BLOBULATOR

Maik Flanagin and Pascal Rettig

6.837 Final project 1999

1. Abstract:

The purpose of our project was to create a particle system modeling amorphous objects, or blobs. These blobs, through forces on the particles behave and interact with their environment in an interesting way. The user is provided with a graphical interface that gives control over a number of parameters controlling the motion and interaction of the particles with both other particles and objects in the environment.

2. Introduction:

Tired of your old rendered 3D environment? Slime it! The Blobulator is an "immersive" interactive experience in which users can create realistic slime and ooze everything in sight. Imagine the non-stop fun as your unsuspecting models are slopped by a merciless, unrelenting wave of slime.

A little squirt or a crushing wave, messy or gooey, it?s up to you. Our Blobulator allows the user to decide the quantity and consistency of his slime. Furthermore, as an added bonus (time permitting) the Blobulator will also allow the user to create "intelligent" blobs. Blobs with brains will run amuck in your simulated world engulfing everything in sight as it makes a futile attempt to satisfy its voracious appetite.

3. Goals:

We had the following goals:

1. ***Simulate realistic per-particle physics.***

   We wanted to have the particles act with each other and geometric objects in the scene in a realistic manner.

2. ***Integration into a 3d scene viewer.***

   We wanted to create an application with a GUI which provided for easy interaction with the blob.

3. ***Simulate formless objects.***

   We wanted to correctly simulate the behavior and appearance of formless objects

4. ***Simulate blob translucency***

   We wanted to correctly simulate the appearance of a translucent blob (Not done)

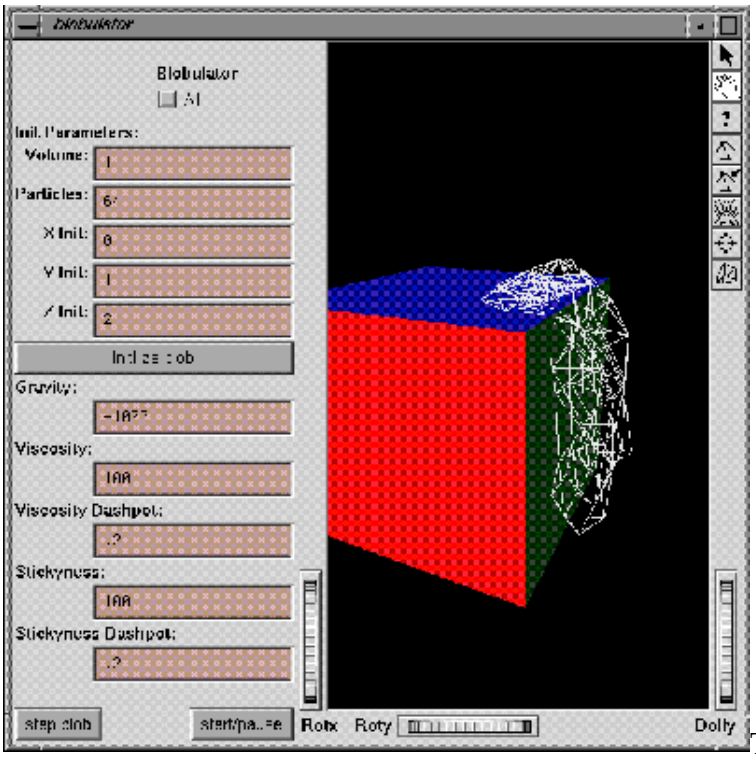
5. ***Accurate interaction with arbitrary 3d-models***

We wanted any Inventor scene to be fair game to "slime."

4. Achievements:

The blobulator can be broken down into the four distinct parts: the GUI, the interface between the GUI and the Blob, the Blob class, and the code to create the blob shell.

1. ***The GUI***



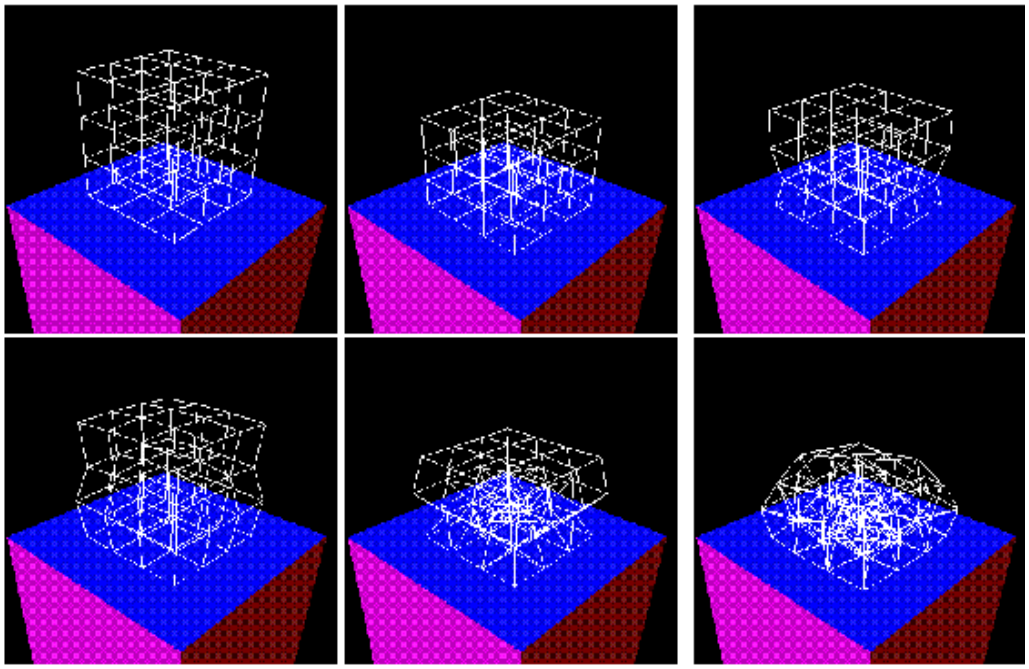
The GUI provides the user with an easy to use interface window that both takes in user input and displays the program output. The window can be functionally divided into two horizontal sections. The left section has a number of input fields that control the parameters of the blob, and the right section is an OpenInventor RenderArea widget that displays a rendering of the blob and its environment. The picture shows a capture of a preliminary blob window.

1. Input Area:

   The input area two main sections, the top section which controls blob initialization and the bottom section which controls step parameters.

   1. **Initilization section:**

      The top section has fives fields and an initialization button. The initilization button creates a blob as defined by the values of the five fields. It removes the current blob and creates a new one. The first field, volume controls the size of the initial blob. Upon creation the blob is a square with no particle-to-particle forces on any particle. Each dimension is the cube root of the volume long. The second field, Particles defines the approximate number of particles to be created in the new blob. This value is used as a rough estimate, because the blob is created as a square, the floor of the cube root of this number is actually used to determine the number of particles in each dimension. Entering a perfect cube (8, 27, 64, 125, 216, ?) will guarantee that that the exact number of particles will be used. The next three fields X Init, Y Init, Z Init, define the starting position of the center of the square blob in 3d space.

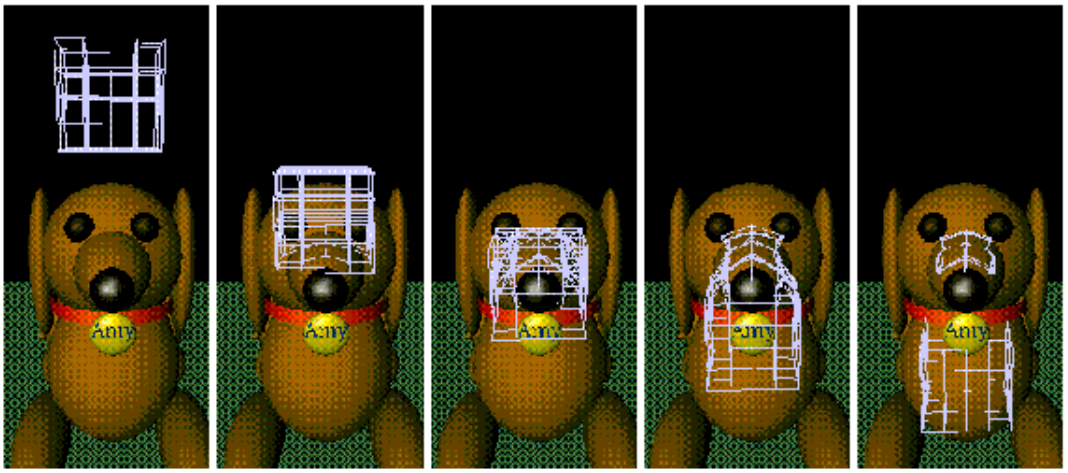


square form that the blob is initialized to is generally not very stable, as interaction with other surfaces demonstrates. Shown is a sequence of pictures showing how the blob reacts to being set as a square on a flat object. The blob reacts and settles into a form similar to blobs in real life.

2. Step Section

The step section has an additional five fields and two buttons. The leftmost button "step blob" takes one small step in time an displays the results. The right button "start / pause" begins and pauses a loop where step is called repeatedly. The five fields control the real time parameters of the blob. While the initialization parameters only take effect when the blob is initialized, the step parameters are recalculated every time either "step blob" is pressed or the animation is restarted with the "start / pause" button.

The first parameter, gravity, controls the acceleration due to gravity. The Z direction is the direction on which gravity acts. A positive value will move the blob in the positive Z direction, while a negative value will move the blob in the negative Z direction.



The second and third parameters, viscosity and viscosity dashpot, control how the particles interact with each other. The viscosity value acts like a particle to particle spring constant, with a higher value meaning a higher stiffness. The viscosity dashpot acts like a particle to particle mechanical resister (or dashpot), that dissipates energy between the two particles. A high viscosity will result in a blob that sticks together under more stress while a low viscosity will result in a blob that tends to fall apart under strain. Shown is a series of frames with a low viscosity value.

The third and fourth parameters, stickyness and stickyness dashpot, behave similarly the second and third, except they control the interaction from a particle to an object. When a particle comes into contact with an object, it sticks to it. How high the stickyness value controls how big the spring constant attaching the particle to the object is. The dashpot works similarly, only controlling the energy loss of the object.

2. Render Area

The render area, as mentioned before is a RenderArea widget taken from the InventerApi. It behaves similarly to the examiner application, accepting input to control the viewpoint of the camera.

2. The Interface Functions:

Contained in the files Interface.C and Interface.h, The interface bridges the GUI to the blob model. It has procedures to initialize blob scenes specified by the user input file and conains the instance of the "Blob" class. It steps through an animation of the sequence by first updating the blob?s state and then making the appropriate adjustments to the scene graph through calls to makeblob. Finally interface updates the picture in the user interface based on the updated scene graph.

The work horse function of interface is MakeSoBlob, which builds Inventor objects out of the blob representations. It is called during every iteration of step. There are two versions of makeSoBlob; one uses the Blob?s particles and connections to form a mesh of line segments. The other places a shell around the blob to give a realistic look. As seen in the screen captures, the blob the former represents the blob through the connections of the particles to each other, creating a wire mesh. The latter is not yet completed as of this writing. The details of the latter will be discussed in another section.

3. The Blob Class:

The blob class, define in Blob.h and Blob.C, consists primarily of an array of particles and functions to act on those particles and return their status.

1. Blob Initialization

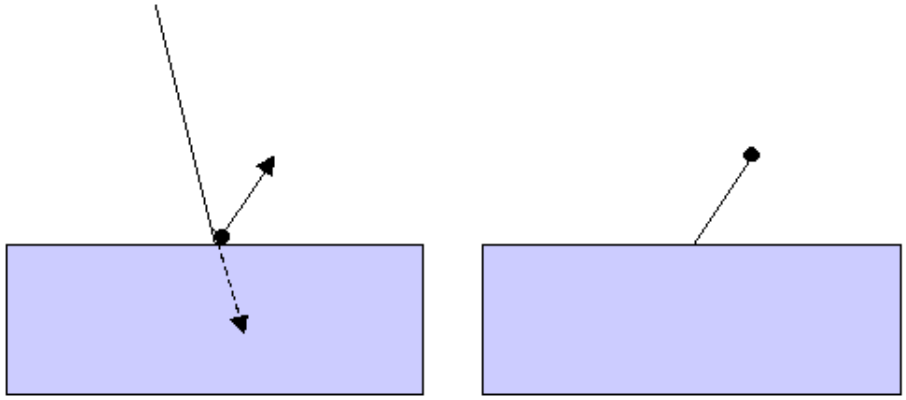
   Blob provides one public function to initialize a new Blob, Blob::Init. This function takes in an initial position, volume, rough number of control points, and a Octree used in object collision. It creates an array of particles (deleting any pre-existing particles), and places them in a 3d cube. It updates the bonds between the particles so that mesh looks like a normal cube.

2. Blob Step

   Blob::Step updates the particles positions by iterating through them 3 times: once to update the forces, once update velocity and position, and once to update bonds.

   The forces are updated through pUpdateForces, which takes in a particle and calculates the forces on it, which include gravity, particle-to-particle connections and particle-to-surface connections. The velocity and position are updated by pUpdateMove, which takes in a particle and moves it. pUpdateMove also checks for collisions with geometric surfaces. Collision detection will be discussed later. pUpdateBonds would iterate through the distances between particles to determine where connections should be made or broken.

3. Blob Collisions



It was essential in creating the Blobulator that collisions between particles and surface geometry be accurately handled. Particles flowing into surfaces would be a inconsistency that would ruin the entire system. The way we decided to handle collisions was by pulling in the ray casting code from assignment 6. A ray would be cast in the direction of the velocity vector of each particle, and the first collision would be noted. If this collision were at a smaller length away than the length of the incremental position, a collision would be considered to have taken place and a reflected direction vector would be calculated. The particle would then be moved to the surface of the object and the incremental position subtracted by the length already traveled to reach the object. The velocity would change direction to the new direction vector, the collision point would be noted, and the particle considered attached to the object at that point. The whole process would then recurse, with the collision point considered the new starting position, and a new incremental position vector used, the old one reflected and diminished by the distance already traveled. Thus if over the course of one step the particle were to come into contact with two ore more geometric objects its updating would be handled correctly. Lastly, when the incremental position and velocity are reflected, a fraction of each is subtracted to exhibit elastic collision losses and to prevent the algorithm from recursing forever.

4. Blob GetFrame

This function returns the geometry of the blob to the interface functions.

It returns a linked-list so that each connected portion of the blob can be correctly handled and shelled.

4. The Blob Shell:

To give the blob a physical appearance, it is necessary to define it as a solid shape, specified by the outermost points and edges in the blob?s particle system. However, finding these

points and edges requires it?s own special algorithm, which presented a significant challenge in our project and still to this day vexes us with bugs.

1. Overview:

   By using a combination of a convex hull algorithm and a single-source shortest path algorithm, it is possible to precisely determine which points and edges define the blob?s shell. Given a set of vertices, a convex hull algorithm determines the shape formed by the outermost points and edges. However, blobs are not necessarily convex. However, the connections between the particles in the blob?s model define a set of possible outside edges that can be exploited to form the shell of the blob through the following algorithm:

   ConcaveHull(list of vertices, list of constraint edges) {

   {convex vertices, convex edges, convex faces}=ConvexHull(list of vertices)

   for each face in convex faces

   for each edge in face

   if (edge is not an element of list of constraint edges)

   v=vertex that the other two edges in face share

   path=ShortestPath(edge.endpoint1,edge.endpoint2, list of constraint edges)

   vertex1=the first vertex in path

   vertex2=the second vertex in path

   do

   add vertex to convex vertices

   add the edge from vertex to endpoint2 to convex edges

   add the edge from v to vertex2 to convex edges

   add the face of (v,vertex1, vertex2) to faces

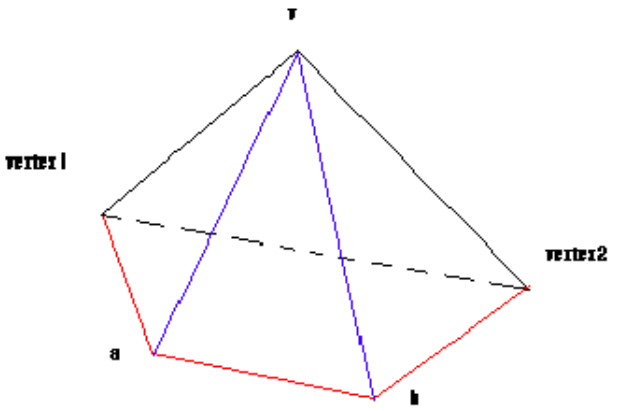
   vertex1=vertex2;

   vertex2=next vertex in path

   while vertex2 is not NULL

return convex faces

The Concave algorithm returns the faces representing the blob?s shell. Initially, it runs a convex hull algorithm. After which, it checks every edge against the list of constraint edges, which is a list of connections in a given blob. If there is an edge in the convex hull that?s not in our list of constraint edges, then the given blob has a concavity and we must now run an error-correcting sub-routine.



In the figure about, (v,vertex1,vertex2) form as face on the convex hull, but edge(vertex1,vertex2) is not and edge in the blob and therefore, not in the constraint list.

the edges through a and b are the shortest path from vertex1 to vertex2. We add those edges along with an edge from vertex v to form new faces to make a concavity

To correct for a concavity, concave hull calls a single-source shortest path algorithm over the list of constraint edges. In our implementation, we used the Bellman-Ford algorithm, since it is fairly simple to implement. The shortest path algorithm will return a list of vertices from the destination point to the start point, which we can use to add new edges to our edge list.

To build our new faces, we increment over the list of vertices in our path from vertex 1 to vertex 2. Since our faces are triangles, we need to find three edges to define them. One edge is the edge from one vertex to the next in the path. The second edge is the edge from the vertex that would?ve formed the triangle face if a path from vertex 1 to vertex 2 would?ve existed to our new vertex. The third edge is the second edge we formed in the last iteration or, in the first iteration, the boundary from v to vertex2. Since we loop over every face we repeat this process on at least two different "v" so there are no "holes" in our final shape.

2. Convex hull algorithm:

The convex hull algorithm takes a set of points and returns the polyhedron's faces, edges, and vertices. I chose Joseph O?Rourke?s implementation of the convex hull algorithm for its relative simplicity. The key to it?s success lies in the computation of visibility, that is, given a vertex v and a face f, f is visible to v if f?s normal points out towards v. The normal of a face are determined by the right hand rule.

Initially it chooses three non-collinear points and builds two faces out of them. One face orders the points clockwise, while the other orders them counterclockwise. That way, normals are pointing in both directions. Convex hull iterates over each remaining vertex. If no faces are visible to the vertex, then it is inside of the hull and marked. Next, the algorithm iterates over each edge. If both faces adjacent to the edge are visible to the vertex, the edge is marked for deletion. If just one face is visible to the vertex, then a new face is constructed with the edge and the vertex.

3. Bellman-Ford algorithm

The Bellman Ford algorithm is used for determining single-source shortest paths. Like all path-finding algorithms, it relies on relaxation techniques. The algorithm keeps an array of predecessors and a corresponding array of upper-bound distances to these predecessors in which each array slot maps to a node in the given graph.

Initially, the distances are set to infinity and the predecessors are set to zero. Bellman-Ford sets the distance to the source to be zero and iterates over all edges for every other vertex and checks whether a given edge can improve the distance between a vertex and its predecessor.

When the Bellman-Ford algorithm completes, the path from a to b can be found by looking at the successive predecessors of b until a is reached, assuming a was given as the source.

5. Individual Contributions:

While the project was for the most part a team effort, Maik Flanagin worked primarily on the interface functions and the outer shell, and Pascal Rettig worked on the GUI and the Blob class.

6. Lessons Learned

The project provided us with a number of interesting learning opportunities. First, working in unison on a single coding project provided a good opportunity to learn how to share filed. Second working on a deadline with weekly status meetings provided a change from the average MIT all at the last minute philosophy. In terms of design and coding, we ran into a number of hurdles. As neither of us knew how to program anything in X-Windows, we were forced to go through a significant learning curve to bring up a window with our desired interface. Widget placement by hand proved to be quite an ordeal, and in the future we would probably prefer to use a resource file or GUI designer (The code in main is a testament to why). For particle-object collisions, pulling in the ray casting material to initialize and work properly, while not a lot of code, nevertheless

proved to be tricky. Working with a particle system in general was new to both of us. Getting the particles not to fly every which way proved to be more difficult then initially expected. Since we used Inventor to do all the rendering of the blob and scene objects, we had to learn the inventor API in order to create scenes and objects and animate those scenes. Lastly, the surface determination proved to be fairly difficult and algorithm intensive. While it is currently not in working order, hopefully by the presentation we will have a properly skinned blob. An interesting the we found out playing with the Blobulator was that some fairly interesting behavior could be gotten with a fairly small number of particles, coming in we expected that hundreds or maybe even thousands of points would be required to create an interesting blob.

7. Acknowledgements

We would like to thank our TA advisor, Damian for pointing out potential pitfalls and solutions. We used code from the ray casting problem set in object intersection determination. In addition we used Introduction to Algorithms and Joe O?Rourke

Computational Geometry in C in surface determination.

8. Bibliography

Cormen, Thomas, Leierson, Charles, and Rivest, Ronald. Introduction to Algorithms. Cambridge, Massachusetts: The MIT Press, 1998.

Hearn, Donald and Baker, M. Pauline. Computer Graphics C Version. New Jersey: Prentice Hall, 1997.

O?Rourke, Joe. "Collections: Code from Computaional Geometry in C." http://www.geom.umn.edu/software/cglist/libs.html#cgic. 1999