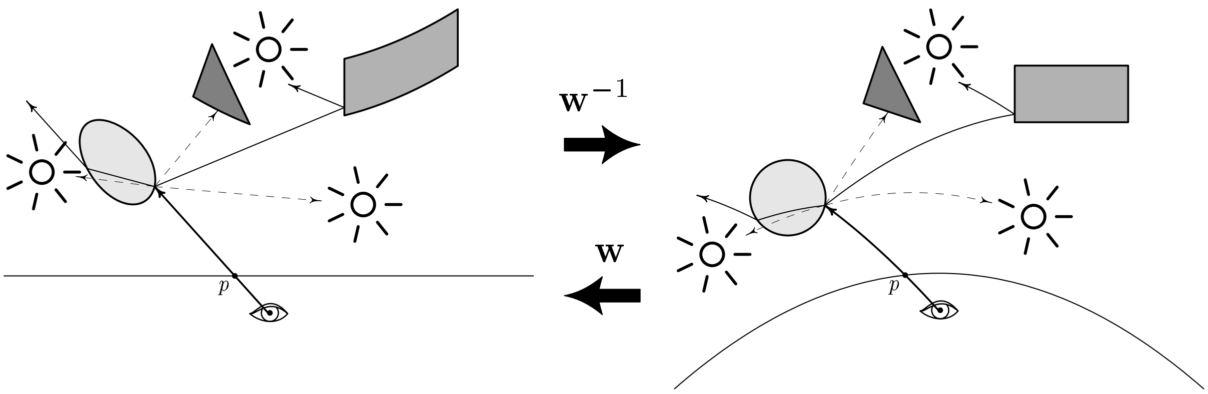


Figure 3.8. Ray tracing a scene in warped space. Duality of tracing straight rays in deformed scene and tracing curved rays in unchanged scene. Note that the final color of the pixel p will be identical in both cases.

Identical result can be derived from intersection of a ray and a parametric solid in warped space given by formula (3.26). Applying $\mathbf{w}^{-1}$ to both sides we get an equivalent equation (since $\mathbf{w}^{-1}$ is bijective)

$$\begin{aligned} \mathbf{r}(t) &= \mathbf{w}(\mathbf{f}(\mathbf{t})) \quad / \, \mathbf{w}^{-1} \\ \mathbf{w}^{-1}(\mathbf{r}(t)) &= \mathbf{f}(t) \end{aligned} \tag{3.33}$$

which is clearly an intersection of the inverse warped ray and the original solid.

Hence in the text we will use also the terms "ray space" and "object space" for warped and common space respectively. Keep in mind the main characteristics – in *ray space* the rays are straight moving in the deformed scene, whereas in *object space* the objects are intact and the traced rays are curved. In figure 3.8 the ray space is on the left and object space on the right.

Finally we will deal with some features and pitfalls related to the light calculations. Illumination of a point on the surface and direction of the reflected ray both depend on the angle between the incoming ray and the normal to the surface. But calculating the angle in the object space or in the ray space, the result may be different, see figure 3.9. Also direction of the refracted ray depends on that angle. It's a question in which space should be the angle evaluated. Realizing it in ray space with straight rays corresponds to only the shape warping, material that the solids are made from remains identical. On the other hand when warping the space, optical qualities of materials (e.g. density or index of refraction) may change and the light behaviour should be different. This approach corresponds to calculating light in the object space. We use the former approach as closer to one's expectations[4] and a little bit simpler to implement in POV-Ray than the latter. However it's just the matter of choice.

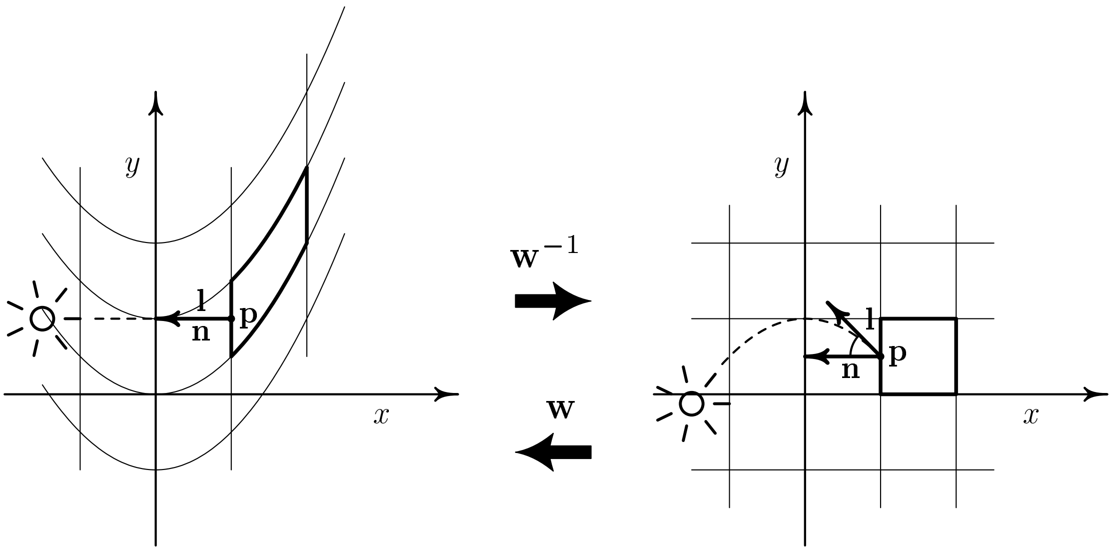


Figure 3.9. Angle between a ray and surface normal may be different in ray space (left) and in object space (right).

The light in the real world attenuates due to the traveled distance. In ray tracing it is possible as well and like the other light calculations it will be evaluated in ray space. The only calculations related to the light and performed in object space is about spot light sources. Intensity of the emitted light ray depends on the angle between the ray and the direction of the spotlight. And that angle must be evaluated in the object space, since the cone of emitted light is deformed in the ray space (it doesn't have to be a cone at all). It is essential for preserving the consistency between the shape of the light "cone" and a deformed object modeling the light source (e.g. a reflector or a lampshade) in the warped space.

[4] Look at the figure 3.9. In case that illumination is evaluated in ray space, around the point **p** will be a bright spot with maximal light intensity in **p** since the light is perpendicular to the surface, while if doing so in object space, the bright spot of maximal intensity will be elsewhere.

Chapter 4
Realization of the Mathematical Model or Return to Reality

In previous chapter we have derived formulas needed by ray tracing the warped scene. We have presented a new view to the problem as well, based on tracing curved rays. Those results have, however, highly theoretical character and an effort to realize it all in practice turns into a tough struggle.

4.1 Numerical Methods

All the intersection calculations and hence the entire ray tracing in warped space stands on the equation (3.22). Roots of a nonlinear continuous function $G = \mathbf{r} \circ \mathbf{w}^{-1} \circ F$, $G : \mathbb{R} \to \mathbb{R}$ are being found there, corresponding to intersections. We have simply said "...and solving the equation...", but for a more sophisticated spacewarp that equation can be really "monstrous", in fact. Even for as simple spacewarp as the parabolic one (see example 3.1) a fourth order equation is obtained reaching the frontiers of analytical solubility. Though we cannot find the roots of $G(t)$ analytically, its values in any particular point t may be evaluated. Our hope is then the numerical analysis and approximation methods.

Unfortunately it isn't an universal remedy either. The means of numerical math are based on evaluating some selected discrete points and estimating the course of the function between those points. For a true "black-box" nonlinear function though evaluating the value in any number of points, nothing can be concluded about the other points. Or in a geometric view the curve of the ray can always contain a very thin "tentacle" skipped in evaluating the discrete points of the ray but intersecting a surface. So from a theoretical point of view there is *no* universal and absolutely reliable method for equation (3.22) solving, moreover such method *cannot* exist for the above reasons. This is the price of freedom for the spacewarp function. As few restrictions as possible has been prescribed. On the other hand the function G may be approximated with any precision. And last but not least the goal is a visual impression and for this purpose is a too much treacherous and shaggy spacewarp function useless in practice.

Iterative methods

There are some clever iterative methods for finding roots of a non-linear function $f : \mathbb{R} \to \mathbb{R}$ approximately in the numerical analysis, e.g. regula falsi or the Newton's method. They are fast convergent and quite reliable, but all are based on one key assumption – an interval $\langle a, b \rangle$ containing a single root is known, that $f(a)$ and $f(b)$ differs in sign in case of regula falsi method, or the root is relatively near to the initial iteration for Newton's method. In our instance we have no such knowledge, the function G characterizing intersections may attain the zero value in several different points and we are interesting in the smallest one. This is the reason precluding one from using mentioned methods directly – a *separation of the roots* is needed first.

Approximation

Another way is *approximation* of the function G by a combination of a few elementary functions, e.g. polynomials, manageable to work with. The roots of polynomials can be separated using algebraic methods and Sturm's theorem. The methods from previous paragraph are afterwards feasible to use for an interval containing the smallest root. A more practical way unlike approximation by a high order polynomials, separation of the roots and an iterative method consecutively is a spline approximation using a number of low-degree, e.g. cubic polynomials. The cubic polynomials are soluble analytically and more suitable for modeling the course of an arbitrary function due to their lesser oscillation. So the final algorithm is as follows. Evaluate the function G gradually in small discrete steps starting with $G(0)$ and interpolate each four consecutive values by a cubic polynomial. Running into a polynomial with a root within the interpolated interval, its smallest root is a good approximation of the G's smallest root. Moreover it can be further iterated by Newton's method if even more precise solution is needed.

The proposed algorithm is absolutely useless, in fact. The intersection function G is constructed for each ray-object pair and evaluated in a number of points. It includes also the inverse spacewarp evaluation which may be quite a time consuming task as we will see in the next section. Rendering one image would take an unimaginable amount of time. This extreme approach takes a lot of time, but requires almost no memory (storing only four iterations at once). Its pole apart is the approach presented in introduction – warping vertices of tesselated model of the scene. While rendering rapidly its memory requirements are high to store smoothly tesselated model. Those are extremities of the time-space trade-off known in the other areas of computer science as well. Let's try to find a reasonable compromise. So what about approximation of the ray curve $\mathbf{w}^{-1}(\mathbf{r}(t))$ once, storing it and intersecting with the objects? The amount of calculations is considerably lower comparing to G approximation and the memory requirements are medium as well. We will investigate that idea in section 4.3. Computation of the inverse function $\mathbf{w}^{-1}$ to the spacewarp is treated before, since it strongly affects the way of the curved ray approximation.

4.2 Inverse Spacewarp Calculation

In equation (3.22) characterizing the intersection of a ray and a warped solid or equivalently a curved ray and an original solid is the inverse function to the spacewarp comprised. Even when using a parametric representation of the solid, formula (3.26) is a kind of the inverse problem again. Resolving the system of equations, the value of inverse spacewarp may be derived (hint: for cleverly chosen function $\mathbf{r}$ and applying $\mathbf{w}^{-1}$ to both sides of the equation), thus it cannot be easier task than inverting $\mathbf{w}$. An cardinal problem is that given an arbitrary function its inverse may be impossible to express. Not hard to find, but its expression in closed form through elementary functions really does not exist. That fact clearly forces the use of numerical methods again. Our spacewarp function is moreover multidimensional, making the evaluation of the inverse function value even harder. We propose a couple of methods.

Supplied by designer

The first and simplest way for the inverse spacewarp computing is expect the designer of the spacewarp to supply a procedure calculating the inverse spacewarp in any given point. The designer has the best picture of the spacewarp in his mind, the intuition or an other information about the function; a computer program has none of that. Designer may also adjust the spacewarp function to get a similar visual result but with a more feasible inverse function. Maybe it can sound a little bit alibisticly but one can write inverse function quite often and that is the case of almost all spacewarp examples in the thesis. This way of working out is also the fastest one in rendering stage.

Reducing the dimensionality

Another method, a little bit similar to the previous one – it is also designer assisted is reducing the dimensionality of the inverse spacewarp function. Consider a function $f : \mathbb{R} \to \mathbb{R}$, bijective and continuous. Those properties implies also a monotonicity of the function f. Approximation of the inverse function value in a given point is for univariate monotone function f quite simple. Given a y we wish to find such x that $f(x) = y$. That corresponds to finding a zero point of the function $g(x) = f(x) - y$. Function $g(x)$ is bijective and monotone as well, so a wide enough interval $\langle a, b\rangle$ can be chosen such that $g(a)$ and $g(b)$ differs in sign and thus the desired x lying within the interval $\langle a, b\rangle$ is located using regula falsi method after a couple of iterations.

Actually a number of spacewarps modifies just a one dimensional value, though maybe hidden on the first sight. For instance a black hole spacewarp, although changing all three coordinates only a new distance to the black hole center is assigned to any point in fact, which is one-dimensional. Then inverse spacewarp just computes the distance to the black hole center, applies the one dimensional inverse distance function (see previous paragraph) and adjusts the x, y, z components due to the new distance. Or in a whirl-like spacewarp may change two more or less independent one-dimensional values – rotation and radius.

Another improvement in our case when computing the inverse function millions times is not using the regula falsi method on $f(x) - y$ but to approximate the function $f(x)$ by a table of precomputed values (x_i, y_i) in a reasonable wide interval, sorted by y_i. Given a y, an interval $y_i \leq y \leq y_{i+1}$ is located first – corresponding value x lies then between x_i and x_{i+1} and its value is interpolated from x_i, x_{i+1}. The table may consist of thousands of entries for faithful approximation, still occupying some insignificant kilobytes of memory and for the uniform intervals $\langle y_i, y_{i+1}\rangle$ locating for any given y may be realized in the time $\mathcal{O}(1)$.

General inversion method

We have developed a general iterative algorithm approximating the value of inverse function for the spacewarps which cannot be handled by previous methods. It is a heuristic one, of course (reliable method cannot exist as we have explained), but quite successful. It's based on the following idea: given a $\mathbf{y}$ and searching for $\mathbf{x}$ that $\mathbf{w}(\mathbf{x}) = \mathbf{y}$, let's have an iteration $\mathbf{x}_i$. Transforming a neighborhood $\mathcal{N}_{\mathbf{x}_i}$ of the $\mathbf{x}_i$, a neighborhood $\mathcal{N}_{\mathbf{y}_i}$ of $\mathbf{y}_i = \mathbf{w}(\mathbf{x}_i)$ is obtained (from continuity of $\mathbf{w}$). There are some points in the $\mathcal{N}_{\mathbf{x}_i}$, whose images in the $\mathcal{N}_{\mathbf{y}_i}$ are closer to $\mathbf{y}$ as $\mathbf{y}_i$ is. Take one of them as a new, better iteration $\mathbf{x}_{i+1}$ and iterate the process. Again from continuity of $\mathbf{w}$ when $\mathbf{y}_i, \mathbf{y}_{i+1}, \mathbf{y}_{i+2}, \ldots$ are approaching to $\mathbf{y} = \mathbf{w}(\mathbf{x})$ then $\mathbf{x}_i, \mathbf{x}_{i+1}, \mathbf{x}_{i+2}, \ldots$ are approaching to $\mathbf{x}$ and $\mathbf{y}_i$'s can get arbitrarily close to $\mathbf{y}$, thus also $\mathbf{x}_i$'s can to unknown $\mathbf{x}$.

In practice the neighborhood of the point $\mathbf{x}_i$ is approximated by a set of points on a sphere surface with center in $\mathbf{x}_i$. The points on the sphere surface are uniformly spread in θ parallels, ϕ points on each, except the north and south pole with only a single point. The constants ϕ and θ are user-definable yielding the number of points $(\theta - 2)\phi + 2$.

Triangulated sphere surface is then transformed by the spacewarp $\mathbf{w}$. An intersection is calculated between the ray from transformed center $\mathbf{y}_i$ to the inversion point $\mathbf{y}$ and the mesh of warped sphere surface, and the vertex $\mathbf{v}$ of intersected triangle nearest to the intersection point is identified. If $\mathbf{v}$ is nearer to the inversion point $\mathbf{y}$ than i-th iteration $\mathbf{y}_i$ is, then the counterpart of $\mathbf{v}$ on the original sphere surface will be the next iteration $\mathbf{x}_{i+1}$. Else the sphere radius is reduced and the above process repeated. The iteration process is started with $\mathbf{x}_0 = \mathbf{0}$ and the sphere radius $r_0 = 1$, and stopped when $\|\mathbf{y} - \mathbf{y}_i\| < \epsilon$ for a chosen error ϵ or if a maximal allowed number of iterations was reached, with result $\mathbf{x}_i$ as an approximation of $\mathbf{w}^{-1}(\mathbf{y})$. See Figure 4.1 for 2D visualization of the iteration process (further improved following the next paragraph).

There are some important enhancements possible to the basic algorithm. First, the vertices on the sphere surface are rotated by a random amount in each iteration yielding a more stable algorithm avoiding to repeat a special unfortunate case. Second, the intersected triangle of the sphere surface is recursively subdivided several times to get a vertex closer to the ray $\mathbf{y}_i\mathbf{y}$. The sequence $\mathbf{y}_i, \mathbf{y}_{i+1}, \mathbf{y}_{i+2}, \dots$ is approaching towards $\mathbf{y}$ directly and the convergence is faster. And last, it is highly inefficient to step from $\mathbf{0}$ towards a very distant $\mathbf{x}$ in hundreds of iterations using the spheres with radius $r = 1$. The radius for the next iteration is estimated from the previous progress. Let in the iteration $\mathbf{x}_i$ there was distance $d_i = \|\mathbf{y} - \mathbf{y}_i\|$ and using radius r_i the distance was reduced to $d_{i+1} = \|\mathbf{y} - \mathbf{y}_{i+1}\|$. A step with the length $\|\mathbf{y}_{i+1} - \mathbf{y}_i\|$ was done with radius r_i and there is still the distance d_{i+1} remaining. So the new radius r_{i+1} should be $d_{i+1}/\|\mathbf{y}_{i+1} - \mathbf{y}_i\|$ times greater than r_i, yielding

$$r_{i+1} = \frac{d_{i+1} r_i}{\|\mathbf{y}_{i+1} - \mathbf{y}_i\|}. \tag{4.1}$$

It is a linear extrapolation in fact. For instance in case of the identity spacewarp a point in any distance is inverted within some 5 iterations to a reasonable precision ($\epsilon = 10^{-5}$) using this improvement.

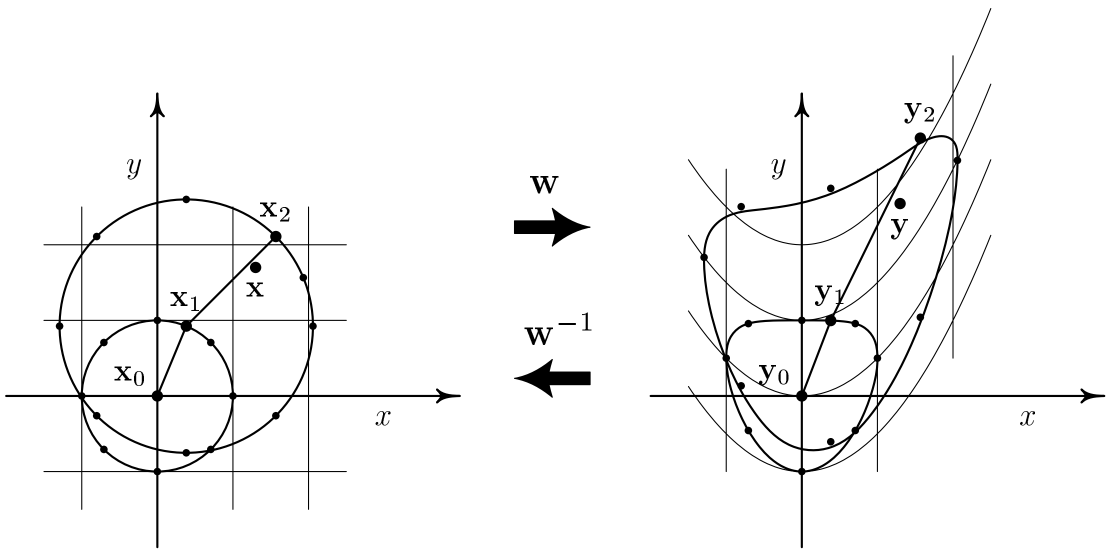


Figure 4.1. Progression of the general inversion algorithm for the 2D parabolic spacewarp (see formula (3.1)). Inversion is computed in the point $\mathbf{y} = (1.3, 2.545)^T$, the iteration process converges to the point $\mathbf{x} = \mathbf{w}^{-1}(\mathbf{y}) = (1.3, 1.7)^T$.

The proposed method is pretty universal and it achieves the best results with a light deformations of the space. The stronger the deformation, the more iterations is required and the less reliability. Serious problems occurs near the singularities of the spacewarp. In case of the 2D parabolic spacewarp the singularities are $(-\infty, 0)^T$ and $(\infty, 0)^T$. Spheres with high x-coordinates are stretched to a long "spaghetti" causing troubles to handle. This method should be used only if the previous ones fails. It is incomparably slower, because of tens or hundreds of ray-triangle intersection calculations and the spacewarp function evaluations.

4.3 Tracing the Curved Ray

In section 4.1 we have presented an approximation of the ray curve $\mathbf{w}^{-1}(\mathbf{r}(t))$ and intersecting it with the objects in the scene as a reasonable compromise between the time and memory requirements. There are several types of approximation, e.g. interpolation, the least squares method, Chebyshev approximation (see [RALS78]), appropriate due to the character of approximated data. The difference is in the value to be minimized in a particular type of approximation. For our purpose the best choice is the interpolation, with zero error in interpolated points, since we know exact (or almost exact in case of $\mathbf{w}^{-1}$ approximation) values there.

So we evaluate the points of the curved ray (we sample the curve) in some discrete steps and store them. By interpolation the shape of the continuous curve is obtained and may be intersected with objects in the scene instead of approximation of the function $G = \mathbf{r} \circ \mathbf{w}^{-1} \circ F$ and finding its zero value for each ray-object pair. The time requirements are medium compared to the other approaches (see end of section 4.1). However, consider that an intersection is found in the first third of the curved ray. Then approximation of the remaining two thirds is just wasting the time and memory. Much more clever is approximating and testing for intersection gradually and stop when the first intersection is found. The time is saved and memory requirements are reduced to minimum (just a few last points of the curved ray to interpolate the last step).

Another issue is when to stop sampling of the ray. Even if a curved ray leaves the scene it may later turn back and return or to cycle in an infinite spiral. Because of the black-box conception of the almost unrestricted spacewarp function we cannot predict anything about the curve shape. The only way is to stop sampling after a given number of steps. If the ray intersects nothing so far, the background color is returned. If some parts of the scene were lost in the rendered image, the number of steps should be increased.

The most reasonable interpolation is polynomial, easy and fast to calculate by computer. The last question is degree of the polynomials. Higher order polynomials are absolutely inappropriate because of the strong oscillation and computational time. Commonly used interpolations are linear, quadratic or cubic. The quadratic and cubic interpolation is more precise than linear with the same number of sampled points, the intersection calculation is however more complex and slower. The great advantage of the linear interpolation (i.e. approximation of the curve by the line segments) is in possibility to use all existing procedures for ray-object intersection, only a test if the intersection is within the line segment interval is needed. That is the reason why we choose it; more faithful approximation can be achieved by denser sampling.

A couple of sampling methods varying in efficiency follows. The ray to trace will be given in ray space by its origin $\mathbf{o}$ and direction $\mathbf{d}$ as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$. Since the inverse spacewarp $\mathbf{w}^{-1}$ is needed in evaluating the curved ray $\mathbf{w}^{-1}(\mathbf{r}(t))$ points and it is a great difference whether $\mathbf{w}^{-1}$ is calculated directly by a given procedure or by the time-consuming general inversion method, the sampling methods are divided into two parts. Those using the inverse spacewarp – appropriate if the procedure is given and those using only the spacewarp function $\mathbf{w}$ avoiding to compute $\mathbf{w}^{-1}$ by general inversion method.

Sampling the curved ray using the inverse spacewarp

The following methods may be characterized as *piecewise inversion* – the points of a ray given in ray space are evaluated for increasing t and inverted by $\mathbf{w}^{-1}$ to object space each.

Constant step sampling. The simplest and naive method of sampling the curved ray is the one with a constant step Δ in parameter t. The $\mathbf{r}(0), \mathbf{r}(\Delta), \mathbf{r}(2\Delta), \mathbf{r}(3\Delta), \ldots$ are evaluated gradually and inverted to object space determining the curve approximated by a polygon (see figure 4.2(left)). Each line segment is immediately tested for intersection with objects in the scene. The classical problems of uniform sampling appears here, e.g. inaccuracy in highly curved regions and overall inefficiency.

Curvature based adaptive sampling. Curvature based sampling eliminates the inefficiency of the uniform sampling. If the angle between two consecutive line segments in curve approximation is lesser than a given δ, they are substituted by one line segment tested for angle again with the next line segment.[1] The result is that in flat regions a lot of short segments is substituted by a long one which is tested for intersection afterwards (see figure 4.2(middle)). More efficient sampling allows to reduce the basic step Δ, yielding more precise approximation in highly curved regions comparing to the previous method.

Proximity based adaptive sampling. The previous method gives a fine approximation of the curve. But note that the goal is not to achieve a faithful approximation of the curve, but a reasonably accurate position of the intersection point, in fact. So absolutely sufficient will be dense sampling near the objects of the scene, while a sparse sampling elsewhere. Let the last iteration was $\mathbf{r}(t_i)$ in the ray space and $\mathbf{p} = \mathbf{w}^{-1}(\mathbf{r}(t_i))$ in the object space (the first iteration is the ray origin $\mathbf{o}$ and $\mathbf{w}^{-1}(\mathbf{o})$).

1. Find the distance to the object nearest from $\mathbf{p}$ and denote it as d. Any point of the curve nearer to $\mathbf{p}$ than d cannot be the intersection point.
2. So evaluate the ray samples in $t_i + \Delta, t_i + 2\Delta, \ldots, t_i + k\Delta$ until the distance of $\mathbf{w}^{-1}(\mathbf{r}(t_i + k\Delta))$ from $\mathbf{p}$ exceeds d.
3. If $k > 1$, next iteration will be in $t_{i+1} = t_i + (k-1)\Delta$ and there is no need for intersection test. If $k = 1$, there is a very near object, so test a line segment from $\mathbf{p}$ to $\mathbf{w}^{-1}(\mathbf{r}(t_i + \Delta))$ for intersection and if none was found next iteration is $t_{i+1} = t_i + \Delta$.
4. Repeat from 1. until a maximal number of samples was reached, if so, then the ray doesn't intersect anything.

See figure 4.2(right) for proximity based ray sampling.

Sampling the curved ray without the inverse spacewarp

If computing of the inverse spacewarp is very time-consuming (the case of general inversion method), more appropriate is an approach called *iterative tracing*. It is based on

[1] Actually more precise way would be calculating the angle between the line segment and the tangent to the curve which avoids some pathological cases. However the tangent obtained by forward difference requires in each sample an additional one.

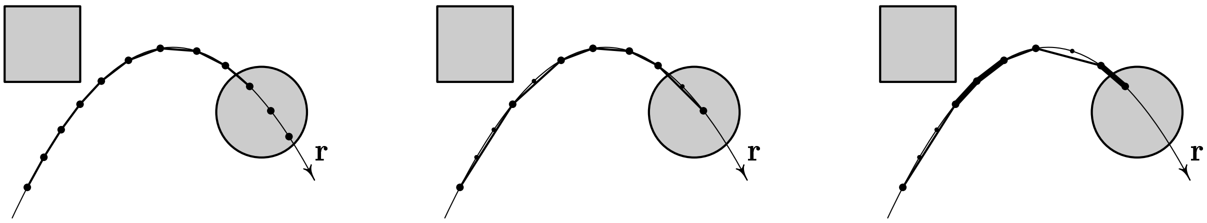


Figure 4.2. Comparison of the three sampling methods using inverse spacewarp with the same basic step. On the left the constant step sampling, in the middle curvature based adaptive sampling and on the right proximity based adaptive sampling. Note that only the thick line segments are tested for intersection in the right image.

a similar idea as the general inversion method algorithm. Let's have a ray iteration $\mathbf{r}(t_i)$ in the ray space and $\mathbf{w}^{-1}(\mathbf{r}(t_i))$ in the object space and a Δ – basic distance to the next iteration. Consider a sphere with radius Δ and center at current iteration $\mathbf{w}^{-1}(\mathbf{r}(t_i))$, next iteration is somewhere on the sphere surface, it is particularly the intersection point of the unknown curve of the ray and the sphere. How do we find it? Approximate the spacewarp by a triangular mesh and transform its vertices by spacewarp to the ray space. Here the ray is straight and the mesh approximates some "potato-like" shape. Intersection of the ray and mesh (farther along the ray than the current iteration – the center of the "potato") is located in some triangle – then in the same triangle in object space is the intersection of the curved ray and the sphere (see figure 4.3). By further subdivision of that triangle, intersection may be localized with any precision. The first iteration $\mathbf{w}^{-1}(\mathbf{r}(0))$ is evaluated by the general inversion method. Note that the general inversion method is actually an iterative tracing of the ray from $\mathbf{0}$ to the inverted point (in ray space).

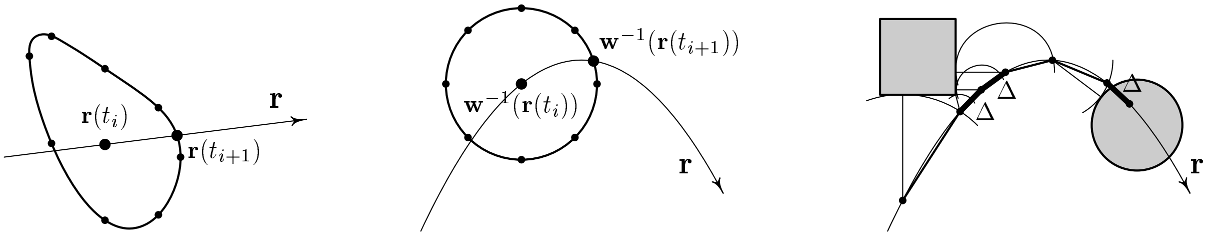


Figure 4.3. One step of iterative tracing a curved ray. The surrounding sphere in the object space (middle) is warped to a "potato-like" shape in the ray space (left). On the right is shown an example of proximity based adaptive sampling. Note that only the thick line segments are tested for intersection.

According to radius of the sphere in the individual iterations we can again differentiate methods to

constant step (uniform) sampling with constant radius Δ in all iterations,

curvature based adaptive sampling where the radius is reduced in curved regions, i.e. where angle between two consecutive iterations is too high, and

proximity based adaptive sampling, the most advanced one. Here the step to the next iteration is equal to the distance of the nearest object from current iteration. The curve surely doesn't intersect any object within that distance. Only if some object is nearer than a basic step Δ, the next iteration will be at Δ-distance and the intersection test will be performed between the line segment and objects in the scene (see figure 4.3).

Note. Note that the given basic step between consecutive iterations Δ is in case of piecewise inversion measured on the straight ray in ray space, while in iterative tracing it is on the curved ray in object space.

4.4 Illumination calculations

In chapter 3 we have derived a formula for surface normal in warped space (ray space) needed to evaluate illumination and directions of secondary rays. Described method using Jacobian matrix of the spaceewarp function bears several drawbacks: spacewarp function must be differentiable, partial derivatives are needed and finally it's quite lengthy to compute. It is very uncomfortable to require all partial derivatives from designer and their estimation by forward or central differences is a pretty lengthy job. It's fine for mathematicians, but for practical usage we propose another simpler method.

Let $\mathbf{p}$ be an intersection point where the surface normal is needed. Take two independent vectors $\mathbf{u}$, $\mathbf{v}$ tangential to the surface in object space in the point $\mathbf{p}$ and denote $\mathbf{q} = \mathbf{p} + \mathbf{u}$, $\mathbf{r} = \mathbf{p} + \mathbf{v}$.[2] Then warp all points $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$; if the $\mathbf{u}$, $\mathbf{v}$ were short enough, then the vectors $\mathbf{u}' = \mathbf{w}(\mathbf{q}) - \mathbf{w}(\mathbf{p})$ and $\mathbf{v}' = \mathbf{w}(\mathbf{r}) - \mathbf{w}(\mathbf{p})$ are close to tangents in the point $\mathbf{w}(\mathbf{p})$ in ray space.[3] Finally, normal to the surface in warped space is obtained as the cross product of tangential vectors: $\mathbf{n}' = \mathbf{u}' \times \mathbf{v}'$ and adjusted to the unit length. This way is much faster than the one through Jacobians. Continuity of the spacewarp function guarantees arbitrary precision of the warped normal if $\mathbf{u}$, $\mathbf{v}$ were chosen short enough (bounded by a floating point number precision of course).

[2] Vectors $\mathbf{u}$, $\mathbf{v}$ can be obtained as follows: Get the surface normal $\mathbf{n}$ and any vector $\mathbf{u}$ perpendicular to $\mathbf{n}$. Then $\mathbf{v} = \mathbf{n} \times \mathbf{u}$.

[3] From continuity of the spacewarp function.

Chapter 5
Implementation

A part of this work is an experimental implementation. Its objectives are verifying the correctness and feasibility of the idea and openess – possibility to modify parameters of the spacewarp and write and plug the new ones. Two options were considered – to start from scratch and develop a completely new implementation of a simple ray tracer or add the feature of curved rays to the popular freeware Persistance of Vision Ray Tracer.

A specific implementation is favorable for its simplicity and perfect fitting to the requirements. On the other hand a lot of work is needed to write the basic functionality like parsing of input file or even a graphical modeling tool, intersection routines for a variety of geometric primitives, texturing stuff, etc. which is out of the scope of this work. This is the reason why a custom version of POV-Ray has been chosen, where all those things are ready-made. Though it wasn't a painless process anyway. POV-Ray comprises a plenty of miscellaneous features and for some of them it was quite difficult to achieve compatibility with the spacewarps. Moreover POV-Ray source code developed gradually over the years is poorly structured, tangled and commented only on the lowest level. A comprehensive user documentation as well as thousands of source lines had to be analyzed.

5.1 What Is POV-Ray

Persistance of Vision Ray Tracer (POV-Ray) is a popular free rendering tool available on the internet from `http://www.povray.org`. It runs on perhaps all platforms and moreover the source code in C language can be downloaded and compiled on any exotic platform or a custom version with additional features may be built, as in our case.

One of the greatest power of POV-Ray is a wide variety of handled shapes. Besides the standard geometric primitives like sphere, box or cone it renders more complex non-trivial objects e.g. blobs, algebraic surfaces, heightfields, CSG objects or even Julia fractals. Exceptional are POV-Ray's texturing capabilities; tens of procedural texture patterns can be composed to layers, blended or applied as a normal modifier (a bump map). There is a large number of predefined impressive textures e.g. stones, metals, glasses or woods. Interesting and uncommon features of POV-Ray are atmospheric effects, media sampling used for a smoke or flame rendering or radiosity based illumination calculations. The code of POV-Ray is highly optimized, including also automatic bounding hierarchy building, light and vista buffers (see the speed up techniques of ray tracing in section 2.3)[1] yielding a pretty good performance. POV-Ray uses the left-hand coordinate system (see figure 2.1).

Our first intention was preserving the platform independence of the program. The spacewarp function is an arbitrary procedure satisfying an input/output specification and given by designer. We need to integrate it with the main code. To add it to the source code and to recompile the entire POV-Ray source is a quite inconvenient way, a new spacewarp should be plugged directly into POV-Ray executable. For this purpose the *dynamic link libraries (DLL)* on MS Windows platform have been chosen. Thus POV-Ray with spacewarps is running only under MS Windows, nevertheless porting to any platform with similar dynamic linking mechanism will be trivial.

[1] However, note that light and vista buffers from their principle cannot be used with curved rays.

5.2 POV-Ray Scene Description Language

Input of POV-Ray is a text file describing the scene composition. The scene is described in a Scene Description Language (SDL) similar to C-language syntax. Here is an example of a simple scene:

```
#include "colors.inc"
camera { location <0, 2.5, -4> direction <0, -0.3, 1> }
plane { y, 0 pigment { checker Yellow Black } }
sphere { <0, 1, 0> 1 pigment { color Blue }
                     finish { phong 0.7 phong_size 80 } }
light_source { <20, 20, -10> White }
```

This is description of a classical ray tracing scene – a shiny sphere on a checkerboard. The code is pretty well self-explanatory. Firstly a file with symbolic names of colors is included exactly like in C. Then the camera position in the space is specified and direction of the view. Some objects follows – a plane with normal in y-axis direction[2] translated by zero along its normal, with yellow-balck checkerboard and a blue sphere centered one unit above the origin with radius 1 and highly polished surface – specular reflection coefficient 0.7 and exponent 80 in Phong illumination model. Finally a point light source is placed emitting white omnidirectional light.

The POV-Ray scene description language consists of directives. The major part are directives defining objects, their geometric qualities, appearance, textures and modifiers like additional scaling, rotation or translation. A camera must be specified and usually some light sources. Another directives are for atmospheric effects, media, interior material properties or various global settings. An overall ambient light in the scene, for instance, may be set by

```
global_settings { ambient_light 0.3*White }
```

Pretty useful are higher level structuring directives. They comprise including another file, an object declaration and its multiple using, if-else conditions, while cycles handy for arrays of objects generation or a macro definition. Variables can be declared and calculated by a plenty of mathematical functions on real numbers and vectors or read from random streams. A complete reference to POV-Ray SDL can be found in documentation to POV-Ray distribution [POV] or on `http://www.povray.org`.

5.3 Spacewarp Extensions of POV-Ray SDL

Spacewarps have been implemented to POV-Ray by modification of the rendering core functions to work with curved rays, as an extension of SDL by a few new directives and a command line switch to turn on/off spacewarps feature.

The spacewarp command

A spacewarp is added to the scene by command `spacewarp`. For instance

```
spacewarp { "blackhole", 1, 1 }
```

[2] Remember that POV-Ray uses the left-hand coordinated system.

defines the black hole spacewarp with one initialization parameter – intensity of the black hole equal to 1. The black hole spacewarp has its center at origin, if you wish it was in let's say $(3, 5, 8)^T$ write

```
spacewarp { "blackhole", 1, 1
            translate <3, 5, 8> }
```

The general syntax of the `spacewarp` command is

```
spacewarp { "SPACEWARP_NAME",
            NUMBER_OF_PARAMS, [PARAMS...],
            [SPACEWARP_TRANSFORMATIONS...] }
```

where:

SPACEWARP_NAME is type of the spacewarp (the name of DLL library with corresponding spacewarp function, in fact).

NUMBER_OF_PARAMS is the number of following parameters passed to set-up the spacewarp, e.g. intensity or another constants in spacewarp formula. If the spacewarp expects no parameters, write 0.

PARAMS is a sequence of exactly NUMBER_OF_PARAMS real numbers – parameters passed to spacewarp initialization. If more parameters than needed is given, the superfluous ones are ignored. In case of less given parameters than expected, it depends on the particular spacewarp initialization function. It may report an error or use some default values.

SPACEWARP_TRANSFORMATION may be one of the `translate`, `rotate` or `scale` directive followed by a vector in form `<`a`,`b`,`c`>`. Those modifiers affect the area of spacewarp influence – translate it by a given vector, rotate around x, y and z axes by a, b and c degrees respectively and finally non-uniformly scale by a, b, c coefficients in x, y, z axes directions respectively.[3] Another ways of adding a linear transformation is by its matrix – write `matrix {` a_{00}`,` a_{01}`,` a_{02}`,` a_{10}`, ...,` a_{32} `}`, where $a_{00}, \ldots, a_{32}$ are 12 significant elements of the transformation matrix (See POV-Ray SDL documentation), or by `transform` *TRANSFORM_IDENTIFIER* directive referencing a previously declared transformation by its identifier. Any number of transformations may be given, their effects are composed in the respective order (note that the order of transformations is significant).

Note that you need to know the number of initialization parameters and their purpose in order to use a particular spacewarp. List of all predefined spacewarps, their formulas and required parameters is provided in Appendix B. If more than one `spacewarp` directive is given in the scene description, the last one will be used.

Spacewarps switch

In addition to `spacewarp` command, the switch **+SW** must be provided among the command line switches or a line

```
Spacewarp = on
```

[3] Note that in POV-Ray SDL you can write one number k whenever a vector is expected, which stands then for the vector `<`k`,` k`,` k`>`.

in the .ini file to turn the spacewarp feature on.[4] If there is **-SW** switch or `Spacewarp = off` in the .ini file, the program behaves exactly like standard POV-Ray and so does also in case that no `spacewarp` directive was found in the scene description.

Spacewarps settings

There are also some general settings and constants related to spacewarps, mentioned in chapter 4. They all are defined in the `spacewarp` block of the `global_settings` command, for instance:

```
global_settings {
  spacewarp { SPACEWARP_SETTINGS... }
  ambient_light 0.3*White
}
```

In the `spacewarp` block there are two nested blocks of settings – `tracing` and `inversion`. The full syntax of the `spacewarp` block is as follows:

```
global_settings {
  spacewarp {
    tracing {
      step s
      max_iterations m
    }
    inversion {
      tolerance t
      max_iterations n
      theta_steps h
      phi_steps p
      depth d
      epsilon e
    }
  }
}
```

where s, m, t, n, h, p, d, e are numbers. Any of the settings or an entire block may be omitted keeping a default value in corresponding constant. List of the individual settings follows, with a brief meaning of each one; for an algorithmic context see the appropriate section in chapter 4.

The `tracing` block aggregates two settings related to tracing the curved ray.

`step` (default 0.1) is the Δ constant from section 4.3, the step with the curved ray is sampled. Note that in case of iterative sampling it's just the length of basic line segment approximating the curve while in piecewise inversion method the Δ step is measured on the straight ray and inverted afterwards.

`max_iterations` (default 100) is maximal number of basic steps along the ray while searching for an intersection with the curved ray. If no intersection is found within that many steps, the ray is assumed not to intersect anything.

[4] Consequently the light and vista buffers are automatically turned off due to the principal incompatibility with curved rays.

The inversion block is dedicated to parameters affecting the general inversion method process. If an inverse spacewarp is given, the general inversion method isn't used and the entire `inversion` block takes no effect.

`tolerance` (default 0.0001) is permitted error of the inversion in the image space – inverting a point $\mathbf{w}(\mathbf{p})$ the tolerance is compared with $\|\mathbf{w}(\mathbf{p}) - \mathbf{w}(\mathbf{p}_i)\|$ for i-th iteration $\mathbf{p}_i$, not with $\|\mathbf{p} - \mathbf{p}_i\|$ in the domain space since the $\mathbf{p}$ is not known. Whenever the error is lesser than `tolerance`, the iteration process stops.

`max_iterations` (default 100) is maximal number of iterations performed during inversion process. Then iteration stops and the last one is returned.

`theta_steps`, `phi_steps` (default 5, 8) are the number of parallels and the number of samples on each parallel respectively, when sampling and triangulating the sphere surrounding current iteration. Both must be at least 3.

`depth` (default 20) is the maximal number of recursive subdivisions of the triangle in the surrounding mesh intersected by the ray.

`epsilon` (default 0.0001) – when the intersection is nearer than `epsilon` to a vertex of the triangle, subdivision process stops.

When the inversion process finishes with error not within the given tolerance, the particular ray cannot be traced properly and a warning is displayed. In this case some play with above constants is recommended. Generally it is useful to take note of the warnings reported by renderer. Finally one another useful tip: if some object in the scene (or nothing at all) is not displayed, perhaps the `max_iterations` in the `tracing` block is too low. Try to set higher value to both `max_iterations` and `step` first. Then decrease the `max_iterations` as possible yielding identical image to not waste the time. Finally if some edges or artifacts are visible due to the large `step`, decrease the `step` while increasing `max_iterations` simultaneously to preserve the overall length of the ray.

5.4 Creating New Spacewarps

From programmer's point of view the spacewarp is a procedure taking a 3-component vector and returning another one. This solution was chosen aver a mathematical formula because of providing more freedom for spacewarp computation. It can be realized as interpolation of numbers in a table or by another means besides the composition of common elementary functions in case of standard mathematical notation. And finally it's usually faster to evaluate than parsing and interpreting a formula.

Adding a new spacewarp is in fact creation of a dynamic link library (DLL) containing several important functions. What is a DLL? DLL is a file, a library containing a set of functions which other programs can make use of. If a program wants to use functions from a particular DLL it loads that DLL into memory and requests for a pointer to the desired function. All that is realized in runtime by operation system services. Then it can whenever call the function through the obtained pointer. DLLs are a very clever mechanism providing a couple of benefits: sharing the code – if several programs are using the same DLL simultaneously, the DLL is stored in memory only once, hence the programs are shorter and an additional functionality can be added to existing programs without editing the source code – just the our case.

There are four important functions in a spacewarp DLL: `init_spacewarp`, `spacewarp`, `inverse_spacewarp` and `destroy_spacewarp`, and only the `spacewarp` function is obligatory. Besides them there may be any number of auxiliary functions called by those four.

The `init_spacewarp` function is intended for initialization of the spacewarp function setting some internal variables acting in the spacewarp formula as constants, performing necessary memory allocations and may be some look-up tables creation for inverse spacewarp computation. It is called after parsing the input file, before rendering. The function prototype is:

```
int init_spacewarp(int n, double params[], char *msg)
```

where in `n` enters the number of parameters, in `params` is an array of `n` parameters read from the `spacewarp` directive in the scene description and `msg` may be pointed to an error message if initialization failed. The function must return 0 if initialization passed or nonzero if it failed, in this case the error `msg` is reported and rendering is terminated. If there is no initialization needed, the `init_spacewarp` function is not required.

The `spacewarp` function is the only one obligatory function. Its prototype is:

```
int spacewarp(VECTOR arg, VECTOR res)
```

transforming an argument vector `arg` into a result vector `res`.[5] It should return TRUE (non-zero).

The `inverse_spacewarp` function with prototype

```
int inverse_spacewarp(VECTOR arg, VECTOR res)
```

computes the inverse of the spacewarp function or its approximation. It returns TRUE if a satisfyingly precise result was achieved and FALSE (zero) if inversion failed (hence a warning is reported). If this function is not given, the general inversion method will being used instead.

Finally the `destroy_spacewarp` is a function called after rendering was finished or terminated. There is the place for various clean up actions and memory deallocations if necessary. Its prototype is

```
int destroy_spacewarp(int n)
```

Purpose of the input and return value is not defined now, maybe the future shows a reasonable usage.

So what is the complete process of a new spacewarp creation? Let's add a spacewarp called "mywarp". Open a new file `mywarp.c` and write a code for the `spacewarp` function and some of the other three above functions. On the top add

```
#include "warp.h"
```

to include some useful declarations and function prototypes. Save it to directory <POV-Ray>`/source/spacewarp/warps` (<POV-Ray> is a path to your installation of POV-Ray). You can read the file `warp.h` there and examine the source codes of predefined spacewarps to better understand and inspire yourself. Write a file `mywarp.def` containing

```
EXPORTS
  init_spacewarp
```

[5] Note that `VECTOR` is defined as `double[3]` in `warp.h`.

```
spacewarp
inverse_spacewarp
```

or so, with list of those functions from the above four, present in `mywarp.c` and save it to the same place. Then set an option like "build DLL" in your compilation tool for MS Windows and compile it. If you are using GCC compiler for MS Windows from MinGW32 package you can make use of the prepared makefile. Just edit there a line setting the `WARPNAME` variable to

```
WARPNAME = mywarp
```

and type `make all` in the command line. The `mywarp.dll` should appear. Last of all copy that DLL to <POV-Ray>`/bin/warps` directory and you can start to use the new spacewarp in the scene description files by

`spacewarp { "mywarp",` *N*, *[PARAMS...]* `}`

Finally one more note about the spacewarp modifiers `translate`, `rotate` and `scale` implementation. You needn't take care about them when writing a new spacewarp, just write it in some "unit-sized" normalized manner. The effects of all modifiers are composed into one transformation matrix $\mathbf{T}$ since the result is a linear mapping. Just before calling your `spacewarp` function coordinates of the input argument are automatically transformed by $\mathbf{T}^{-1}$. After execution of the spacewarp function the result is transformed by $\mathbf{T}$ again. The result of that is as if the spacewarp effect was transformed by $\mathbf{T}$. So is done for the `inverse_spacewarp` function but in the inverse order.

5.5 Implementation Notes

The source codes of the implementation are in the directory <POV-Ray>`/source`. There is a `spacewarp` subdirectory containing the spacewarp stuff – the files `spacewarp.c` and `spacewarp.h` and the nested `warps/` subdirectory with individual sources of predefined spacewarps. Due to a poor modularity of POV-Ray source and modification of the core of rendering, not a complete spacewarp code could be gathered into `spacewarp.c`. There is a few lines in other files as well (namely: `bbox.c`, `bbox.h`, `frame.h`, `lighting.c`, `lighting.h`, `media.c`, `optin.c`, `optin.h`, `optout.c`, `optout.h`, `parse.c`, `parse.h`, `parstxtr.c`, `point.c`, `point.h`, `povray.c`, `povray.h`, `radiosit.c`, `radiosit.h`, `render.c`, `tokenize.c`, `warps.h` and `windows/config.h`, `windows/pvengine.c`, `windows/pvengine.h`, `windows/pvengine.rc`, `windows/pvengine.def`). They are marked by tags `/* +++ giro */` at the beginning of the added or modified block and `/* +++ */` at the end ("giro" is my nickname). Studying the POV-Ray core functions which trace the rays and especially calculating illumination and textures was very hard and tedious work. It consist of tens of functions calling each other with a number of parameters forming a tangled labyrinth. A couple of special effects is integrated with the core. Some effort was put into making the spacewarps compatible with the other features, however, it hasn't being tested thoroughly and some problems cannot be excluded.

The implementation has been compiled by Minimalist's GNU compilers for Windows 9x/NT (MinGW32) distribution. It is a Windows port of legendary `gcc` originated on UNIX. GCC (GNU Compiler Collection) is a set of free compilers and development tools, a part of the large GNU[6] project for free software distributed under GNU General

[6] GNU is a recursive acronym meaning "GNU's Not Unix".

Public Licence. See `http://www.gnu.org` or `http://www.fsf.org` for more information. The MinGW32 package (C/C++/ObjectiveC/Fortran compilers, binary utilities, standard include files and libraries, MS Windows headers and import libs for Windows DLLs) may be freely downloaded from its website `http://www.mingw.org`. A reputable power of GCC compilers is producing of a highly optimized code. Unfortunately no visual environment is a part of MinGW32, compiler options are passed through command line switches. Thus an external GUI is needed to edit files with full comfort and compile with appropriate switches. There is plenty of free GUI environments with varying quality available on the internet.

Chapter 6
Extensions of the Mathematical Model

Here we propose some new features not covered by the basic mathematical model in chapters 3, 4 and not implemented as well. Some of them are just open ideas, which could be investigated in the future.

Warp of the light rays

One can ask why not the inverse spacewarp function is given directly by designer instead of the spacewarp itself, when reading about the troubles with inversion of the spacewarp function. The major reason is designer's convenience it's much more easy to imagine an intended warp of the space than its inverse. The designer has the visual result in his mind, how the matter should be stretched or squashed, not the paths of inversely warped rays. The concept is to give the desired result instead of the means how to achieve it. Another minor reason is that the spacewarp function is still used in lighting calculations. Thus the inverse spacewarp had to be inverted anyway to calculate the spacewarp value. However, it would be overall faster, since the inverse spacewarp is evaluated many times in sampling the ray, while the spacewarp is only in intersection points needed. So one way how to extend the basic model is allowing to give the inverse spacewarp – deformation of the rays instead of the spacewarp itself or in other words to give at least one of the spacewarp function and its inverse. It would be useful for visualizing some physical phenomena e.g. light bending in a strong gravitational field and its duality – straight rays and the space warped around the matter. In this case we are describing the rays trajectories, not the actual warp of the space.

Multiple spacewarps

Pretty useful feature is placing more spacewarps into the scene. The same result could be principally achieved by one sophisticated global spacewarp, but it's a very inconvenient way. In each point of the space are the influences of all spacewarps summed together and averaged. Denote the vectors from an original point $\mathbf{p}$ to its images under various spacewarps $\mathbf{w}_i(\mathbf{p})$ as $\mathbf{d}_i$, the global spacewarp will be calculated as $\mathbf{w}(\mathbf{p}) = \sum_i \mathbf{d}_i$. Note that the composed spacewarp is supposed to be bijective, it is important to check it. The general inversion method will be used most likely to invert an composite spacewarp.

Local Spacewarps

Another extension of the simplest model are local spacewarps. Two kinds of the local spacewarps can be differentiated – *space-local* spacewarps and *object-local* spacewarps.

A *space-local* spacewarp's influence is restricted to a bounded volume of the space. That restriction may be comprised in the spacewarp formula (procedure) itself, returning identity outside the region of influence. Another way is adding a `bounded_by` spacewarp modifier allowing to specify an object in the scene bounding the spacewarp effect. Only inside that object the spacewarp is applied. Consider that some problems with continuity of the spacewarp on the object's boundary may occur. Space-local spacewarps allows very efficient rendering compared to the global ones. The rays are curved and sampled only inside the bounding object; a straight ray is shot into the scene first and just in the case that the nearest intersection is with an object bounding a spacewarp the portion of the ray inside the object is sampled and curved. This approach with bounding spheres is used in [KuYa95, 96, 97].

An *object-local* spacewarp's effect is bound to a particular object or a set of objects regardless of their position in space. Formalization is an object-spacewarp binding relation $\mathcal{B} : \{spacewarps\} \times \{objects\}$ describing which spacewarp affects which object. The nearest intersection of a ray with the scene is then determined as follows: Let $\mathcal{S}$ be the set of spacewarps and let $\mathcal{O}$ be the set of objects in the scene, $\mathcal{B} : \mathcal{S} \times \mathcal{O}$ gives the bindings. Let $S_i \subseteq \mathcal{S}$ be the set of spacewarps affecting an object o_i. Tracing a particular ray $\mathbf{r}$ denote the ray curved by the inverse of the composite spacewarp S_i as $S_i^{-1}\mathbf{r}$ and let its intersection with o_i be in parameter t_i (or ∞ if there is no intersection). The nearest intersection of the ray $\mathbf{r}$ and the scene affected by object-local spacewarps $\mathcal{S}$ is the one with the smallest t_i.

In rendering the object local spacewarps as many curved rays is traced for each pixel in parallel as distinct subsets of $\mathcal{S}$ assigned to an object there is. Intersections of individual curved rays are tested only with the objects affected by the same subset of spacewarps as the ray is.

Alternative definition of the spacewarp

Even the definition of the spacewarp itself may be considered. Some "hyperspace tunnels" could be modeled by a discontinuous spacewarp. Or a non-bijective spacewarp function makes squashing of the whole space to a finite volume possible or the effects like knotting the space. Continuity and bijectivity are however essential assumptions in the presented algorithms. To omit those two conditions a very clever trick is needed.

Animation

The spacewarps may also be animated. Animation is a change of any parameters or qualities in the time. The most often animated qualities are position of the objects, their shape or material. The parameters of the spacewarp may be animated as well. There is a special variable `clock` available in the scene description, running from 0 in the first frame of an animation up to 1 in the last frame; for each frame is the input file parsed again. All issues of animation in POV-Ray are realized through the `clock` variable (see [POV] for details) and it can be used among the spacewarp parameters alike, since the spacewarp is initialized after parsing each frame again. Spacewarps influencing only in an interval of time can be achieved too in this way, if designed with this purpose in mind and an appropriate parameter is provided. Animation is the only implemented issue from this chapter.

From an intuitive image that the spacewarp "attracts" or "repels" the objects, one can suggest doing the animation as iteration of the spacewarp function – a point $\mathbf{p}$ is transformed to $\mathbf{w}(\mathbf{p})$ in the first frame, to $\mathbf{w}(\mathbf{w}(\mathbf{p}))$ in the second, to $\mathbf{w}(\mathbf{w}(\mathbf{w}(\mathbf{p})))$ in the third, etc. That is, however a fundamental misunderstanding. Note that the objects are *not* forced to move by spacewarps, there are no forces. The objects are still in the warped space. just the parameters (e.g. intensity) of the spacewarp are varying. On the other hand a simultaneous animation of either object positions and spacewarp parameters is not excluded (e.g. a black hole attracting the objects around can be simulated this way).

Chapter 7
Conclusion, Results and Visions

We have proposed an unconventional approach for rendering warps of the space by ray tracing with curved rays. Bending the sight rays by inverse warp evokes an illusion of deformed space. We have derived analytic representations of objects in warped space, their intersection with straight rays and we have showed its duality with intersection of a curved ray and the non-deformed objects in common Euclidean space. Illumination in the warped space has been also covered in the mathematical model. We have discussed a couple of solutions to realize the proposed model by computer. Tracing of the curved ray through the space has been investigated as well as the inverse spacewarp calculation and a general inversion algorithm has been developed – a heuristic method for calculation of the inverse to a multi-dimensional function in a given point. Correctness and feasibility of the approach is confirmed by an experimental implementation based on the free renderer POV-Ray, where selected methods have been implemented. Finally some possible extensions of the basic model and further ideas are discussed.

Comparing to the common approach requiring tesselation of the scene, the proposed method brings several advantages. Perhaps all used representations can be rendered by ray tracing and hence deformed by our method. There is no need of triangulation yielding a huge number of faces. Moreover only the visible part of the scene is actually deformed. Note that tracing a curved ray it is also sampled, in fact, and aliasing problems can occur like in tesselation. However, when precision is increased by denser sampling, amount of triangles after tesselation is raising much more faster than number of line segments approximating the curved ray and hence the rendering time behaves in the same manner. The number of samples on the curved rays is actually independent of the scene complexity; if the scene is larger a longer portion of the ray is needed to sample, but the sampling can be sparser and vice versa. It depends on resolution of the output image. A clever idea is denser sampling near the viewpoint while sparser sampling in farther regions.

On the other hand a drawback of our approach is in computational time. But although ray tracing of a warped scene is several-fold slower than ray tracing the original one, the rendering times are still reasonable. It is strongly dependent on resolution of the output image and computer performance, but generally rendering a simple scene takes minutes, a complex one can take hours and if specific effects are used requiring to trace a great number of additional rays, e.g. area lights or focal blur, the rendering can take even more. Some images of deformed scenes rendered by our implementation are presented at the end of this chapter.[1] The respective rendering times (on standard PC 500MHZ) are mentioned in descriptions below the images. A positive fact is that although relatively high constant factor, rendering times increases with logarithm of the scene complexity due to hierarchical bounding structures used by POV-Ray. And finally the POV-Ray based implementation is not optimal (for curved rays is more suitable space subdivision than bounding hierarchies).

A weak point of our implementation is the general inversion method, used if the inverse spacewarp was not given by designer. Though its reliability is about 99% it is still insufficient and there are visible artifacts degrading the image quality. The method (or

[1] For color version of the images see `/Results` directory in enclosed CD.

its implementation) needs further improvement.

To give a spacewarp as a mathematical function requires a pretty good imagination and mathematical background. For designer's convenience it would be useful to have a visual tool for designing spacewarps like in image warping is common. Local spacewarps can be used as deformations to render soft objects (e.g. a depression on the soft bed under heavy ball) or as more faithful substitution of bump mapping (maybe a spacewarp derived from bitmap).

Maybe a new trend in computer graphics will arise in next years, characteristic by tighter cooperation of the renderer and the scene, called "render-time modeling". Rendering of extremely complex environments like hair, grass or vegetation is still not solved satisfyingly by present-day approach – create a detailed scene with millions of objects and passively render it afterwards. Instead of that only the simple geometric parts of the scene could be modeled analytically and the complex objects passed to renderer as a procedure or parameters describing the way how to render it. It is like procedural modeling just in render-time, from that the term "render-time modeling". There is no intermediate representation it is directly rendered. It is specially suitable for natural phenomena where stochastic implicit functions or fractal techniques could be used. Just model each object in the right way – either in the scene or by the renderer on the fly, where is it easier, yielding the best quality image. Spacewarps in our thesis are just an early experiments with the render-time modeling.

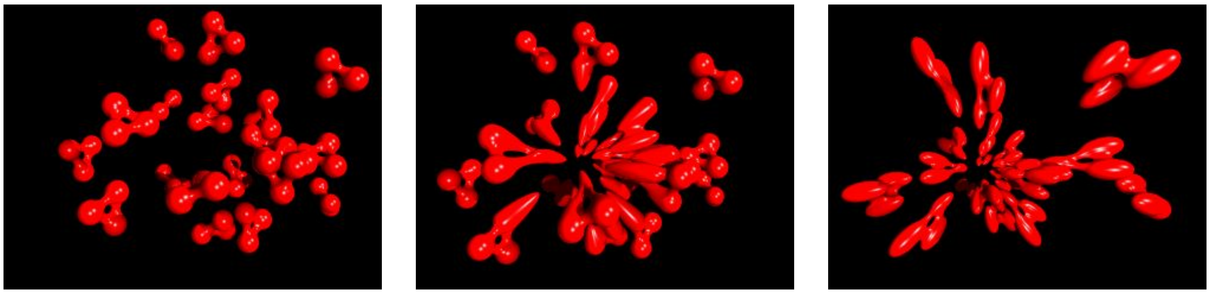


Figure 7.1. A cluster of blobs in bounded black hole spacewarp. In the middle only a part of the scene is affected, while on the right the entire cluster is within the effect region. the lighting crystal there is a halo effect.
[Rendering times ($320 \times 240\,aa$) 13s vs. 55s and 46s.]

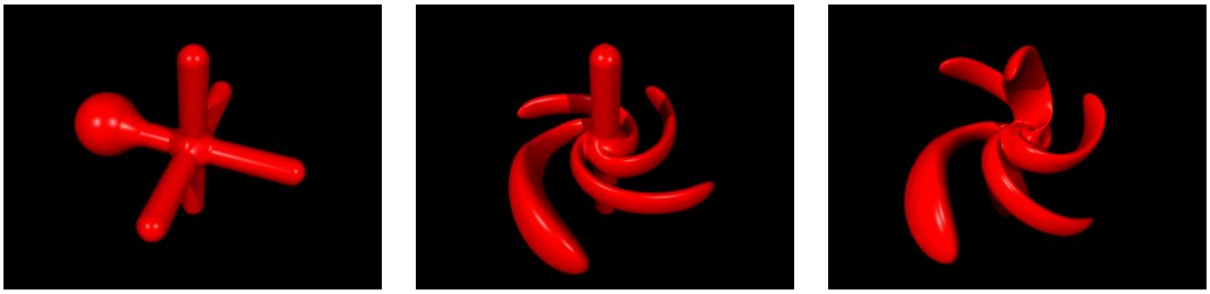


Figure 7.2. Another blob in planar whirl spacewarp. The planar whirl acts in the horizontal plane, the interesting shape in the right image has been achieved by rotating the whirl plane by 30 degrees around the z-axis.
[Rendering times ($320 \times 240\,aa$) 5s vs. 1m 35s and 1m 25s.]

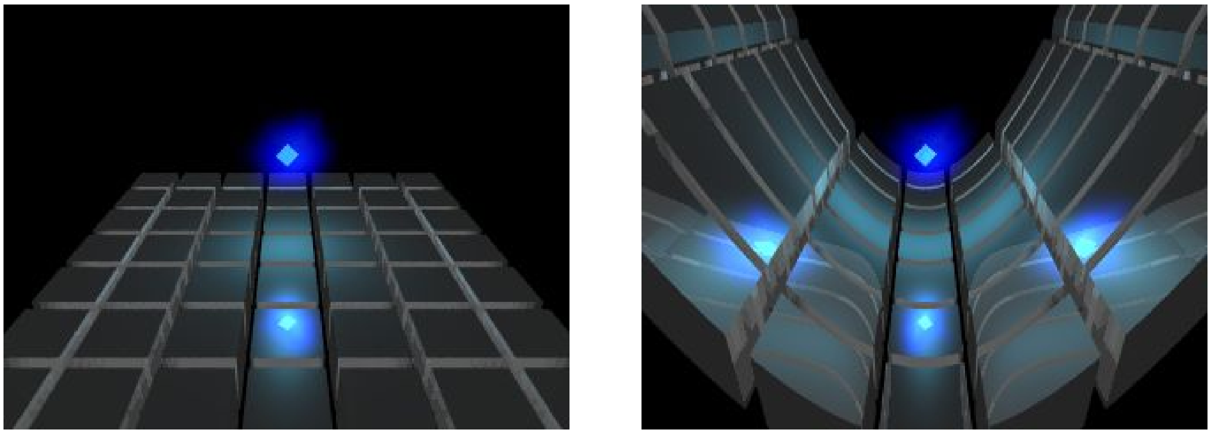


Figure 7.3. An array of chrome cube in parabolic spacewarp. Note the precise illumination and the individual wings reflecting each other. Arround the lighting crystal there is a halo effect.
[Rendering times (320×240, *antialiasing*) 17s vs. 5m 26s.]

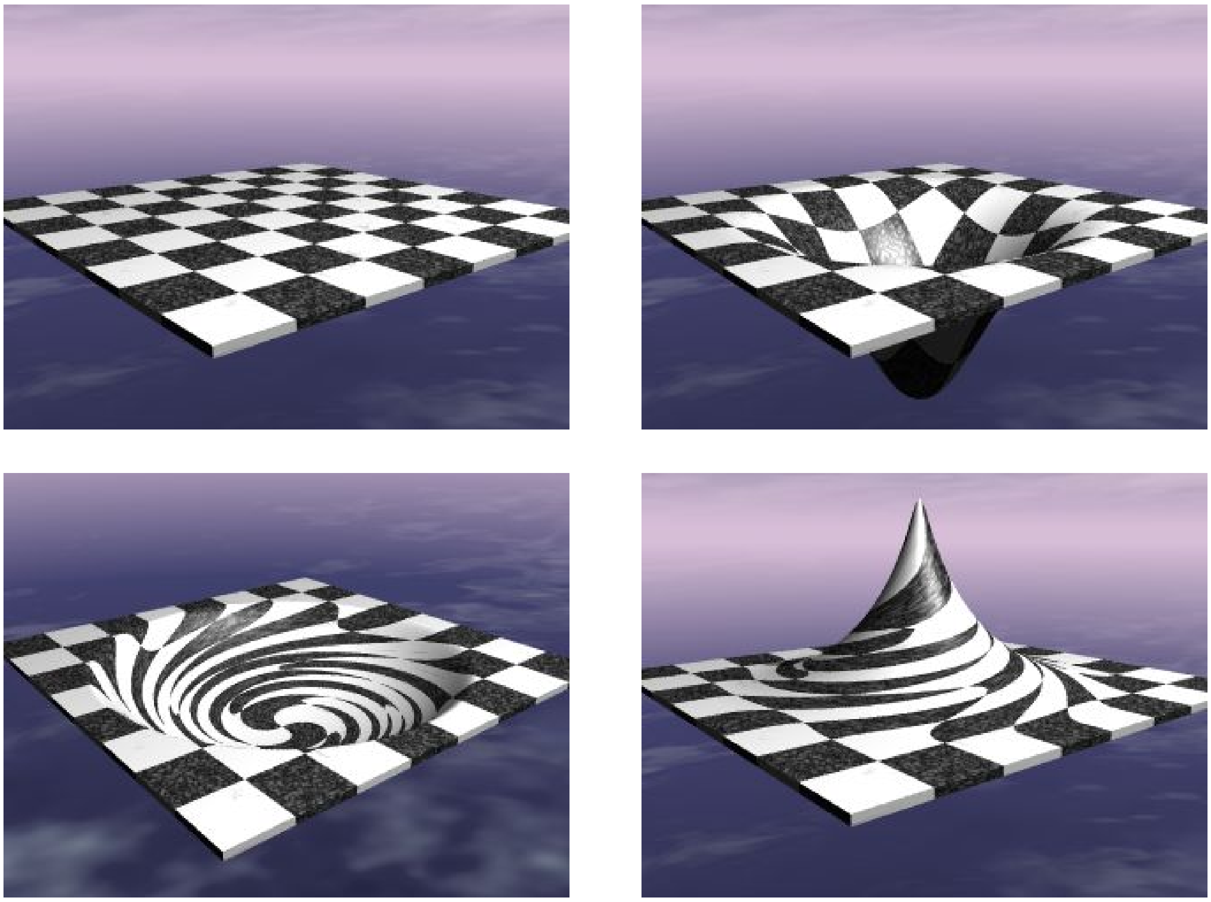


Figure 7.4. Variations of the whirl spacewarp demonstrated on the flying chessboard. The last whirl was upturned by negative scaling in vertical y-axis. [Rendering times ($320 \times 240\,aa$) 9s vs. 1m 17s, 1m 51s, 1m 19s respectively.]

Figure 7.5. "A premature end." The standard POV-Ray chess scene with the whirl spacewarp. [Rendering times ($320 \times 240\, aa$) 1m 20s vs. 1h 13m.]

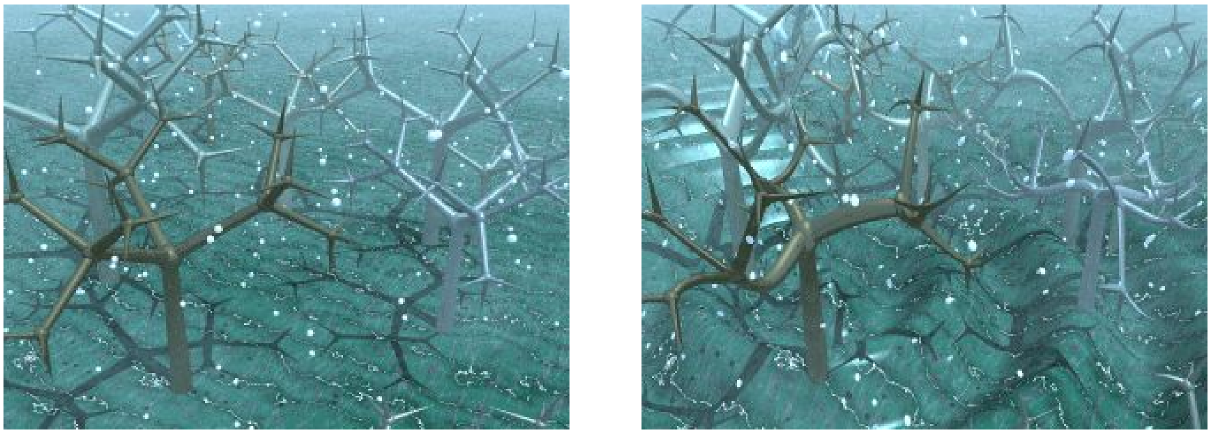


Figure 7.6. "Revival of the Sunken Forest." The ripples spacewarp applied to the forest scene. The underwater look has been achieved by a blue fog. [Rendering times ($320 \times 240\, aa$) 1m 40s vs. 21m 26s.]

References

[Barr84] BARR, A. H. Global and Local Deformations of Solid Primitives. *Proceedings of SIGGRAPH'84, Computer Graphics 18(3)*. pp. 21-30. July 1984.

[Barr86] BARR, A. H. Ray Tracing Deformed Surfaces. *Proceedings of SIGGRAPH'86, Computer Graphics 20(4)*. pp. 287-296. August 1986.

[Fole95] FOLEY, J. D. – VAN DAM, A. – FEINER, S. K. – HUGHES, J. F. *Computer Graphics – Principles and Practice, Second edition in C*. Addison-Wesley. 1995. ISBN 02-0184-8466.

[Gröl95] GRÖLLER, E. Nonlinear Ray Tracing: Visualizing Strange Worlds. *The Visual Computer 11(5)*. pp. 263-274. Springer Verlag. 1995. ISSN 0178-2789.
`http://www.cg.tuwien.ac.at/research/rendering/nlinrt/index.html`

[Kaji83] KAJIYA, J. T. New Techniques for Ray Tracing Procedurally Defined Objects. *Proceedings of SIGGRAPH '83, Computer Graphics 17(3)*. pp. 91-102. July 1983.

[KuYa95] KURZION, Y. – YAGEL, R. Space Deformation Using Ray Deflectors. *Rendering Techniques '95 (Proceedings of the 6th Eurographics Workshop on Rendering)*. pp. 21-32. New York: Springer Verlag. 1995. ISBN 3-211-82733-1.
`http://reality.sgi.com/yair_engr/papers/deflect_eg95.ps.gz`

[KuYa96] KURZION, Y. – YAGEL, R. Continuous and Discontinuous Deformation using Ray Deflectors. *Proceedings of GRAPHICON '96, vol. 2*. pp. 102-110. Saint Petersburg, Russia: July 1996.
`http://reality.sgi.com/yair_engr/papers/graphicon96.ps.gz`

[KuYa97] KURZION, Y. – YAGEL, R. Interactive Space Deformation with Hardware-Assisted Rendering. *IEEE Computer Graphics & Applications 17(5)*. pp. 66-77. September/October 1997. ISSN 0272-1716.
`http://reality.sgi.com/yair_engr/papers/cgna.ps.gz`

[MöHa99] MÖLLER T. – HAINES, E. *Real-Time Rendering*. Massachuttes: AK Peters, Ltd-Natick. 1999. ISBN 1-56881-101-2.
`http://www.acm.org/tog/resources/RTR/`

[Neyr95] NEYRET, F. Local Illumination in Deformed Space. *Technical report RR-2856, INRIA*. 1995.
`http://www.syntim.inria.fr/syntim/textes/RR-2856-eng.html`

[POV] POV-Team. POV-Ray Scene Description Language. *User's Documentation to POV-Ray 3.1g*. May 1999.
`http://www.povray.org`

[PR] The Mathematical Art of M. C. Escher. *Math Academy Online^{T}M/Platonic Realms^{T}M*.
`http://www.mathacademy.com/pr/minitext/escher/index.asp`

[Rals78] RALSTON, A. *Základy numerické matematiky*. Praha: Academia. 1978. English original: *A first course in numerical analysis, 2nd ed.* New York: McGraw Hill, Inc. 1978. ISBN 0-07-051158-6.

[Wijk84] VAN WIJK, J. J. Ray Tracing Objects Defined by Sweeping Planar Cubic Splines. *ACM Transactions on Graphics 3(3)*. pp. 223-237. July 1984. ISSN 0730-0301.

[Žára98] ŽÁRA J. – BENEŠ, B. – FELKEL, P. *Moderní počítačová grafika*. Praha: Computer Press. 1998. ISBN 80-7226-049-9.

Appendix A
Contents of the CD and Installation

A CD disk with the original POV-Ray distribution, our implementation, the thesis in digital form and some results is enclosed with this thesis. Its contents is as follows:

```
/readme.txt
/POV-Ray/
   povwin3.exe
   Spacewarp/
      bin/povsweng.exe
         warps/
      scenes/spacewarps/
      source/
/Thesis/
   spacewarps.ps
   spacewarps.pdf
/Results
```

To install POV-Ray for Windows run `povwin3.exe`. Then copy the contents of the `Spacewarp/` subdirectory to your main POV-Ray installation directory and you can start using POV-Ray with Spacewarps. The original POV-Ray is started by `bin/pvengine.exe`, for POV-Ray with Spacewarps run `bin/povsweng.exe`, in both cases a GUI interface is opened.

The directory `scenes/spacewarps/` contains the scene description files for the images presented in the thesis. In `source/` there are the source codes of our implementation written in C language and the makefiles for various compilers.

The `Thesis/` directory contains the full text of this thesis with color images in Postscript format (`spacewarps.ps`) and Portable Document Format (`spacewarps.pdf`). To view documents in Poscript or PDF use viewers GhostView or Acrobat Reader respectively, available on the internet for free.

In `Results/` directory there are some images produced by our implementation, in JPEG and PNG format.

Appendix B
Predefined Spacewarps

The list of all predefined spacewarps follows here. The respective DLL to each spacewarp is in <POV-Ray>`/bin/warps/` and its C source code in <POV-Ray>`/source/spacewarp/warps/` with the same name. For each spacewarp here is its formula, list of parameters and their default values and a brief description. Note that the order of parameters is significant, the numbers given in the `spacewarp` command will be passed to parameters in the order they are listed here. Remaining parameters take their default values. Parameters in the spacewarp formulas are typeset by ***boldface italics***.

identity

$$identity \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

0 parameters

Description: The identity spacewarp is just for completeness and testing purposes.

parabolic

$$parabolic \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y + \boldsymbol{a} x^{\boldsymbol{e}} + \boldsymbol{b} z^{\boldsymbol{f}} \\ z \end{pmatrix}$$

4 parameters: a, b, e, f (defaults: 1, 1, 2, 2)

Description: Generalized parabolic spacewarp raising in y-direction (it can be changed by `rotate` modifier). For instance parameters 1, 1, 2, 2 yields paraboloid, parameters $\frac{1}{2}$, 0, 2, 2 yields parabolic spacewarp from example 3.1.

blackhole

$$blackhole \begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1 - e^{-\boldsymbol{k} d}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

1 parameter: $k > 0$ (default: 1)

Description: The black hole spacewarp from example 3.2, attracting its surroundings, with an unbounded area of influence. The k parameter controls the intensity of the black hole, see figure B.1(left). Together with the white hole they are the only predefined spacewarps without a given inverse function thus they illustrate employment of the general inversion method leaving visible artifacts in the image.

whitehole

$$whitehole \begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1 + e^{-\boldsymbol{k} d}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

1 parameter: $k > 0$ (default: 1)

Description: An opposite to the black hole above, the white hole spacewarp repels its surroundings. Parameter k is the intensity of the white hole alike. Note that the white hole spacewarp is not inverse to the black hole, i.e. $\mathit{whitehole}(\mathit{blackhole}(\mathbf{p})) \neq \mathbf{p}$.

boundedbhole

$$\mathit{boundedbhole}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1 - f(d))\begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \text{where}$$

$d = \sqrt{x^2 + y^2 + z^2}$
f is an intensity function given by a cubic Bezier curve (see figure B.1(middle)

$$f(d) = \begin{cases} y\text{-component of the curve respective to } x\text{-component equal to } d, & d < 1 \\ 0, & d \geq 1 \end{cases}$$

The cubic Bezier curve is given by

$$\mathbf{P}(t) = \sum_{i=0}^{3} \mathbf{V}_i B_i^3(t), \quad \text{where}$$

$$B_i^3(t) = \binom{3}{i} t^i (1-t)^{3-i} \quad \text{are Bernstein polynomials and}$$

$$\mathbf{V}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \mathbf{V}_1 = \begin{pmatrix} \boldsymbol{x_1} \\ 1 \end{pmatrix}, \quad \mathbf{V}_2 = \begin{pmatrix} \boldsymbol{x_2} \\ 0 \end{pmatrix}, \quad \mathbf{V}_3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

2 parameters: $0 \leq x_1 \leq x_2 \leq 1$ (default: $\frac{1}{3}$, $\frac{2}{3}$)

Description: The bounded black hole is more practical and flexible than the above black hole spacewarp. Parameters x_1 and x_2 controls the intensity function of the black hole the points farther than 1 from b.b.h. center are not affected. To change this radius use `scale` modifier.

boundedwhole

$$\mathit{boundedwhole}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = (1 + f(d))\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

See boundedbhole spacewarp for d and f definition.

2 parameters: $0 \leq x_1 \leq x_2 \leq 1$ (default: $\frac{1}{3}$, $\frac{2}{3}$)

Description: The bounded white hole is an opposite to above bounded black hole spacewarp (again not its inverse). Parameters $x1$, $x2$ have the same meaning.

ripples

$$\mathit{ripples}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y + \cos(\sqrt{x^2 + z^2} + \boldsymbol{p}) \\ z \end{pmatrix}$$

1 parameter: p (default: 0)

Description: The ripples spacewarp produces concentric waves in horizontal plane (may be rotated) like a fallen drop leaves on the water surface. Parameter p is a “phase” in cosine function (given in degrees), it can be used to animate the waves.

planewhirl

$$planewhirl \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x\cos\phi - z\sin\phi \\ y \\ x\sin\phi + z\cos\phi \end{pmatrix}, \quad \text{where}$$

ϕ is derived from the logarithmic spiral equation $r = e^{\phi \cot \boldsymbol{a}}$ as $\phi = \frac{\ln r}{\cot \boldsymbol{a}}$, $\quad r = \sqrt{x^2 + z^2}$

1 parameter: a (default: 60)

Description: A whirl-like spacewarp in horizontal plane (may be rotated) derived from the logarithmic spiral (see figure B.1(right)). Parameter a is the angle (in degrees) of radius and the spiral's tangent and it controls the spirals growing.

whirl

$$whirl \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x\cos\phi - z\sin\phi \\ y - f_p(d) \\ x\sin\phi + z\cos\phi \end{pmatrix}, \quad \text{where}$$

$d = \sqrt{x^2 + z^2}$
$\phi = \boldsymbol{k} f_r(d)$
f_p, f_r are functions for the profile of the whirl and amount of rotation based on Bezier curve (see above) with parameters px_1, px_2 and rx_1, rx_2 respectively.

5 parameters: $0 \le px_1 \le px_2 \le 1, \quad 0 \le rx_1 \le rx_2 \le 1$ (default: 180, $\frac{1}{3}$, $\frac{2}{3}$, $\frac{1}{3}$, $\frac{2}{3}$)

Description: The horizontal whirl spacewarp with effective radius and depth equal to 1 (may be scaled). The vertical profile of the whirl is controlled by Bezier curve with px_1, px_2 parameters (see figure B.1(middle)) and the ammount of rotation depending on the distance from whirl center alike, with parameters rx_1 and rx_2. Parameter k is an intensity of the twist (in degrees).

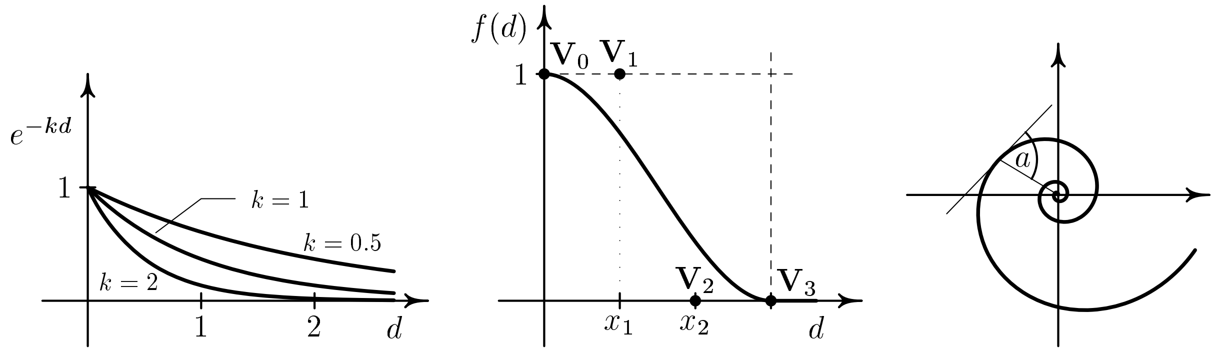


Figure B.1. Graph of the black hole intensity function for various parameters k, on the left. Intensity function controlled by Bezier curve in the middle. The logarithmic spiral on the right.