A Programmable Particle System Framework For Shape Modeling

Abstract

Particle systems are an effective tool for visualizing information in a variety of contexts. This paper focuses on the use of surface-constrained particles to visualize information about the surface. We have designed a particle system programming framework consisting of behaviors, attributes and shaders that allows users to rapidly create, debug, and deploy particle systems for sensing and extracting specific surface information and displaying this information in an visually effective manner. We also introduce a simple particle system “little language” to facilitate the articulation of these particle programs. We demonstrate the flexibility and power of this framework for surface visualization with the applications of singularity detection and display, non-photorealistic surface illustration, and surface mesh algorithm visualization.

1. Introduction

Particle systems are widely used in many areas of computer graphics and scientific visualization. Among these uses, they have also become an effective tool for the visualization and interactive modeling of surfaces, and there are now a wide variety of different particle systems devised for performing various shape modeling tasks. This paper described a new particle system programming framework (and accompanying open source library) designed to facilitate the implementation of these systems and the exploration of new particle-based shape modeling applications.

The novel contributions of this systems paper include (1) the decomposition of particle programs into attribute, behavior and shader building blocks, (2) the decomposition of behaviors into force, constraint, integration and cleanup phases, (3) the derivation of new behaviors that cause particles to move to the silhouette or to singularities of implicit surfaces. The remainder of the paper is devoted to motivating and describing the design of this particle system framework and demonstrating its usefulness for rapidly implementing previously published particle systems and exploring new particle system ideas.

Particle systems were first used to model natural phenomena like fire, smoke and water [20]. Though an early application described how an oriented particle system could self-organize to define a surface [24], the idea of using particles to track, display and control a predefined dynamic surface has achieved more success with a wider variety of applications. Witkin and Heckbert [32] showed how a particle system can be used as an effective tool for real-time display and manipulation of implicit surfaces. They constrained a particle system to lie on the surface, displaying it with a textured field of opaque oriented polkadots on an implied clear transparent surface (e.g. Fig 1(a)). This particle system also served as a manipulation widget by selecting some particles and solving a dynamic constraint to force the implicit surface to pass through these particles.

A large number of applications that rely on surface-constrained particle systems have appeared. The mutual point repulsion of surface-constrained particles yields a useful method for evenly sampling a surface, and has been used for reaction diffusion texturing [26], retesselating meshes [27], cellular textures [7], local surface parameterization [18], morphing [14], free-form modeling [30, 16, 10] and surface texture synthesis [28, 29], to name a few.

The use of a surface constrained particle system is thus an important tool for computer graphics. However, its varied applications often require sometimes subtle, sometimes major, alterations of (1) the dynamics of particles as they distribute across the surface, (2) the state carried by each
particle and the particle system as a whole, and (3) the particle-user communication through appearance and interactivity. Though minor alteration can often be accomplished through parameter changes in an existing particle system, major alteration has often, as reported by many in the literature, required the construction of new particle system applications largely from scratch.

A similar situation occurred in the 1980’s when the graphics community aggressively investigated the utility of writing programs that procedurally generated texture and other appearance phenomena, e.g. [4, 19]. New shading systems were reconstructed, largely from scratch, until in 1990 the Renderman shading language [8] showed how a small, focused language could simplify, clarify and standardize the articulation of procedural shading algorithms. Our goal is to provide the same level of assistance to the articulation of particle systems, focused primarily on the particle systems used for shape modeling and texturing.

Fleischer et al. [7] took a step in this direction by decomposing particle behaviors into component pieces (e.g. separating interparticle repulsion from surface adhesion), then integrating these pieces through a biological metaphor. Alias’s Maya graphics system contains an little language called MEL that includes Particle Expressions capable of customizing the behavior of a particle system, but requires the composition of custom behaviors largely from scratch and lacks the modularization of behaviors and attributes.

Our proposed framework extends Fleischer et al.’s behavior decomposition concept into a more complete particle programming framework capable of broader applications than cellular texturing. We also seek the programmability found in MEL’s Particle Expressions, though embedded within the modular approach of Fleischer et al. we find so attractive.

As described in more detail in Sec. 3, our system organizes a particle system into (1) behavior objects that decompose and compartmentalize the action of a collection of particles into reusable and interchangeable modules, (2) shader objects that likewise encapsulate the appearance and user interactivity of the particles, and (3) attribute objects that contain state information and utilities that can be easily shared and accessed among the different combinations of behavior and shader objects. These objects are collected together to describe a homogeneous collection of particles in a Particles object, and multiple Particles are collected into a ParticleSystem object. For example, the Witkin-Heckbert particle system [32] would consist of two Particles objects: floater particles that disperse across the surface to display it, as in Fig. 1(a), and controller particles that serve as direct manipulation widgets used to model the surface.

The modular design of this framework provides an intuitive mental model for the rapid prototyping of particle programs from reusable high-level building blocks, while the programmability of the building blocks allows the programmer to add functionality efficiently while focusing only on the added component. This organization resembles that of Renderman, which similarly allows the custom coding of shader building blocks that could be reused in high-level shading networks.

This new framework enables the easy implementation of existing particle system systems from their description in the literature and the exploration of new applications. Sec 3 concludes with a demonstration of the Witkin-Heckbert particle system implemented in this framework. We then use the framework for other applications. For example, Fig. 1(b) demonstrates the use of the system to find and visualize the singularities of an algebraic system, described in more detail in Sec 4. We also investigate the use of particle systems for non-photorealistic rendering in Sec. 5, polygonization in Sec. 6 and mesh clustering in Sec. 7.

2. Previous Work

Witkin and Heckbert’s floater and controller particles [32] inspired the framework described in this paper. They used a surface constrained oriented particle system to display an implicit surface in real-time, faster than possible then by ray tracing or direct polygonization [1, 2]. Their floater particles could be decomposed into three basic behaviors. An adaptive repulsion system centers a variable-width Gaussian energy function at each particle whose gradient exerts a force on neighboring particles in search of an energy minimizing equilibrium. A surface adhesion constraint restricts the force created by repulsion on the particles, by subtracting the force component normal to the surface. Finally, a population control system subdivides isolated particles and deletes crowded particles, and also dynamically grows or shrinks the support of the per-particle Gaussian energy function depending on the proximity of neighboring particles to hasten equilibrium. The main behavior of the controller particles is to convert their velocity to a corresponding change in the implicit surface parameters. The end of Sec 3 describes how our framework implements this particle system.

Our framework more closely resembles that of Fleischer et al. [7], which decomposed the Witkin-Heckbert system into component behaviors. Our framework extends this decomposition into behaviors and shaders, the latter of which controls the appearance and user interaction of particles. The Fleischer et al. system also assumes a global state which we have further decomposed into shared attribute modules that can be loaded into the particle system on demand. Furthermore, we have subdivided the specification of behaviors into four key phases: force, constraint, integration and cleanup, and interleaved these phases among the behaviors to ensure they occur in the proper order.
An alternative to our particle system programming framework is provided by commercial graphics systems, exemplified by Alias’s Maya package. Maya includes MEL, a scripting language designed for programming custom behavior that includes Particle Expressions for controlling the behavior of particle systems. These Particle Expressions operate on a global collection of particle attributes that resemble the attributes of our framework except they have not been modularized, and neither have the behaviors and shading instructions specified by the Particle Expressions. While any of our examples could have been implemented in Maya, this lack of clear modularization hinders the ability to reuse individual particle behaviors or rapidly construct applications from a collection of building block modules.

3. A Particle System Framework

In the Wickbert system, a Particles is a homogeneous collection of particles that behave similarly, and one or more of these can be collected together in a ParticleSystem object. A Particles object contains a sequence of ParticleBehavior objects that collectively describe the action of the particles, a sequence of ParticleShader objects that describe how the particles are drawn (and manipulated), and a collection of ParticleAttribute objects that contain shared data used by the behaviors and shaders.

Each of the ParticleAttribute, ParticleBehavior and ParticleShader objects inherits the same base class ParticleStuff which provides a uniform interface for managing per-object and per-particle parameters. This design allows an application to provide a single dynamic GUI interface to adjust parameters for any of the attributes, behaviors and shaders. Thus new behaviors, shaders and attributes can be designed and plugged into the system, and existing ones can be adjusted at run time.

3.1. Attributes

The per-particle and global data elements and procedures of a Particles object are organized into several atomic ParticleAttribute objects. These attributes are stored and referenced by name in an associative array. Behaviors and shaders access an attribute at an initialization phase by the attachAttribute(attr,name) method, which finds the attribute named name and assigns the pointer attr to it. If no such attribute exists, it finds and assigns the first compatible attribute of the type of the pointer attr.

Examples: An ImplicitInterrogator attribute contains a pointer to an implicit surface object similar to the those described elsewhere [32, 9]. An AdaptiveRepulsionData attribute contains the per-particle radii of the particles as well as the global parameters setting the desired and maximum radii values of the particles.

3.2. Behaviors

The motion of particles is represented by an ordered collection of ParticleBehavior objects, each of which represents a different atomic component of the behavior of the particles. The contribution of a ParticleBehavior to the behavior of a particle is divided into four computational steps:

- applyForce() - add this behavior’s force to a force accumulator (a per-particle attribute),
- applyConstraint() - modify the current force on each particle to satisfy a constraint on its motion,
- integrate() - update the particle (e.g. its position, velocity) based on the constrained force held in each particle’s force accumulator, and
- cleanup() - create and destroy particles based on population dynamics.

These steps are interleaved among all of the behaviors such that the applyForce() method of each of the behaviors (in order) is called first, then applyConstraint() of each behavior, then integrate() and cleanup(). Thus the constraint operates on particles after all of the forces have been computed and applied, integrate operates after all forces have been modified by all constraints, and cleanup operates after all particles have updated their new state.

Examples: The applyForce() method of the ParticleRepulsion behavior computes a force that drives particles away from each other, whereas the applyConstraint() method of the SurfaceAdhesion behavior removes the component of the accumulated force that would cause a particle to leave an implicit surface.

![Figure 2. Particles object data flow. Attributes collect information shared by Behaviors and Shaders. User observes particles and interactively adjusts parameters.](image)
3.3. Shaders

An ordered collection of ParticleShader objects controls particle appearance and user interaction. The ParticleShader object includes a draw() method which defaults to calling its drawParticle(i) method on each particle i. A shader can define a new particle appearance by redefining drawParticle(), or can alter a global appearance by redefining draw(). Since each shader can change the graphics state, their order is important. We also control user interaction through the shader, by redefining its event() method, which is called by the application whenever a user interaction event occurs (e.g. a mouse click).

Examples: The shader ParticleShaderDisk redefines drawParticle(i) to draw a disk in the current coordinate system, which is automatically set up by ParticleShader defaults for particle i whereas the shader ParticleShaderConstMaterial redefines the OpenGL material appearance attributes for all shaders that follow it by redefining its draw() method. The shader CopyParticle waits for a certain mouse event to occur on one of its particles, and when it does, directs a different Particles collection to create a new particle at the current particle’s position.

3.4. The Programming Framework

We have found that this organization of the state, behavior and appearance of particles provides an intuitive mental model for the articulation of particle systems. It forces the programmer to dissect the conception of a particle system into its component behaviors, the state needed by those behaviors, and how it should communicate the results to the user through its appearance and event processing. This organization also promotes the abstraction of behaviors, attributes and shaders into reusable components that can be connected in a variety of different configurations for the rapid prototyping of new particle systems.

As such, this programming framework serves the same purpose for specifying particle systems as did Renderman for specifying the appearance of scenes. Rather than develop a “little language” such as Renderman does for its shaders, we have opted for the familiarity and speed of the C++ language. This framework provides the details of particle system maintenance allowing the programmer to focus on the kernel of the desired new particle action or state.

This framework also benefits from the object-oriented structure of C++, supporting the inheritance and specialization of attributes, behaviors and shaders. For example, the ParticleLocality attribute provides a method for finding the neighbors of a given particle within a given radius, and implements this by a simple linear-time global query of the particles, whereas the ParticleLocalityGrid attribute inherits the ParticleLocality attribute, but redefines its data structure with a uniform 3-D voxel grid and its neighboring-finding method with a constant-time grid lookup. Behaviors and shaders that depend on particle neighborhood queries can find an attribute of type ParticleLocality and use its interface, even though the actual attribute is the more efficient ParticleLocalityGrid. Thus the user can use such modules interchangeably depending on application specific needs ranging from simple proof-of-concept prototypes to efficient production runs, or in other cases for time/space complexity tradeoffs.

The rest of the paper demonstrates the benefits of this framework by showing how it can be used to articulate a variety of different particle systems that would otherwise require a significant amount of effort to implement.

3.5. Example: Witkin-Heckbert Particles

Table 1 demonstrates this framework applied to the particle system described by Witkin and Heckbert [32]. They constrained a mutually repulsive particle system to an implicit surface to display the surface. The mutual repulsion of these particles was controlled by a dynamic Gaussian energy function and the width of these Gaussians can vary across particles and over time. These particles would also subdivide when isolated and die when overcrowded.

These surface display particles are described by the “Floaters” collection of particles in Table 1. The AdaptiveRepulsionData holds the Gaussian energy function information. The ImplicitInterrogator attribute is used to query the implicit surface to which the particles adhere. The ParticleLocality attribute contains a getNeighbors(i,radius) method that returns all particles within a distance of radius from particle i. This Witkin-Heckbert particle system was based on viscous dynamics where force equals mass times velocity, which his handled within the framework by the ViscousState attribute and the ViscousDynamics behavior.

The second half of the Witkin–Heckbert approach was to select some particles as “controllers” and perform a minimum work adjustment of the implicit surface parameters to ensure the surface always passed through these particles. When a controller particle is dragged, the implicit surface will dynamically adjust to pass through it and the other controller particles.

We implemented this action with the “Controllers” collection of particles in Table 1. This particle collection is initially empty, and particles are added to it by the CopyParticle shader in the “Floaters” collection. This particle collection also does not contain any behaviors with an integrate() method. Particles are dragged by the DragParticle shader which updates its position and stores the difference as a velocity. The applyConstraint() method of the SurfaceDeformation behavior then converts this particle velocity into a
Particles: “Floaters”

ParticleAttributes:
- AdaptiveRepulsionData
  - Per-particle: radius
  - Global: desired_rad, max_rad, repulsion_amp
- ImplicitInterrogator
  - GetImplicit() { return an implicit }
- ParticleLocality
  - getNeighbors(i, radius) { return neighbors }
- ViscousState
  - Per-particle 3-vector: x,v

ParticleBehaviors:
- ParticleRepulsion
  - attachAttribute(AdaptiveRepulsionData)
  - applyForce() { compute repulsion }
  - integrate() { change radius }
- SurfaceAdhesion
  - attachAttribute(ImplicitInterrogator)
  - applyConstraint() { remove non-tangent vel. }
- ViscousDynamics
  - attachAttribute(ViscousState)
  - integrate() { x += v * dt }
- ParticleFate
  - attachAttribute(AdaptiveRepulsionData)
  - cleanup() { fission and death }

ParticleShaders:
- ParticleDisk
  - attachAttribute(AdaptiveRepulsionData)
  - drawParticle(i) { draw disk }
- CopyParticle
  - event(doubleclick) { create new particle in “Controllers” }

Particles: “Controllers”

ParticleAttributes:
- ImplicitInterrogator
- ViscousState

ParticleBehaviors:
- SurfaceDeformation
  - attachAttribute(ImplicitInterrogator)
  - attachAttribute(ViscousState)
  - applyConstraint() { transfer particle velocity to implicit parameter velocity }
  - integrate() { update implicit params. }

ParticleShaders:
- ParticleCylinder
- DragParticle

Table 1. The “Witkin-Heckbert94” particle system.

4. Singularity Particles

Surface constrained particle systems perform well on smooth surfaces, but can be problematic on surfaces with singularities such as creases, cusps and self-intersections. Such singularities occur on an implicit surface where the gradient vanishes or becomes discontinuous. In such cases, the orientation of particles becomes discontinuous and the particles tend to oscillate about the singularity, as can be seen at the bottom of Figure 1(a).

Rosch et al. [21] first identified this problem, noticing that when two sheets of a surface intersected (e.g. from the 3-D immersion of a Klein bottle) or in areas where curvature increases to infinity (e.g. the cusp of a cone) the particles were sparse. They solved the intersecting-sheets problem by allowing particles to only repel other particles with similar orientation. They overcame the sparseness in high curvature areas by making particle repulsion radius inversely proportional to curvature.

These enhancements could be integrated directly into our framework through an enhanced behavior that inherits ParticleRepulsion but compares particle orientations, and through the addition of a ParticleCurvature attribute that effects the dynamic resizing of the particle radius in the AdaptiveRepulsionData attribute. But we instead propose an alternative solution suited more for the visualization of singularities on an isosurface.

Rather than manipulating the repulsion energy of a particle, we instead use our framework to construct a new collection of particles, called “Singularity Particles”, whose behavior causes them to adhere to the singularities of the surface, and whose shaders indicate they are unoriented with spheres (since the gradient vanishes at a singularity).

Figure 3. (a) The crease in the middle of the heart is caused by singularities. (b) Regions with small gradients are indicated in blue.
Figure 4. (a) Singularity particles find features in a CSG model. (b) Singularity particles display degenerate portions of the Steiner surface that ray tracing would miss.

We will apply these singularity particles to a heart shaped implicit surface [25] of the function

\[
f(x,y,z) = (x^2 + 2.25y^2 + z^2 - 1)^3 - x^2z^3 - 0.1125y^2z^3 = 0.
\]

(1)

This surface contains rather obvious isolated singularities at its bottom and top cusps. What is less obvious is that its gradient magnitude reduces to zero along a closed loop at its midsection, which causes problems with the surface normal when ray tracing it, as shown in Fig. 3. Our goal for singularity particles is to interrogate an implicit surface function to find these problem areas and make them obvious to the investigator.

4.1. Implementation

We implemented singularity particles by creating a new behavior module called SingularityInterrogator that adds a force onto particles in the direction opposite to the gradient of the gradient norm squared:

\[-\nabla(\nabla^2 f)\].

This moves the particle in the direction of decreasing gradient magnitude, in search of a minimum whose value is zero. In other words, we want to move in the direction where the gradient norm squared is small.

The collection of singularity particles is initially empty. We could seed them with particles in random positions across the surface but we found it was better to use the floater particles, which give us the ability to interrogate a sampling of the entire surface, to trigger the creation of new singularity particles. A DetectSingularity behavior is added to the floater particles that tracks the gradient of the implicit surface at the particle’s position. When the gradient’s magnitude falls below a preset threshold, or when its direction changes faster than a present threshold, the behavior directs the “Singularity Particles” to create a new particle at that location. The singularity particles otherwise follow the same behaviors as the floaters (though with the added force of the SingularityAdhesion) and subdivide in an effort to grow and cover whatever singularity exists, such as the closed loop at the midsection of the valentine heart.

We also alter the ParticleLocality attribute in the “Floaters” particles so its getNeighbors() method returns not only nearby particles from “Floaters” but also from “Singularity Particles”. Thus the singularity particles serve as a warning barrier to prevent floater particles from venturing too close to regions of the surface which cause floaters to malfunction.

4.2. Discussion

We found that the combination of sampling with the floaters and searching with the singularity particles converges quickly for many shapes including those surfaces with hard corners. We tested it a number of surfaces including CSG models and the Steiner surface.

The singularity particles also serve as a feature detector as they naturally find creases. The dumbbell shape formed by the union of a capped cylinder with two spheres contains two circular creases at the sphere-cylinder intersections. The floater particles are unstable near these locations,
and so generate singularity particles that find the creases and move the unstable floaters away, as shown in Figure 4(a).

Steiner’s surface

\[ f(x, y, z) = x^2y^2 + y^2z^2 + z^2x^2 + xyz \]  

includes the three coordinate axis, because if any of \( x, y, \) or \( z \) is zero then \( f \) is zero. These axes are infinitely thin and a simple ray tracer would likely miss these features. The addition of singularity particles makes clear where these degenerate surface segments lie, as shown in Fig. 4(b). This figure reveals some floater particles (the ones that spawned the initial singularity particles) remain trapped on the axes by the repulsion of the singularity particles, but these floaters will eventually die due to isolation.

Thus with the addition of a single new behavior and some reconnection of modules, we adapted a copy of the floater particles to interrogate the singularities of an implicit surface.

5. Silhouette Particles

Silhouette curves are widely used in the illustration of surfaces as an effective method for visually conveying the shape of a surface without the overhead or distraction of photorealistic shading [17, 12, 31, 3]. What we call the silhouette curves are defined as all points on the surface where the normal of the point \( N \) and the view vector \( v \) are perpendicular or \( N \cdot V = 0 \). (This is perhaps more precisely known as the contour.) We can use our new framework to construct a new collection of silhouette particles, and use these particles to construct a particle system that yields a non-photorealistic rendering of the implicit surface.

5.1. Implementation

Our silhouette particles begin as a copy of the floater particles. To this, we add a new behavior called SilhouetteAdhesion that moves the particles toward the silhouette. The silhouette is the zero set of the function \( g(x) = N \cdot V \) where \( V \) is the view vector. Instead of solving for \( x \) directly, we use gradient descent search to move particles in the opposite direction of the gradient of \( h(x) = (N \cdot V)^2 \), the squared magnitude of the silhouette function. This gradient is thus

\[ \nabla h(x) = \nabla (N \cdot V)^2 \]

\[ = 2(N \cdot V)((\nabla N) V + N \nabla V). \]  

We really care only about the direction of the normal and not that it is unit length, so we can replace \( N \) with \( \nabla f \) and thus the gradient of the non-unit normal \( \nabla N \) is the Hessian matrix of second derivatives \( H \). The view vector, which depends on \( x \) and the camera position \( c \), is defined as \( V(x) = c - x \) and its gradient is the Hessian \(-I\) which can be simply replaced by the scalar \(-1\). This yields

\[ f = 2(N \cdot V)(H V - N) \]

as the force \( f \) implemented in the applyForce() method of the new behavior SilhouetteAdhesion.

We also created a behavior called SilhouetteFate that replaced ParticleFate in the silhouette particles. ParticleFate was tuned for the distribution of particles on a surface whereas we now need particles to subdivide and populate alive along a space curve.

5.2. Discussion

Fig. 5(a) demonstrates the results. We also devised a new shader called ParticleChain that finds the two closest particles in the neighborhood and connects them with a line. This provides a nice visualization of the silhouette curves, including its cusps and occluded portions. However, this would not make a very convincing illustration of the torus.

Fig. 5(b) demonstrates the same behavior with different shaders. We modified to ParticleDisk shader of the floater particles to draw large overlapping disks of the background color. We deleted the sphere particle shader and altered the ParticleChain shader to display more artistic stroke shapes between the silhouette particles. The result is an illustration of the surface that removes the hidden portions of the silhouette curves.

By copying the floater particles, we were able to depend on the fact that our silhouette particles would adhere to the surface and repel each other. This allowed us to focus our effort on the additional force needed to push particles along the surface to the silhouettes.

![Figure 5. Silhouette particles interactively find the silhouette curves on a torus (a). A stroke shader yields a hand drawn look for the same torus, using large floater disks of the background color to hide occluded silhouette segments (b).](image)
6. Dynamic Meshing

While evenly distributed particles allow us to infer a surface, it would be preferable to represent a surface with a triangulation. The speed and robustness of the Witkin Heckbert approach [32] is due partially to their ability to ignore the overhead of connecting particles into a triangle mesh and to maintain the validity of this triangle mesh as the underlying implicit surface changes. The connection of particles into a dynamic triangle mesh has been investigated before (e.g. [23] which also included a topology guarantee). Welch and Witkin used adaptive meshing for free form modeling [30]. Markosian et al. used a particle based dynamic mesh that allows users to sculpt free-form surfaces [16]. Here we discuss the implementation of a dynamic mesh using our proposed framework, assuming the per-timestep change in the surface is small and ignoring changes in topology of the underlying surface.

6.1. Implementation

Our approach again begins with a copy of floater particles. We augment these particles with a ParticleMesh attribute that encapsulates a half-edge mesh data structure that references particle positions (such as those in ViscousState) as the vertices. A ParticleLocalityMesh attribute inherits and replaces ParticleLocality, and uses the half-edge mesh in the ParticleMesh attribute to accelerate the getNeighbors() function.

The ParticleRepulsion and SurfaceAdhesion behaviors keep the mesh vertices on the surface, but their motion can deform and even invert faces in the mesh. An additional behavior called MeshShape is constructed to perform edge flips in its cleanup() phase in an effort to keep the mesh as close to Delauney as possible. While some have recommended maximizing the minimum face angle [16], we instead use the criterion of flipping an edge only when the new edge will be shorter. This criterion was easier to implement and appears to work satisfactorily for our example, as shown in Fig. 6.

We also implemented vertex split and edge collapse routines in the cleanup() phase of MeshShape attribute. Some have recommended edge length as a criterion for creation/deletion of new vertices [16], but we have found that setting triangle area and triangle count are also useful factors. We use the Markosian et al. [16] idea to sort the potential splits or collapses based on area and only apply to the top portion of them to keep an interactive rendering rate.

6.2. Discussion

As before with the shaded strokes of the chain connecting the silhouette particles, we have augmented our particle system with an attribute structure whose contents are not necessarily related to a single particle. In this case, the ParticleMesh half-edge data structure does not have a one-to-one correspondence with the particles, which serve as its vertices. Nevertheless, the framework is flexible enough to handle such an attribute.

We also benefit from inheritance by specializing ParticleLocality into ParticleLocalityMesh which uses the mesh datastructure for acceleration. This specialization means we do not need to change ParticleRepulsion, even though the new ParticleLocalityMesh attribute is reporting a different collection of neighbors.

7. Mesh Algorithm Visualization

We can use the addition of a mesh in our particle system framework for the application of mesh algorithm visualization. In addition to the ability to implement mesh processing algorithms using our programming paradigm, the execution of these particle programs allows the user to observe the algorithm in action which is useful for the purposes of debugging, presentation and education.

The k-means algorithm is widely used for clustering data because of its ease of implementation [11], though a variety of variations exist [6, 22, 13, 15]. Since each particle acts independently we can implement cluster growth as a per-particle operation.

7.1. Implementation

We first create a dynamic mesh collection of particles as specified in the previous section that holds the input mesh. We will call this collection “Mesh.” Even though the mesh is dynamic, it does not change in this example so we delete the ParticleRepulsion behavior. Furthermore, since we are loading a mesh, there is no underlying implicit surface so we delete the ImplicitInterrogator attribute and the SurfaceAdhesion behavior.
We then create a second collection of particles called “Clusters” where each particle will represent a cluster and is positioned at its cluster center, the centroid of its seed face. The “Clusters” particles contain a ClusterMesh behavior that performs the clustering operation. This behavior begins with an initial collection of particles corresponding to the centroids of faces of “Mesh” that represent cluster centers.

The “Clusters” collection of particles also contains a “FaceCluster” attribute which stores a per-particle list of face indices corresponding to faces in the ParticleMesh of “Mesh.” These lists are initialized with the seed face for each particle in “Clusters.”

ClusterMesh proceeds by growing face clusters in the ParticleMesh held in “Mesh.” Once the clusters cover the entire mesh, ClusterMesh moves each of its particles, which correspond to cluster centers, to the centroid of the centermost face of the cluster it generated. ClusterMesh then repeats until the particle positions converge.

A particle shader in “Clusters” is responsible for setting the color of faces in the ParticleMesh attribute of “Mesh” to the correct cluster color. A particle shader in “Mesh” then draws the mesh with its cluster-colored faces and a Particle-Sphere particle shader in “Clusters” indicates the location of the cluster centers.

![Figure 7. (a) Pawn model with 254 faces, 3 clusters, 3 iterations. (b) Cow model with 5802 faces, 5 clusters, 10 iterations.](image)

7.2. Discussion

This particle system implementation of $k$-means clustering provides an intuitive algorithm visualization, allowing the user to click on any particle to expose the current internal variable settings of its attributes. Users can also interactively steer the partitioning, adding and deleting cluster centers through interaction with the “Cluster” particles.

We use our particle system as a visualization tool in this mesh partitioning algorithm, but the idea of using particle systems as a debugging tool is not new. Patricia Crossno and Edward Angel used this as a software engineering technique [5].

8. Conclusion

We have developed a particle system framework and language that allows users to rapidly create and reuse behaviors for a particle system. We have tested this framework with a number of applications that shows the reusability and extensibility due to the design of atomic behaviors.

In applications like singularity searching and artistic rendering for implicit surface, we develop new ways to interrogate properties of implicit surfaces with particles. We also introduce new ways of adapting mesh algorithms in particle systems.

Particle systems are needed to simulate natural phenomena like cloth untangling, water spray, snow, smoke, etc. This idea of iterative refinement implicitly resides in many of these algorithms, which our particle system can exploit. When developing and debugging these algorithms, our particle system helps to visualize each step of the simulation. This provides developers with direct insights to the simulation process. Therefore, our particle system serves as a powerful visualization framework and debugging tool.

8.1. Future Work

There are many applications that fit our generalized particle system model. For the applications we have implemented in this paper, for example, dynamic meshing, we would like to update our topology guaranteed polygonization for more complex implicit surfaces. Interactive topology guaranteed polygonization will improve the implicit surface modeling systems. Then we can have real-time polygonization instead of independent particles.

For the particle system itself, we would like to make the particle shading framework more flexible, for example, changing the behavior and shader at run time. This would requires run-time interpretation of a particle behavior language. For the moment, we are satisfied with the C++ language for specifying the details of particle attribute, behavior and shader functionality.

In recent years, we have seen rapid advances in the graphics hardware along with graphics programming language e.g vertex and pixel shader. This particle system framework holds promise as a conceptual tool for a possible new way to program vertex shaders.

References


