Rendering diffuse objects using particle systems inside voxelized surface geometry

Thorsten Juckel   Steffi Beckhaus
University of Hamburg
Vogt-Koelln-Strasse 30
22527 Hamburg, Germany
juckel@blurredvision.de
steffi.beckhaus@uni-hamburg.de

ABSTRACT

This paper presents an unique method for rendering complex shapes as fuzzy or diffuse objects inside virtual environments. It uses surface geometry that is converted into a voxel-like grid to specify the appearance of the shape. A particle system displays the outline of the object at runtime. Particles are allowed to move freely inside the voxel-grid but obtain certain attributes from the voxels they currently reside in. Those attributes assign color, size, textures, transparency to the particles, as well as forces that influence the movement of the particles. Force effects include linear and spiral movements, gravitational points, and helix-shaped motion of particles. Through this, complex movement of the particles inside the voxel-space can be created at interactive rates, while still maintaining the approximate form of the original surface geometry.

Keywords: Particle systems, voxels, interaction, real-time rendering

1 INTRODUCTION

Rendering diffuse changing shapes like fire, dust, or smoke is a common task in computer graphics. Since such shapes have a limited complexity, rendering them can be done efficiently in real-time using simplified models and techniques. When extending those shapes to more complex models, such as the rendering of ghost-like figures, this is no longer easily performed. Our aim is the rendering of diffuse, fuzzy, potentially ghost-like objects and characters. Like classic polygonal 3D objects, they have a specific shape and texture, but react to collisions, for example, by floating around the colliding solid object and re-assembling their shape afterwards. Waving inside the ghost-like character would result in stirring up the colors and rough shape of the character, as happens when color drops in water are stirred. Once left alone, however, the character should gradually re-adjust itself to its original form. If the character moves or changes shape, this may have a temporary blurring effect or may happen immediately. Furthermore, forces inside and around the character could make the character itself include dynamic components, such that, for example, the eyes glow by emitting fire or light particles moving in form of tornados.

The application of such characters is best exemplified by 3D ghost-like or cartoon characters in 3D games that behave like normal figures, but additionally show the described fuzzy and dynamic visual and shape behavior. Also, virtual narrators and guides in 3D environments and storytelling applications could have these features. They are often superimposed onto the running story and, as they are superimposed external "add-on"s, may potentially be more believable inside a story if they show a more fantasy-like appearance.

The idea is not bound to characters but can also comprise the visualization of normally invisible phenomena like energy fields in open space and aura-like phenomena around characters.

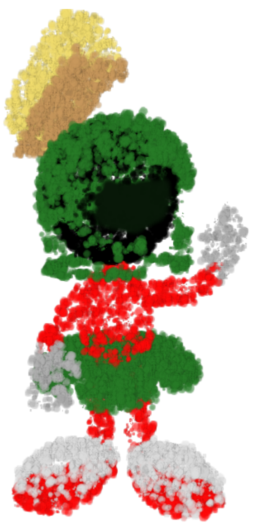

Figure 1: Marvin the Martian rendered as a diffuse character with point-sprite particles.

To visualize characters with the described features and behavior using traditional polygonal representations of objects, complex models are needed. Those tend not to act well in interactive applications such as real-time interactive virtual environments or games. However, a simplified approach can be found using voxelized geometry in combination with common particle systems.

This paper introduces a unique method for rendering diffuse figures that extends the normal use of particle systems to display fuzzy characters and complex shapes (see Figure 1). It uses a representation of surface geometry transformed into the voxel-space to approximate the outline and appearance of the shape. This voxelized grid is generated in advance as a computational step. Particles are later generated inside this space to visualize the diffuse shape in real-time. Additional attributes can be stored inside the voxels to obtain a more complex behavioral model for the fuzzy shape. Currently, size, color, animation speed, and some sophisticated movement models can be used inside the voxels.

In Sections 2 and 3, we will give an overview of the work previously done in this area. A short introduction on particle systems and also on methods used to create a voxel-space model of surface geometry is given. This information is applied later in the paper to give a detailed description of the particle system used. Furthermore, the voxelization algorithm is explained in detail. In Section 4, the implementation of the method is presented. We describe the changes made to the original voxelization algorithm to obtain further information about the original surface geometry. We also describe, how the particle system is attached to the voxel-space and how the attributes of the voxels influence the particles. That information is later extended in Section 5 to include common interaction techniques. Specifically, manipulation practices, complex forces inside the voxels, and collision with solid models will be discussed. Section 6 presents the results of the particle shape and Section 7 concludes the work with an outlook on further work.

2 RELATED WORK

To render atmospheric effects, such as dust, smoke, or fire, the number of approaches is numerous. Realistic rendering of dust has been developed by Blinn [2] to display dusty surfaces and Dobashi et al. [3] introduced a hardware-accelerated method for rendering atmospheric scattering. While those approximations produce extremely good results, they are hardly capable of real-time rendering inside large interactive environments that are often present in 3D virtual environments. They furthermore do not support direct interaction with the effect itself.

In 1983, Reeves introduced *particle systems* to render fuzzy objects like fire, explosions and forests [7]. The complex phenomena were separated into tiny parts that could individually move around given some common constraints. This concept was later converted to use parallel computations [9] and also used by Reynolds inside behavioral systems to model the complex movement of flocks of birds [8]. Particles were not only seen as individual points anymore, but could interact with each other and form a much more complex behavior. Using constraints for the particle system, like attaching the particles to vertex elements, fairly complex shapes can be achieved.

Oriented particles [11] can be attached to surface geometry or be used for point-based rendering or sometimes to display clothing. Here, particles are interconnected by springs to make up a piece of cloth. Simple dynamic computations are performed to simulate the movement of the cloth. Since the tessellation can be varied in the cloth simulation, this method is also capable of running in real-time. Today, particle systems have advanced to be the common way for displaying fuzzy phenomena like smoke, dust, or fire. Unfortunately, those phenomena are often limited in their behavior and complexity, when rendering them in real-time. Current systems, like the one Kipfer and Segal introduced, are capable of rendering up to a million particles in real-time [4]. Even though a mass of particles are computed and rendered, interaction with the particles and control over them is very limited. Other uses of particles include point-based rendering that very closely defines the shell of a geometric object but act as another approach for surface modeling and accelerated rendering.

Since we are looking for dynamic movement of the surface and close interaction with the particles, normal particle systems are sufficient. Particle systems are also used as the integral part of fluid simulations using the Navier-Stokes equation as proposed by Stam [10]. Here, particles are moved through a computational grid to represent the density of a simulated fluid. Even though particles are used there, they only contribute to the calculation of the fluid dynamics and not directly to the visual appearance.

An image-based method to represent three dimensional objects is the use of voxels. Individual pixels are extended into 3D as a grid of small equidistant boxes to build the shape of an object. This method is often used in interactive visualization of medical data, but can also be used in other fields of real-time computer graphics. Mostly, it is used to represent three-dimensional images of scanned data.

There are several methods to convert image data to surface geometry, with the most prominent being the marching cubes algorithm [5]. Oomes, Snoeren, and Dijkstra looked at the other direction [6], trying to create voxel objects from surface geometry. While their method yields good results, the cost of the computation is high. Since that would have to be done every time the triangle model is updated, this becomes impracticable for real-time animations. Another method, using todays standard graphics hardware, can be used to approximate surface geometry using voxels. Beckhaus et al. used a method to render whole scenes as slices to create a voxel-based representation of the scene used for spacial analysis and collision detection [1].

3 BACKGROUND AND DESIGN

As particle systems and voxelization are the basic techniques from which our method draws, both are introduced here in more detail.

3.1 Particle systems

Most objects in a virtual environment can easily be created as surface geometry and animated using keyframes, functions, or other methods of direct manipulation. Some elements, though, that are common in the real world, are too complex to animate and to display manually. Those include the rendering of rain, where it is not possible to create every single raindrop in the scene. However, since most raindrops exhibit the same behavioral model, it is efficient to just define one object that is generating all the raindrops inside the scene automatically.

Particle system are often used in real-time graphics for this kind of common special effects, because they are very easy to implement and have the visual consistency needed by the programmers or animators. They consist mainly of an emitter, that manages the creation and update cycle of the individual particles. For every time-step in the simulation, the emitter has to perform a couple of tasks to keep the particle system alive. First, old particles that have exceeded their lifetime are extinguished. Then, new particles are generated by the emitter and initial attributes are assigned. The current set of living particles is moved and transformed according to the rules defined inside the particle system.

In the particles themselves, individual attributes are stored. In Reeves' original work, those included color, size, lifetime, and transparency. Additionally, a position and velocity were stored to determine the movement of the particles. As individual particles reach their predetermined lifetime, they fade away and can be deleted by the emitter, since they do not contribute to the simulation any further. During their lifetime, they can be influenced by outside forces, such as wind or gravity, but also other, more direct, forces, like the collision with solid objects.

When rendering the particles, they are often represented as tiny primitives, such as points, lines or small polygons. Later, their representation was extended to use textured billboarded quads or point sprites. Even animated sprites or video-textures can be used.

3.2 Voxelization

Using several slices of image scans through a human body, a three dimensional image can be created. Commonly used in the medical field, voxels are volume elements of a certain size, that are defined by their color or opacity. The information stored in the image scans can be used to obtain a volume representation of a model. Voxels are not limited to store just color information inside them. They can, for example, be easily extended to include velocity information about movement to form a vector field. They can also be used to include all kinds of attributes. We decided to store all the visual characteristics that were discussed earlier as well as information about the forces influencing the particles.

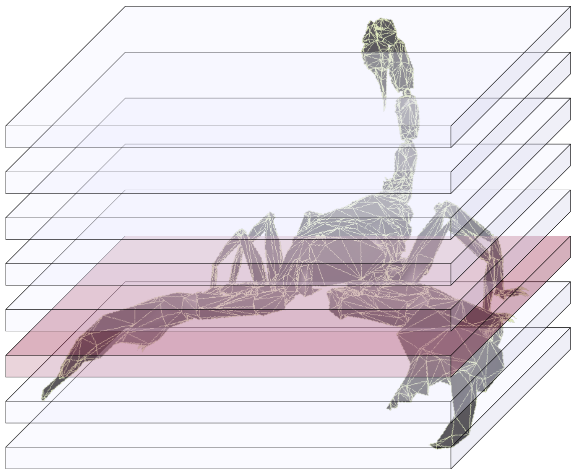

Figure 2: Several render passes through the model create the voxel representation of the shape

With todays standard graphics hardware, a special method can be used to approximate surface geometry using voxels. [1] used a method to render whole scenes as slices to create a voxel representation used for spacial analysis and collision detection. While they used this method to create a voxel representation of a complete scene in a virtual environment, it can also be applied to create texture layers from a single model.

The Marching Cubes algorithm took into account that a triangle, passing through a set of four voxels is always coloring the voxel the same way. Accordingly, a small set of triangles can be defined for different coloring configurations of the voxel-space. Using this information, creation of voxel-based images is similar. At positions where an object, composed of triangles, is inside the viewing plane, the intersection points are colored by the rasterizer unit of the graphics card. Using several viewing planes that are closely fitted behind each other, a whole set of layers can be created (see Figure 2). These layers are then used to generate a voxel representation of the surface geometry that can be used to determine the particles appearance.

4 IMPLEMENTATION

The presented method combines the use of voxelized objects with the benefits of displaying diffuse sets of particle clouds to create complex looking objects or characters in virtual environments. The focus of the design was more on the simulation and interaction process and not on the efficiency of the rendering step, even though different methods were considered or implemented.

4.1 System design

The system is made up of two layers (see Figure 3). In the representation layer, the voxelized version of the surface geometry that was previously created is used. Voxels are either active, or deactivated. If they are activated, the voxels are part of the original surface geometry outline. Those voxels contribute directly to the particle system. Deactivated voxels do not contribute to the particle system in any way. Inside each activated voxel, a set of attributes is stored that defines the behavior and appearance of the particles currently residing inside it. The voxel-space is generated automatically as an early computational step and can later be edited as needed by the animator or designer of the model.

The second layer contains the particle-emitter that is generating new particles inside the voxel-space during runtime. Even though the voxel-space is defined by the whole bounding box of the original surface geometry, particles are only created in activated voxels, i.e. voxels that have color and attributes assigned to them. They define the outline of the surface geometry.

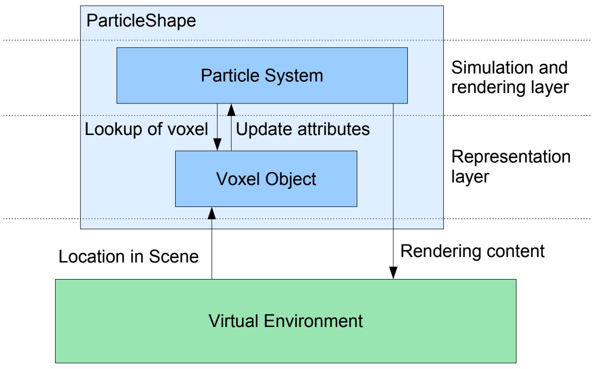

Figure 3: The layout of the system.

4.2 Voxelization

To use the voxelization algorithm of Beckhaus et al. [1] for our purposes, their method had to be slightly modified. Since they needed only spatial information about the geometry, the color information was ignored. We use the color information to store our initial attributes inside the voxel. Since this color can also be the same as the color of the back buffer, we take into consideration values stored inside the depth buffer after the rendering pass. This saves a lot of time in terms of coloring the model afterwards, since this can be done as part of the modeling step. Other attribute values stored inside the voxel are current density, maximum density, force effect, particle color, particle size, particle fading color, particle fading size, particle lifetime, particle interpolation time, animation speed for textures, and a flag for indicating, if the voxel is active, i.e. contributing to the appearance of the shape. This flag can be used during collision detection and response to make the shape move fluently through solid objects. By temporarily deactivating the voxels, particles are not created inside a solid object the particle-based shape is moving through.

Every voxel can have its own set of attributes and manipulates only the particles currently residing inside the voxel. As particles move freely in world space, they check at every time-step, Δt, which voxel they currently reside in and receive updated attribute values from the voxel accordingly. Since the voxels are evenly spaced inside the object, this can be done very fast in one simple computational step for every particle.

4.3 Update cycle

New particles are generated by an emitter inside voxels that have not reached their maximum density yet. Unlike normal particle systems, particles receive their initial values not from that emitter, but from the voxel directly. Particle movement is determined by the current force effect inside the voxel. This can be either a linear force or a rotational force. Rotational forces are projected onto linear forces, because small time-steps are assumed during the calculation steps. This allows a simple calculation of movement inside the particles independent of the complexity of the original forces. For simplification of the calculation, forces are stored at the center of the voxels. Linear forces act uniformly inside the whole voxel. The axis for rotational forces or the gravitational point are also placed at the center of the voxel but act differently on particles at different locations.

Since the voxels are all evenly spaced, the particle–voxel assignment can easily be established as an array access in linear time. The size and resolution of the voxel-grid does not contribute notably to the update time of the animation.

Unlike the common implementations discussed earlier, particles are kept alive for an indefinite period of time while they are inside any activated voxels. This helps to keep the animation stable. When a particle enters a new voxel, it gets new target attributes assigned. Interpolation is done linearly for color and size and texture animation speed. The interpolation time is also retrieved from the new voxel. Particles can, therefore, change their appearance at different speeds inside different voxels. This ensures more freedom for the animator as well as smooth transitions between the voxels, but still makes it possible to have the particles react totally independent inside the separate voxels. Having a set of target attributes also makes it possible to change the attribute instantly, for instance when the voxel attributes change or the particle leaves the defined voxel space, or changes into a new medium, such as bubbles moving from water into the air. This interpolation can be seen in Figure 4. The interpolation between the two voxels is surrounded by the green circle.

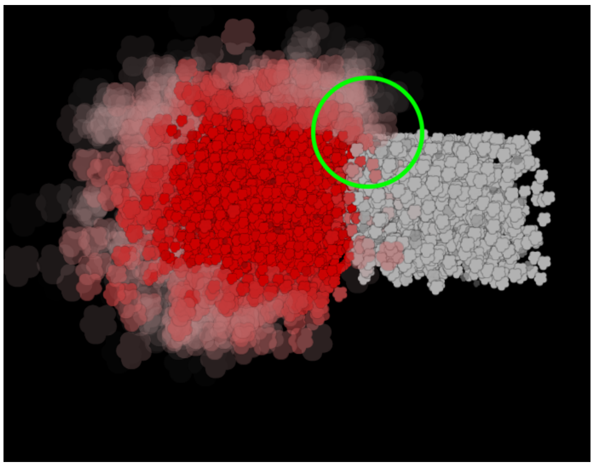
Figure 4: Particle shape with only two voxels. Particles on the left fade away slowly when leaving their voxel, on the right they fade away instantly.

As soon as a particle leaves an activated voxel and enters the free space surrounding the model, the normal fading behavior is triggered. The fading corresponds to the fading behavior attributes of the last visited active voxel and is consistent with the behavior described by Reeves. However, if a particle re-enters an activated voxel, it's lifetime will be reset to zero again and the particle will continue to live. The normal animation model can be followed then as described earlier.

4.4 Displaying the particle system

There are different ways to render particles. Reeves explained in his original paper that he draws the particles as simple points or lines for every particle. This is sufficient for small fast moving objects. However, when displaying larger, slow moving objects, it is common to use billboards to display the particles. These are commonly mapped with textures to provide additional level of detail. These textures can be animated or video textured. In our implementation, texture-sets can be defined and played at different intervals. Those intervals are stored inside the voxel definition.

Today's graphic frameworks like OpenGL or DirectX provide different kinds of functions that define how transparent objects are combined on screen. Blending is the mechanism for combining color already in the frame buffer with the color of the incoming primitive. The result is then stored back in the frame buffer.

Rendering every particle independently may be suitable if additive blending of colors is used. Many overlaid particles become very bright. They get a white, glowing look to them. This may be reasonable for effects like fire and other light emissive special effects. When rendering more complex phenomena, however, such as cloud layers or diffuse objects, it becomes important to use additional alpha values that are stored inside the color information or the alpha channel of the textures to perform the blending operation.

Rendering transparent objects is always performed with the depth-buffer being disabled. When displaying the particles in arbitrary order, overlapping particles from further away may occlude others that are closer to the viewing plane. Because of that, they have to be rendered in the right order from back to front. The hardware's method to achieve this, is writing values into the depth buffer. As new primitives are rendered, their z-values are checked against the ones stored inside the depth buffer. If they are smaller than the previous ones, the primitives can be rendered. To display particles that have alpha values stored for transparency and need to be rendered accordingly, a sorting algorithm should be used. We chose a simple quick sort algorithm, since it is directly supported by C++ through the standard template library and does not need any specialized hardware. Since sorting thousands of particles in real-time can limit the frame-rate drastically, sorting should be limited to a minimum. Sometimes, it is sufficient, to just sort parts of the particles for adequate visual effect.

5 INTERACTION, MANIPULATION, AND DYNAMICS

There are a few common interaction techniques for manipulating scenes and objects in 3D virtual environments. The most basic include translation, rotation, and scaling of objects. Normally those actions are performed by a transformation matrix that can be applied to the original vertices of the geometric models. This works fine there, since the geometry is connected and all the vertices still have the same relation to each other.

As the particles used in the presented method form a highly chaotic system and are not attached to anything solid, transformation matrices can not be applied that easily. Every manipulation of the whole system, i.e. a transformation of the voxelized model, acts as a kind of force onto the particles. The particles, in turn, have to move accordingly. Movement and rotation of the system influences different particles in differing ways. While old particles typically keep their current momentum and move according to their force values, newly generated particles receive additional momentum from the movement of the underlying structure. Therefore, interaction and manipulation with the objects mainly

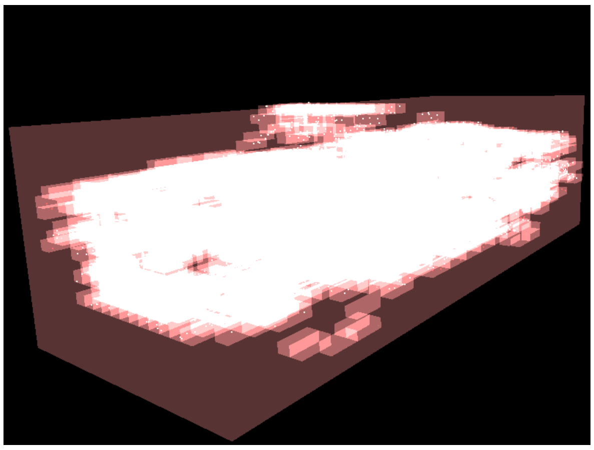
Figure 5: The bounding boxes of the voxelized shape of a car.

has to deal with the application of forces to the particle system, both overall and on voxel level. Those forces can in addition be used to create additional visual features in the resulting shapes and characters. The reminder of this section describes interaction with and animation of the fuzzy objects by discussing forces applied to the system or to single voxels and forces that happen through collisions.

5.1 External forces and forces inside a voxel

In addition to the color, size, or animation speed, particles can be influenced by different forces inside the voxel-space. We distinguish forces that act on the whole system from the outside, which include movement, wind, or gravity, and forces that act on every particle differently. Transformation of the whole system is modeled as just another external force that affects newly generated particles as an initial momentum acting on the particles.

In our implementation, the designer can decide, which force he wants to use in certain voxels and can set them individually. We implemented four different kinds of forces to be used inside every voxel. They are *linear forces*, like wind or gravity; *spiral forces*, that move the particle around an arbitrary axis at the center of the voxel; a *gravitational point* at the center of a voxel, that attracts particles and forces them to move around in an orbit; and a helix-like movement, that rotates the particles around an axis in a specified direction. Fluid dynamics were also considered, especially since the voxel-grid already implies the computational grid used there, but were dismissed in the initial approach, because their calculations were too complex for large voxel-grids in the real-time virtual environments that were targeted.

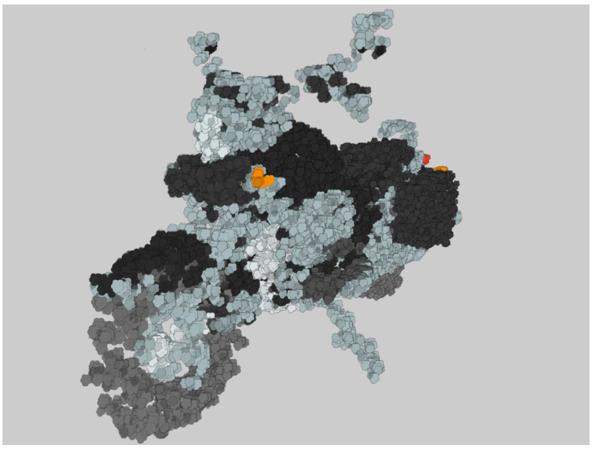
Figure 6: A chopper made of particles. Though it has a quite solid appearance, particles move constantly.

5.2 Collisions with solid objects

When looking at smoke in reality, any collision with the system, like moving a hand through a cloud of fog impacts the movement of the particles. However, since collision is also a very time consuming progress, especially when dealing with thousands of different particles, it has to be limited. To decrease the time for the calculation drastically, a hierarchical system has been added to the particle based shape.

Bounding volume collision of the whole system can nicely be performed by the application's framework. The voxel-object's bounding box is tested against every other object in the virtual environment. If an object is penetrating the surrounding bounding box of the whole system, a more detailed test can be performed. This is done using the volumes of the individual voxel (see Figure 5). If a voxel penetrates a solid object or vice versa, voxels are temporarily deactivated. This is important, since particles would otherwise still be generated inside the voxels, leading to particles appearing inside the solid object.

Additionally, every particle that is currently inside colliding voxels has to be checked for direct collision with the object. This has to be done on a per triangle basis, since particles hit an object directly and the collision response needs to be as accurate as possible. As the particles position before and after the update-step can be considered as a line, collision detection is done using a line-triangle algorithm.

6 RESULTS

This section shows some images we have generated using our particle based system. Figure 6 shows how closely surface geometry can be approximated with our particle system. Even though this example uses only a grid of 40^3 voxels and about 30,000 particles, many details of the chopper are visible. It can be noted, how the initial coloring of the model was used to include additional detail at the position of the indicator.

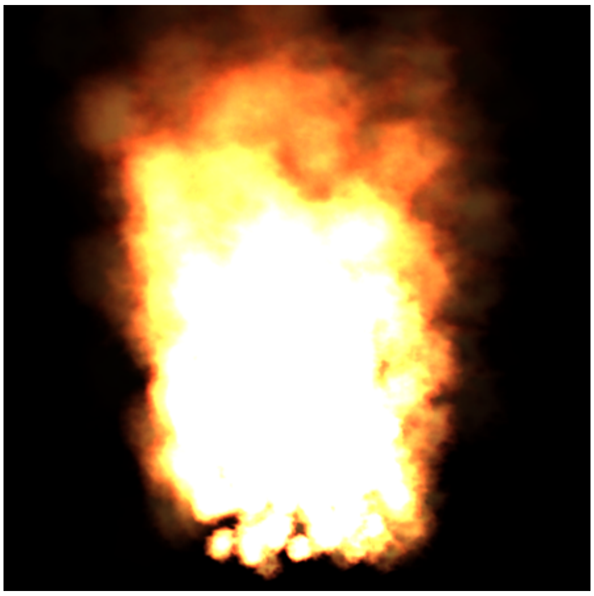
Figure 7: Using just one voxel, even conventional effects, like this fire can be created.

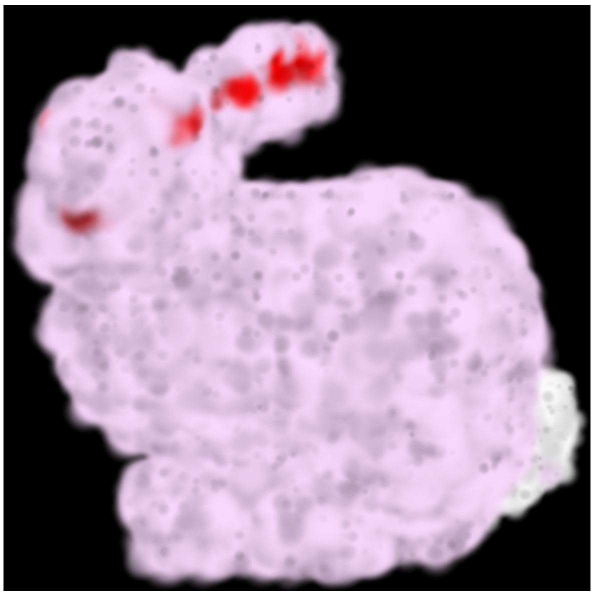
Figure 8: The Stanford bunny as a "fluffy" version using our particle based system.

Simple effects, like the fire shown in Figure 7 can easily be created by our system. This example uses just one voxel and an animated texture for the particles. More complex models or video textures may also be used. Rendering the model using additive blending is useful for rendering fire and other light emitting objects. All results were generated using an AthlonXP 2000+-based laptop with a ATI Radeon 9000 mobility graphic-card. Because of this, most shader-based optimization could not be employed. Further optimizations using more current graphic hardware or multi-processor systems will increase the frame output immensely.

For our method, most objects could be sufficiently represented as a voxel-grid with a dimension of 50^3. As the complexity of the models rises, the voxel-grid needs to be more precise. It should be mentioned though, that memory consumption increases rapidly and large voxel-grids are not encouraged, even though the time for the computation does not rise. Since packing the voxel-grid to save memory will result in eliminating the fast access of voxels through positions of the particles, compression cannot be performed easily.

The image on the first page (Figure 1) shows the popular TV character Marvin the Martian from the show Looney Tunes. The diffuse model was voxelized with a 50^3 grid and rendered with 40,000 particles visualized as point sprites. The rendering times are at about 10 to 15 frames per second.

The model of the Stanford bunny in Figure 8 was rendered to simulate the "fluffy" fur of the bunny. At the nose, eyes and ears voxel where colored red, at the rear, the tail was colored white. Particles were instantly colored in the correct way.

7 CONCLUSION AND FUTURE WORK

This paper presented a new method to approximate fully controllable, complex, fuzzy shapes to be used inside interactive 3D virtual environments. The shape approximates the provided polygonal geometry closely, while still having a fuzzy and dynamic exterior, with interaction and reaction to other objects being possible. Forces are integrated into the shape to change the appearance during runtime. For this method, a fast structure for creating, rendering, changing, and updating structures was required.

To achieve this, regular surface models are converted into voxelized versions. Particles are spawned afterward inside the voxels to create a highly dynamic version of the previous surface geometry. Forces that are stored inside the voxels act on the particles and move them along paths. Complex rotational forces make the particles move in spirals, orbits, or as vortices inside the voxel. Additional attributes inside the voxels, like size and textures define the appearance of the particles during the render process. Particles are created during runtime inside the voxels to approximate the original shape. Generation is done inside the world's coordinate system, to let simplify calculations when the particles interact directly with other objects.

Voxels are created efficiently through hardware, making it possible to create voxels as needed in limited time to let the user focus on the design and appearance of the model using traditional surface modeling tools. Vertex colors are used during the conversion step to create the initial coloring attributes of the voxels. Through simple force calculations inside the particles, the performance impact for updating the particle attributes could be reduced to a minimum. This leaves most of the computational power to the application. Interaction between a particle system and rigid bodies inside the virtual world have been implemented through a structured collision detection and response algorithm.

The applications for this system range from traditional rendering of fire, clouds, and dust to diffuse ghost-like characters and objects. The presented particle system can also be used for visualization of stress fields inside materials. Particles would move through the force field created by the voxels to form clusters, where the forces are too high to maintain further structural integrity of the material. Other, more artistically oriented methods include drawing of voxels in an virtual environment directly to use as an interactive diffuse paint metaphor.

The new particle system produces satisfying results inside a controlled environment. Some design aspects had to be considered during the creation of the system. Using highly detailed geometry is only useful if the voxelization is performed with a fine resolution. This, necessarily, requires smaller particles and, therefore, a higher number of particles inside the system. As a result, the particle system's performance rapidly decreases as the amount of particles rises. Coarser models with a resolution of 60 voxels per side prove to be sufficient for the trade-off between visual complexity and real-time performance. Additional tuning of the particle system can also enhance the visual appearance.

When using models with very detailed parts or small, thin spikes, holes occur, where only a single row of voxels is representing the originating geometry. This can be avoided, by adjusting the forces inside the voxels along the path of the geometry. An automated approach could be employed to aid the design process.

The system works well with simple independent models. To further extend our system, interaction between more than one particle system should be implemented. Pressure fields and fluid dynamics were not added yet, because of calculation time issues, but would enhance the visual complexity of the system. As hardware implementations progress, fluid calculations should be possible even for large grids. Complex lighting calculations inside the particle system would also improve the system and will probably be integrated later.

REFERENCES

[1] Steffi Beckhaus, Jürgen Wind, and Thomas Strothotte. Hardware-based voxelization for 3d spatial analysis. In *Proceedings of CGIM '02*, pages 15–20. ACTA Press, 2002.

[2] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 21–29. ACM Press, 1982.

[3] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 99–107. Eurographics Association, 2002.

[4] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: A gpu-based particle engine. *Graphics Hardware*, 2004.

[5] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[6] Stijn Oomes, Peter Snoeren, and Tjeerd Dijkstra. 3d shape representation: Transforming polygons into voxels. In *Proceedings of the First International Conference on Scale-Space Theory in Computer Vision*, pages 349–352. Springer-Verlag, 1997.

[7] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[8] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[9] Karl Sims. Particle animation and rendering using data parallel computation. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 405–413, New York, NY, USA, 1990. ACM Press.

[10] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[11] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 185–194. ACM Press, 1992.