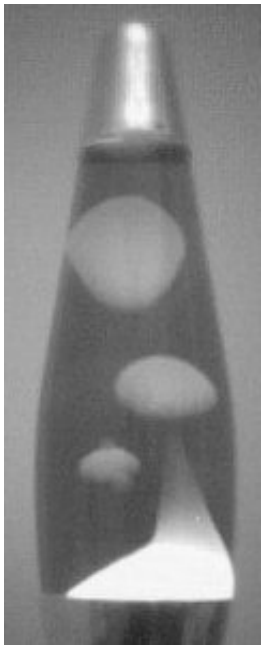
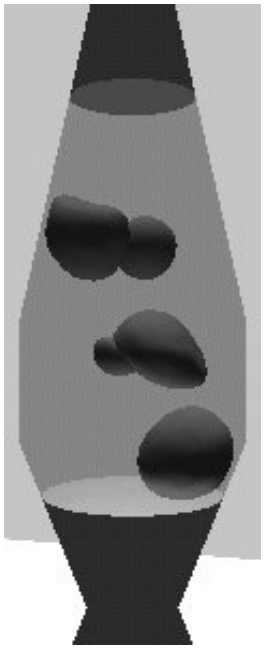
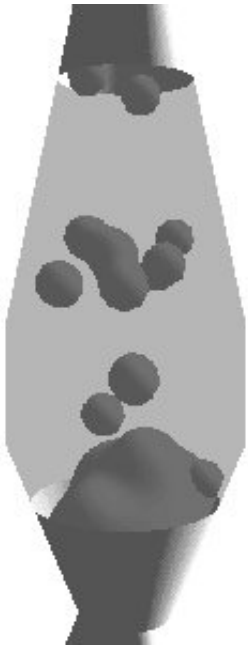
A 3D Lava Lamp Simulation

by Kim Randell, Joint Maths and Computing, Imperial College

Project Supervisor: Duncan Gillies
Second Marker: Guang-Zhong Yang

Abstract

This project attempts to create realistic 3D animation of a lava lamp in real-time, comprising amorphous wax blobs moving and interacting within a bottle of liquid, powered by a light-bulb underneath the bottle. It combines and adapts a number of existing techniques to achieve new results.

Acknowledgements

Mathmos (http://www.mathmos.com) - for creating the original lava lamps
nVidia, whose technical demos inspired me to do this project
Richard Baker (http://www.theculture.org/rich) - for help with molecular dynamics
Peter Hall, for getting me started on OpenGL
And of course Duncan Gillies, for lots of friendly advice and stopping me from worrying myself into oblivion over the project

Introduction

Motivation

In the computer games and digital entertainment industry, there has been a lot of work into developing fast algorithms for approximate but aesthetically pleasing simulations of physical effects. Water ripples, cloth movement, inverse kinematic movement, and particle weather effects have all been attempted, usually with some form of finite element analysis, each to a varying degree of success. I decided to pursue a similar technique of simulation for a phenomenon which has as yet not been developed in such a way. I chose the lava lamp because they are fascinating objects, moving hypnotically in a vast and unpredictable number of configurations (so much so that they are sometimes used as random number generators). I wanted to find out whether it was possible to replicate the organic and amorphous motion on computers, which are typically better at more discrete, rigid motion. The complex interaction between separate wax blobs and the liquid they are suspended in ensure that the problem is non-trivial.

Scope

The project explores two areas of computing: physical simulation, and graphical techniques. The physical simulation takes into account intermolecular forces, gravity, and the effect of heat on density and thus relative pressure. Due the the nature of the hardware I am working with (medium specification PCs), I will not be modelling the molecules in detail, but rather attempting some approximation of their movements. The macroscopic nature of the simulation means that simple observation of the lamp's behaviour is just as important as knowledge of what's happening at the molecular level. The main graphical techniques used are free-form polygonal meshes and iso-surfaces, along with more basic concepts such as alpha blending and lighting using normals. Low level rendering is handled by a library, so I do not have to consider pixel level graphics operations.

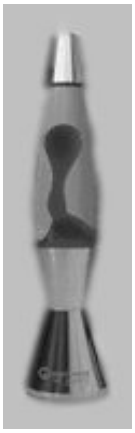
Background

Overview

A real lava lamp itself consists of:

- A base unit, which can be metal or plastic. They are available in a range of shapes, which do not affect the operation of the lamp but are purely aesthetic. They can be cylindrical, in the shape of one inverted truncated cone on top of another, or smoothly sculpted. The most important component is the single lightbulb pointing upwards, which provides light and heat.
- A glass bottle containing wax and a liquid in which the wax is immiscible. It is tall and thin; some models bulge towards the top or bottom, and taper at the opposite end, whereas others are straight cylinders.
- A small metal cap covering the top of the bottle.

Some real lava lamp designs:



Astro Baby Lava Lamp

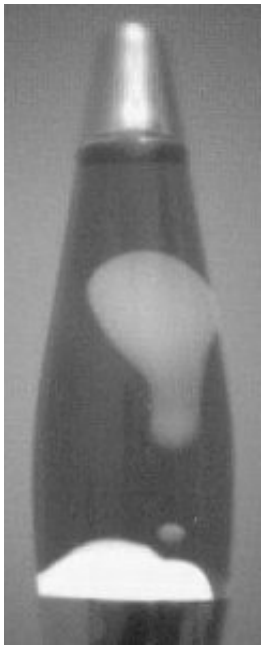
Astro Lava Lamp

Jet Lava Lamp

How it works

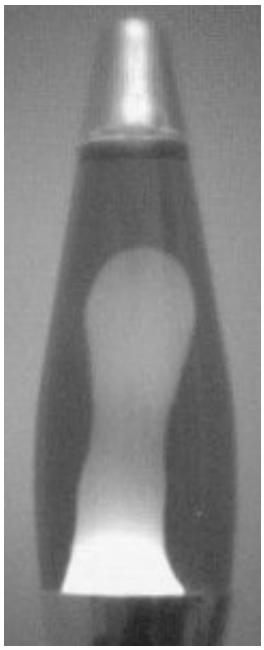
For my program to successfully simulate a lava lamp, it should emulate the following phenomena:

- **Pooling** - The wax collects in a pool at the bottom of the bottle. This pool is not flat as a similarly sized pool of water in the bottom of a bottle would be; it bulges significantly towards the center and slowly ripples as wax enters and leaves it.



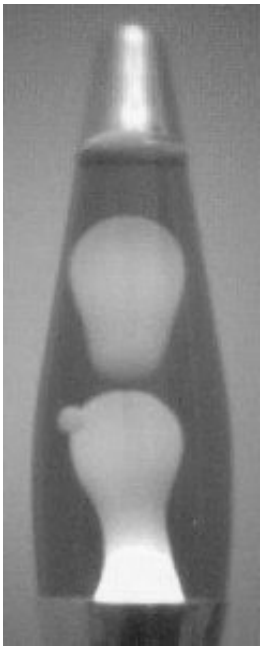
The bulging pool at the bottom of the lamp

- **Expansion and ascent** - The wax is heated by the the lightbulb, and expands accordingly. Once the wax has a low enough density, it rises from the pool, forming an unsteady column. This is close to central, due to the position of the light-bulb; the wax towards the perimeter is not heated as much and does not rise. The column writhes back and forth as the hot wax flows up and down inside it. The diameter of the column is different at different heights and is constantly changing.



A column of wax forming

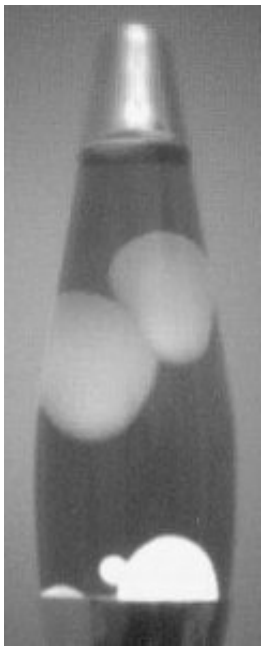
- **Splitting** - Eventually the column breaks off into one or more distinct blobs. The top of the column forms one large blob, and sometimes the rest of the column can break off into smaller ones. The remainder of the column sinks back down into the pool. These blob generated by this process are approximately spherical. In a similar fashion, the larger blobs formed by this process can stretch out vertically and breaks into more blobs. First they stretch into a vertical hourglass shape, with the 'neck' in the middle gradually thinning to a cone, with the large end at the top and the small at the bottom. When the piece at the bottom breaks off, the point of the cone may also generate small blobs.



The head of a column of wax breaking off

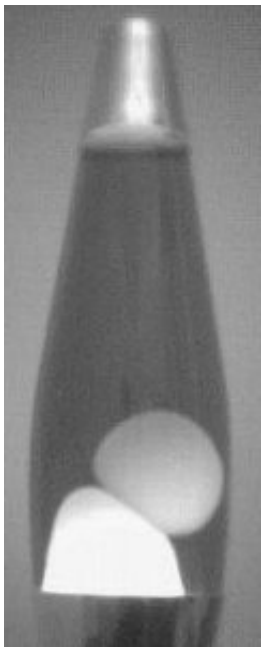
A blob splitting in two

- **Collision** - Once splitting has occurred, surface tension ensures that the wax does not re-combine until it reaches the pool at the bottom again. Instead, blobs deform when they collide with each other, with the impact propogating as a low frequency wave across the surface. It slowly settles back into a flattened near-spherical shape afterwards. The collisions are close to frictionless, and the kinetic enery of the blobs is converted to deformation.



Two blobs colliding

- **Cooling and descent** - As the wax rises, it cools, and when it becomes dense enough it falls back to the pool. Resistance by the surrounding liquid squashes the sphere into a more flattened shape. It has usually reached nearly if not all the way to the top of the bottle by the time its velocity changes direction.



A blob hitting the pool as it descends; it will not merge again until it is hot enough

Existing Implementations

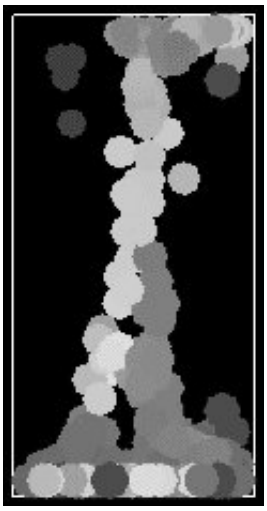
There have been very few attempts at lava lamp simulation. Some lava lamp programs are simply hand-animated or photographed from real lamps, which can look good but are not true simulations [2].

The "i-mate" lava lamp, which uses a set of animation frames

There is a Linux screensaver, which has 2D blobs that simply pass through each other rather than colliding. It is intended more as a fun process viewer than a true simulation of lava lamp behaviour [3].

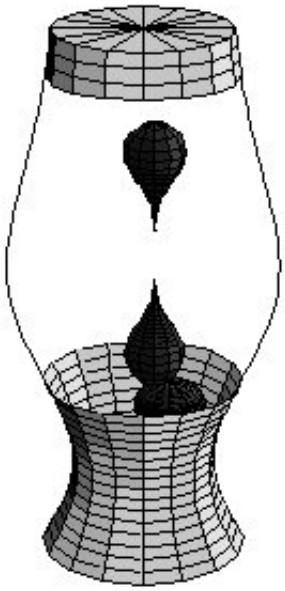
The Linux screensaver/process viewer, LavaPS

One computing project features an extremely low level simulation. It uses realistic smooth particle hydrodyyamics to simulate the motion of individual molecules, giving and interesting insight into the motion at a microscopic detail, but fails to give a decent impression of the overall movement of the wax [4].



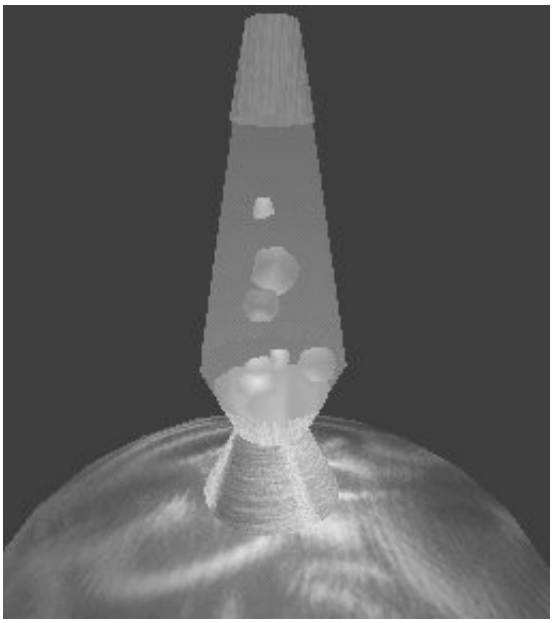
A low level particle simulation

A Mathematica program attempts to show blobs splitting through a parametric plot, but makes no effort to have collisions, pooling, or even more than three blobs at a time [5].



A simple Mathematica graph

One program has appeared since this project was started, which appears to try a similar technique to my mesh-based simulation; each blob is a deformable mesh, which bounces off other blobs and the walls of the bottle. However, their collision detection, while accurate, is so slow that the frame-rate drops below 1 fps for any more than a few high polygon blobs. [6].



Till Gerken and Olaf Roß's Lava Lamp

The best animation features very liquid looking blobs, but their collision is not handled properly, so that they simply merge and break away with no consideration of the real physics. This was seen on a television programme and unfortunately no further information has been found on it.

Platform and language choice

The programming language chosen was C, as due to prior experience with the language. It is both fast and flexible, with good compilers available on most platforms. Microsoft's Visual Studio application is a very good development tool, so the code is mostly written and compiled using it, although it does not use any Microsoft specific libraries and is portable to Linux and other platforms. For the Linux versions, the GNU C++ compiler g++ was used.

Implementation

OpenGL

The scope of the project is the high level graphics work, rather than low-level rendering. For this reason OpenGL is used. It is a fast, portable graphics library in C, which allowed the project to concentrate on the simulation aspect without worrying about the specifics of polygonal rendering. OpenGL provides an interface to whatever 3D hardware is present on the machine, handling polygonal rendering, the z-buffer, texturing, and multiple light-sources. Complementing the rendering interface is GLUT, the GL Utility Toolkit, which can handle input from the keyboard and mouse, and idle function callbacks. OpenGL was chosen over Microsoft's competitor, DirectX, due to its portability and ease of use. DirectX is object oriented and is linked with the Microsoft Foundation Classes, and the interface often changes from with each version, whereas OpenGL has stayed relatively constant.

The key to understanding OpenGL is knowing that it is a state machine. This means that you can put it into a certain state or mode, such as the colour, position, or polygon drawing mode, which affects all subsequent operations until you change or remove it. The primary aspects of OpenGL are the buffers, the viewing and projection matrices, and the state variables. The buffers contain the image as shown on screen, and can be cleared, swapped (for double-buffering), or manipulated directly. The viewing matrix is the transformation which is applied to everything rendered; it can be pushed and popped on a stack so that hierarchical structures can be built. The variables handle colour, lighting information, and rendering styles. For more information on how OpenGL works, the OpenGL Programming Guide is recommended [1].

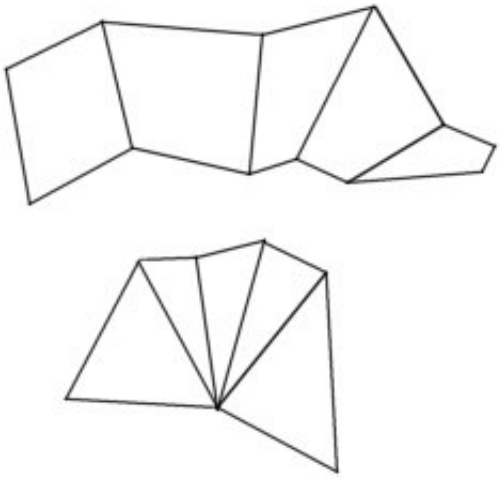
To view the model from any angle requires a simple camera system. OpenGL handles camera movement by moving the scene relative to the viewer. The camera always points directly towards the centre of the lamp, but can be zoomed in and out (by translating closer and further from the centre), and rotated around both the y and x axes (by using OpenGL's rotate functions). The following sequence of commands demonstrate how the camera is positioned.

`glLoadIdentity ();`	We always start with the identity viewing matrix
`glTranslatef (0, 0, -zoom);`	Bring the camera out to the desired distance from the centre
`glRotatef(x_angle, 1.0, 0.0, 0.0);`	This rotates the camera around x-axis so that we look up or down towards the lamp
`glRotatef(y_angle, 0.0, 1.0, 0.0);`	This rotates the camera around the y axis, so that we can observe different sides of the lamp
`glTranslatef(LAMP_CENTRE_X, LAMP_BOTTOM+7, LAMP_CENTRE_Z);`	Point the camera at the centre of the lamp

Note that because we are conceptually moving the scene rather than the camera, the positioning translations and rotations must be in reverse order. The resulting overall matrix will be applied to all primitives rendered, and so the lamp will appear in the correct position.

Lamp model

The static sections of the lamp (those made from metal and glass) are trivial to model, especially due to their rotational symmetry. they are bilt out of two OpenGL primitives: the quad strip and the triangle fan. A quad strip is a series of quads (polygons with four vertices), with adjacent quads sharing two points. A triangle fan is a series of triangles, all sharing one central point, and with adjacent triangles sharing one more point which is not the central one.



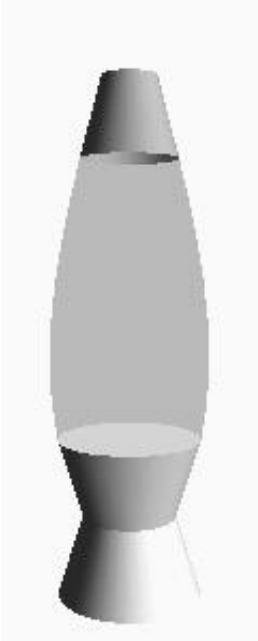
A quad strip and triangle fan

The ’Astro Baby’ model (see overview) is chosen as an example of a lamp model, but the other types can all be constructed using similar methods. The bottom is constructed of two grey quad strips configured as truncated cones; similarly, the top is another truncated cone topped with a triangle fan disc. Sine and cosine tables were stored in memory to ensure speed in calculating the rotation of the strips. The code iterates through deg (the number of degrees the strip is rotated by) from 0 to 360, and plots the following vertices, for example:

Bottom truncated cone quad strip:
glVertex3f(sine[deg]*radius1, 0, cosine[deg]*radius1);
glVertex3f(sine[deg]*radius2, base/2, cosine[deg]*radius2);
(where radius1 and radius2 are constants with radius1 > radius2, and base is the height of the bottom section)
Disc on top of lid triangle fan:
glVertex3f(0, base+glass+top, 0); for the central vertex
glVertex3f(sine[deg]*radius, base+glass+top, cosine[deg]*radius); for the outer vertices

The glass requires a little more thought, as obviously it has to be translucent. This is done using OpenGL’s built in alpha-blending. Ideally we would render the background, then the glass at the back, then the contents of the bottle, then the glass at the bottle, but it is much easier to simply render all of the glass last, without losing much, if any, visual quality. The glass is given the colour of the liquid inside, as it completely fills the bottle. For the bulge and taper of certain models, a function of radius against height is defined; this is also used for the collision detection later on. This function can be adapted to handle any type of lamp: a cylindrical bottle will return a constant radius against height, whereas a curved one can be described by a polynomial. A quadratic or cubic (polynomials of degree 2 and 3) suffice for the real models seen in the overview. To render the glass given this function, a series of quad strips wrapping around the lamp are used once more, with a strip for each section of

height. To give a better impression of a curved surface, more strips have to be used, although only one is necessary for the cylindrical lamp.



The finished lamp model

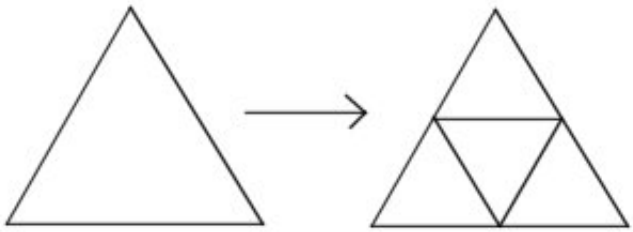
Mesh based simulation

Overview

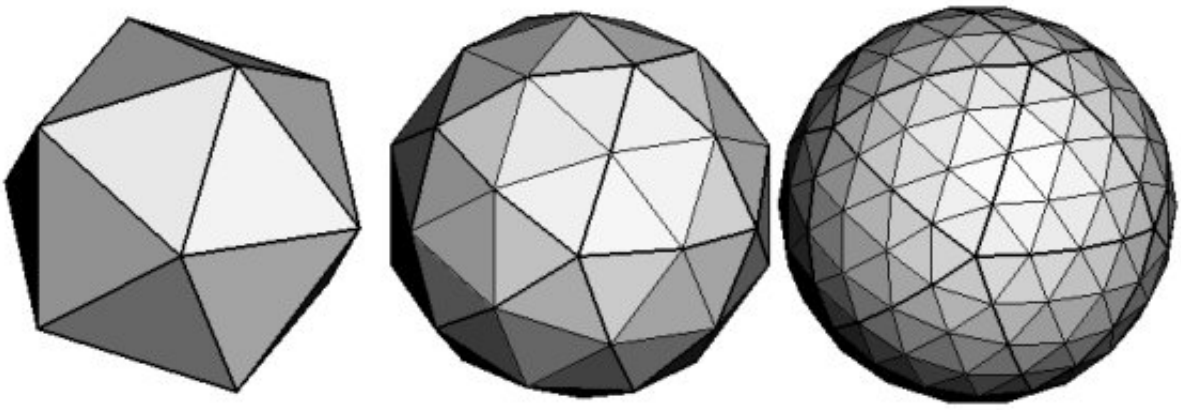
The mesh based simulation treats each blob as a primitive unit. Each blob is composed of a number of polygons, initially approximating a sphere. By moving the vertices of these polygons, the blob is animated; thus the physics of the wax is dealt with at the vertex level, by treating each as a particle. This type of particle simulation is called Finite Element Analysis. OpenGL is specifically designed for polygon rendering, and as such this type of simulation suits the library well.

Blob generation

The standard method for generating a spherical surface is to subdivide a regular icosahedron, which is a very approximate sphere composed of twenty equilateral triangles. To subdivide this initial model, each triangle is split into four new equilateral triangles. The vertices are then normalised to the distance of the sphere's desired radius from the centre. This process can be repeated as many times as is required to get the desired number of polygons and smoothness; around 2 or 3 subdivisions were sufficient for pleasing effects. The number of polygons in the surface, given n subdivisions, is given by the formula: $20 * 4^n$, so for 2 there are 320 faces and for 3 there are 1280 [1].



Subdivision of an equilateral triangle into four smaller equilateral triangles



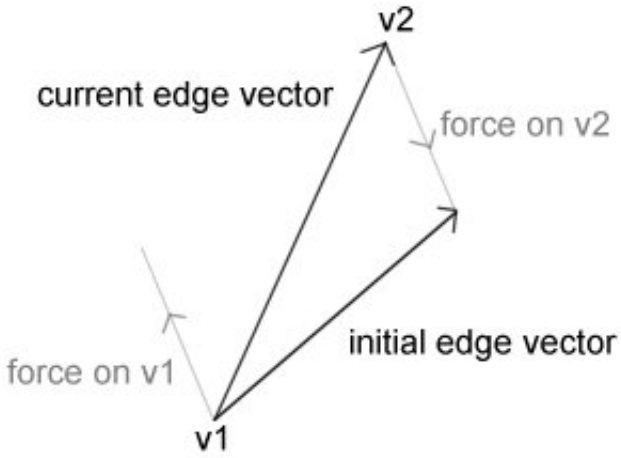
Subdivision of an icosahedron into an approximate sphere (image from [1])

The blob is stored as an array of vertices, an array of faces, given by the indices of the vertices in the vertex array, and an array of edges linking the vertices, also given by the indices (normally this is not required for a model if the faces are defined, but the reason for storing it will become clear in the blob animation section. The normals at each vertex are computed using the standard technique of averaging the normals of every face the vertex belongs to (by taking the cross-product of two edges of the face). Other information stored in the blob structure is the initial radius, the temperature, the velocities of each vertex, the initial edge vectors and lengths, the number of edges, the number of faces, and variables for collision detection.

Blob animation

To realistically simulate the surface of the blob, there has to be a mechanism for propogating the deformation across the surface as some sort of wave. This is achieved by giving each vertex its own velocity (and thus momentum), and giving them feedback across the edges. The first type of feedback is a simple spring effect for every edge; a force proportional to the distance from the equilibrium length is applied in the direction of the edge vector. This spring effect causes each vertex to ’shunt’ the next along when it is moved, resulting in a wave moving across the surface. It also allows blobs to push against one another. The iteration through the edges is reasonably fast, and thus more suited to real-time applications than more complex optimisation routines, while still producing good results.

The second type of feedback is required to keep the spherical shape, and to allow for propogation of movement orthogonal to the edges. When the blob is generated, the direction of the edges is recorded in an array of vectors. The animation algorithm finds the current direction of the edge, and subtracts the initial direction from it. This gives a vector of acceleration to apply to the vertices at either end of the edge, which attempts to spring them back towards their original relative positions.



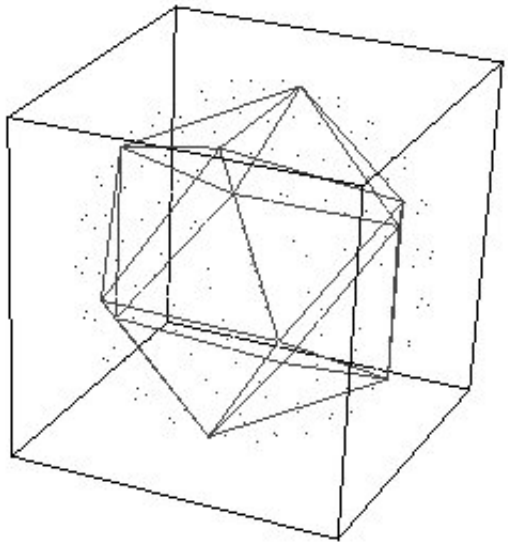
2D demonstration of the second edge spring.

The combination of these two edge springs, lengthwise and directional, gives a decent approximation of the waves across the surface of a blob. The last component of animation that must be considered is the rising and falling in accordance with the temperature of the blob. To this end, each vertex is given a temperature, and is accelerated proportionally to this value. The temperature decreases to simulate heat being given off to the surrounding liquid, and is raised when the vertex is near the bulb at the bottom of the lamp.

Blob collision

3D collision for an amorphous, possibly concave object is particularly CPU intensive. To correctly ascertain whether a point is inside a polygonal surface or not, we must check the point against every single polygon. To check whether a point is in front of or behind the plane given by any one polygon, we use the normal of the polygon. Take the dot-product of the vector from any vertex of the polygon to the point being checked with the normal; if this value is positive then it is outside, otherwise it is inside. As noted in the generation, we can be working with up to 1280 polygons, and so it is infeasible to perform these checks in full. Optimisation and approximation is necessary.

Firstly, a simple bounding box check is made before any more complex operations are performed; the maximum and minimum x, y, and z values of the vertices is taken, giving a cuboid volume. If the point is not in this volume then it definitely isn’t inside the surface. If it is inside the volume, then further checks must be performed. A decent approximation to the shape of the surface is the icosahedron from which it was derived. By using this form, only twenty normal checks have to be performed.



A blob in a bounding box, with the internal icosahedron visible and surface represented by points

Once it is determined that a vertex of one blob is inside another blob, we have to push it out again. To push it in the right direction, the position of the vertex relative to the colliding blob must be calculated. The centre of the blob is found by taking the mean position of all vertices; then it can be split into twenty tetrahedrons, where each takes three vertices from the icosahedron and the last is the centre point. Using more normal checks, it is determined which of these tetrahedrons the vertex lies in. The vertex can then be pushed in the direction of the normal of the corresponding face of the icosahedron (by increasing the velocity in that direction).

Blob merging and splitting

Unfortunately, merging and splitting is the major weakness of the mesh based simulation. There is no simple way of breaking the near-spherical mesh into two or more separate sections which retain the same properties as the parent blob. For this reason, we can no longer treat all blobs as primitive units; instead, they must be composed of several smaller pieces than can be split off when required. This is rather incompatible with the concept of the blob as a single polygon surface, and so a different approach must be taken. For similar reasons, the pool at the bottom, from which all the blobs rise out of, is very difficult to simulate using the mesh method. It could theoretically be done using a polygonal surface, rather than a sphere, which undulates vertically according to the heat below and blobs impacting above, but this would still not be able to easily rise into a column and generate new blobs.

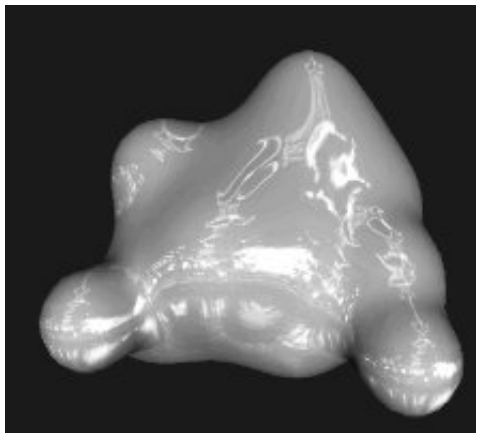
Particle based simulation

Overview

Implementing the mesh based simulation, it was found that taking blobs as the units of simulation restricted the possibility of splitting them. The particle based simulation solves this problem by making smaller particles the units, which can then combine into blobs. These particles are analogous to large versions of molecules comprising the wax; their positions in the blob determine its shape. By comparing them to molecules, they can be treated similarly to the real physics equations regarding interaction of molecules in a liquid. The disadvantage of this approach is that simply rendering points or spheres for each particle will not give the smooth surface required. A different rendering technique must be considered.

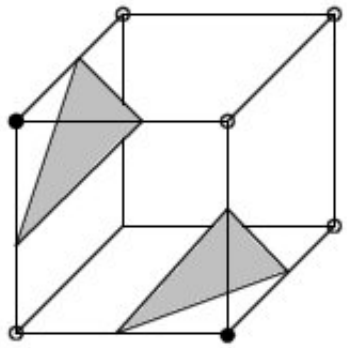
Isosurfaces and metaballs

Given a continuous scalar function in 3D space, we can take all points at which the function is equal to a given threshold. The layer that of all these points form is called an iso-surface. To visualise this, consider the 2D case. It is equivalent to contour lines, for instance, which are iso-lines of the function of height on a top down map of a landscape. One particular application of iso-surfaces is the meta-balls technique. First we need a function which falls off smoothly as radius increases, typically $1/r^2$ (equivalent to electrostatic or gravity potential), but which also can be modified so that it hits zero at a certain distance, rather than trailing off asymptotically [10]. The overall 3D function is given by the sum of the contributions from all the balls. When a threshold is taken on this particular function, we see balls which smoothly merge together when they are close enough. This is similar to the behaviour we want for the blobs, although care must be taken not to have distinct blobs merging into each other when they should not (more on this later).



A smooth, reflective surface built from metaballs

Unfortunately, translating the thresholded function into a form which is possible for OpenGL to render is not trivial. For the best quality, ray-tracing should be used, but this is not feasible in real-time. Two other algorithms are available: marching cubes and active contours. The marching cubes technique discretises the space to be rendered into a grid of cubes. It then traverses the grid, checking each cube to see if a polygon should be rendered within it. The active contour technique defines a surface, then optimises it to fit around the surface at the threshold level. As the marching cubes algorithm has already been developed and optimised, it was the suitable choice.



An example of the polygonisation of a cube grid cell. The open circles represent vertices outside the threshold and the closed represent vertices inside the threshold.

The check to be performed at each cube involves evaluating the function at each vertex of the cube. If the values are all greater than or all smaller than the threshold, then the isosurface does not intersect the cube, and we can move onto the next. If this is not the case, then the cube conforms to one of a 256 combinations of vertices inside or outside the surface. For each of these cases, there is an appropriate polygonal surface inside the cube equivalent to the isosurface [9]. The specific position of the polygons is determined by interpolating linearly along the edges according to the scalar values at the vertices. The normals of the vertices can be found by taking the differential of the overall function at the appropriate point.

Two adaptations of the marching cubes algorithm were chosen. Firstly, the marching tetrahedron adaptation splits each cube into 6 tetrahedrons to polygonise. This is slightly slower, but gets rid of the ambiguous case present in marching cubes, and the number of different cases to be considered is reduced to eight [9]. The other adaptation is an optimisation. Iterating through every single cell in the grid is expensive, so the optimisation only picks out cells likely to contain the surface. A ray is cast from a particle until it crosses the threshold. This cell is then marked for checking, along with all neighbouring cells. The cell checking function extends recursively. If a cell is found to be empty or already checked, then it is ignored. If it contains part of the surface, then all six neighbours of the cell are checked. This continues until all cells containing this section of the surface are checked. The algorithm iterates through all particles repeating this. If a particle belongs to part of a surface which has been already rendered, then it will not waste any time re-rendering, as the relevant cells will have been marked as already checked. If it has not been rendered yet, the new surface will be mapped out.

Blobs

A blob is defined as a non-zero set of particles. The maximum number of blobs is the total number of particles, because we can have blobs of size one, and the maximum size of a blob is also the total number of particles, so they are stored as a 2D array of size #particles * #particles. There is a separate array storing the size of each blob.

To split a blob, it must be determined which particles are still in the original blob and which are in the new blob(s). Obviously a blob of size 1 cannot be split, so these are ignored. First a table of which pairs of particles are linked; if their relative distance is smaller than a defined split distance, then the corresponding entry in the connection 2D array is set to 1. Then the transitive closure of this connection graph should be calculated, using an order n^3 algorithm. This gives a connection table showing whether any pair of particles has a path linking them or not.

The first particle in the blob (an arbitrary choice) is defined to still be in the original blob. The algorithm iterates through all the remaining particles, checking whether there is a path to them or not. If there is path, then the particle under consideration is added to the array of those remaining and the number remaining is incremented. If there is not a path, then a new blob should be created. A while loop is used to determine the next available array for which the blob size is zero; this empty slot is used for the new blob. The particle in question is added to it, then we iterate through the remaining particles to check whether they have a path to this particle and thus connected to this blob. For any particles for which this is the case, they are added to the array, and the value of their path to the first particle is set to a third value. This value indicates to the algorithm that the particle has already been considered, and should neither be added to the remaining particle list nor added as a new blob. Finally, once all particles have been checked, the remaining particles are copied back into the blob.

Blobs only merge when they are hot enough. This means that in practice they only ever merge back together at the bottom of the bottle, and so we can "cheat" by merging on a simple height check. The merge is considerably easier than the split, due to the unordered nature of the blob array. The array of the second is appended onto the first with a series of copies, and its size is added to the first's. The second's size is set to zero, which effectively deletes it.

Particle physics

The interaction between particles is determined by the blobs they belong to. It has been observed blobs only merge together at the bottom of the lamp; this is handled by the merger described above. In all other cases, particles belonging to separate blobs must collide and rebound. On the other hand, particles belonging to the same blob should follow the electrostatic potential rules of molecules in a liquid. Due to this distinction, instead of iterating through the particles, we iterate through the blobs and then the particles within them.

Once a blob is selected, iterate through all blobs with indices higher than the current (each collision affects both particles, so we avoid checking any pair of particles twice). For each of those blobs, iterate through all particles in that blob. To check for a collision, the relative vector from the second to the first is calculated, and the length of this vector is found. This length is compared to the sum of the radii, and if it is smaller than the particles have collided. Using Newtonian physics, the total momentum (mass * velocity) and kinetic energy (1/2 * mass * velocity^2) of the particles must be preserved. By solving these two equations and noting that the change in velocity must be in the same direction as the vector from one to the other, the new velocities of the particles can be determined [13]. To calculate the new velocities after a collision: Let normal be the normalised vector from the centre of the first particle to the centre of the second, and velocity1, velocity2, mass1 and mass2 be the respective velocities and masses of the particles. Then:
P = 2 * (dot product of velocity1 with normal - dot product of velocity2 with normal)/(mass1 + mass2)
New velocity1 = velocity1 - P * m2 * n
New velocity1 = velocity2 + P * m1 * n

For internal interactions, iterate through all the particles with higher index (again to avoid double calculations) in the current blob. Molecules in a liquid follow the Leonard-Jones potential equations [11], which say that there is a repulsive force proportional to r^{-13} and an attractive force proportional to r^{-7}:

The potential at a distance r is V(r) = 4*epsilon*((sigma/r)^{12} - (sigma/r)^6). sigma is a collision diameter (the radius at which the potential is zero) and epsilon is the dissociation energy (or strength of the bond).

However, these large exponents would result in much too large forces on a macroscopic scale (compared to the molecular level they are derived at), so they were replaced with the smaller (and easier to calculate) r^{-4} repulsive and r^{-2} attractive forces. As with the collisions, the relative vector and its length are calculated. This length is divided by the desired bond length. Then the attractive force is the bond energy * ((vector length/bond length)$^{-2}$ - (vector length/bond length)$^{-4}$). This scalar is divided by the mass of the particle and multiplied by the relative position vector to give the acceleration to be applied to the particle.

Design

Both programs are split into a number of separate files, each handling a different aspect of the program: main, graphics, controls, and physics, along with some support code such as vectors and image loading. Particular details of the function workings are omitted, as they can be ascertained from the source code. This section is a guide to understanding the overall structure of the code, and is designed to be read alongside the source.

Main.cpp (both)
Contains only the **main** function. This parses command line options and sets up the GLUT function callbacks. Finally it initialises and starts OpenGL, with a 'try' to catch any exceptions.

controls.cpp (both)
Contains all keyboard and mouse control functions, which are called by GLUT after being set as callback in **main**. For keys which should perform a single action, the code is executed instantly. For keys which are held down for a continuing action, such as camera movement, the **glutKeyboard** function sets the appropriate value in the buttons array to true, which is subsequently read by the idle loop (see graphics section). When the key is lifted, the value is set to false again. Mouse dragging is handled by the **mouse** and **motion** functions. The **mouse** function sets values in the buttons array in a similar method to the held key code, and the **motion** function moves the camera when the corresponding button is held and the mouse moved.

tga.cpp (both)
For loading tga format images. This is not used for the lamp itself but for the textures in the background.

vectors.cpp (both)
Contains a range of useful vector operations: assignment, addition, subtraction, scalar multiplication, length, normalise, dot product, and cross product.

physics.cpp (mesh)
Contains the blob manipulation functions. **generate_blob** creates a blob with a given radius, position, and number of icosahedron subdivisions, and initialises all other variables in the blob. It uses the subdivide function, which given a mesh, performs a single subdivision on it. This in turn needs the **add_edge** function, which avoids duplication of edges by checking the indices of the points it is connecting in a table. **update_v** updates the velocities of the vertices of a blob, by exerting all the edge spring forces and acceleration due to temperature. update_p updates the positions of the vertices of a blob, taking into account collision detection with other blobs and with the sides, top and bottom of the lamp (which are checks against height and the radius function of the lamp). The collision detection is supported by **collision_planes**, which generates the planes used to check whether a point is inside or outside the blob.

physics.cpp (iso-surface)
Contains the particle and blob manipulation functions. **move_particles** handles collision with other blobs, bond forces with particles in the same blob, and collision with the lamp's boundaries. **split** and **merge** perform the blob division and addition algorithms described in the implementation section.

graphics.cpp (both)
Contains the rendering functions. **drawScene** calls all other drawobject functions, building up the image. The lamp is drawn last so that the glass is rendered translucently over the blobs. **drawfloor** renders the base on which the lamp stands, currently a wooden table surface, but which could be anything. **drawlamp** draws the static sections of the lava lamp, using the **lamp_radius** function to determine width at various heights. **glInit** sets up OpenGL, assigning textures, setting light-sources, and setting rendering options. It also initialises the blobs. **glResize** is called whenever the window is resized, and changes the aspect ratio and projection matrix accordingly.

graphics.cpp (mesh)
drawblob renders the blob mesh it is given. It first calculates the normals at each vertex before iterating through the faces and rendering them as triangles. **idle** is the function repeatly called by glut when it has finished rendering a frame. It calls the blob manipulation functions **update_p** and **update_v**, and deals with keys which are held down.

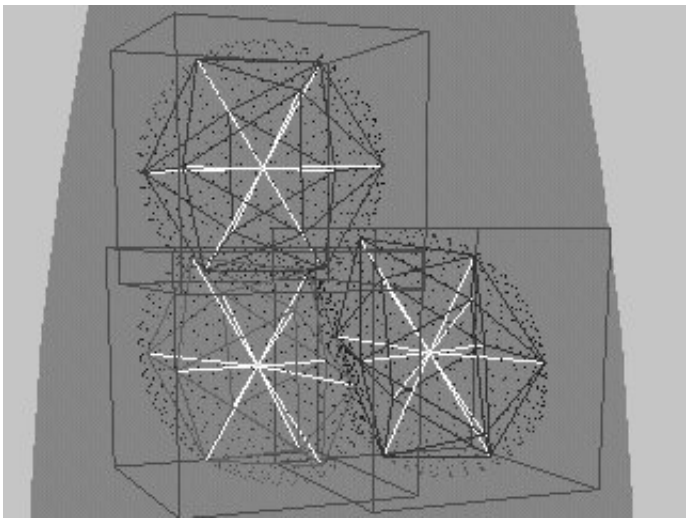
graphics.cpp (iso-surface)
drawscene performs the optimised marching cubes algorithm. The 3D function for each blob is given by **value_at** and the normal at any of these points is given by **normal_at**, which are summations across a blob of **ball_func** and **ball_norm** respectively. The initial ray cast uses **WorldtoGrid** and **GridtoWorld** conversion functions. For each cube grid cell added by **AddGridPoint**, it calls six instances of **PolygoniseTetra**, which creates the polygons within the given tetrahedron. This requires use of the **interpolate** function to estimate the position of the surface between any two points. Finally, **idle** calls all three functions from the physics section in order to animate the lamp.

Testing

For a project of this type, quantitative testing is difficult. It is possible to measure values such as the frame-rate, number of particles evaluated per second, and polygons rendered, but these are more relevant to the efficiency of the code (which was not concentrated on) than the quality of the simulation and its closeness to the real lamp. Instead, almost all the evaluation must be qualitative. The simulations were compared with a real lava lamp side by side, and elements of the behaviour (as described in the overview) which it replicated successfully were noted.

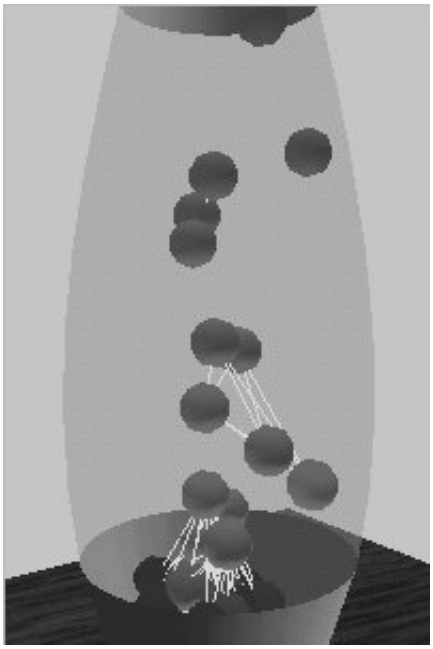
To refine the physics of both implementations, it was necessary to create diagnostic rendering modes.

The mesh based diagnostic mode gives an insight into the approximate collision detection for the blobs. The bounding box is drawn in wireframe, as is the icosahedron central to my routines. The vertices not on the icosahedron are represented as points, so that the overall shape of the blob may still be seen.



Three blobs colliding in diagnostic mode

The iso-surface based diagnostic mode removes the iso-surface rendering completely, and concentrates on the positions of the particles and the structures of the blobs. Particles are rendered as approximate spheres, and any pair of particles belonging to the same blob have a line connecting them to demonstrate the bond between them. It also shows the temperature of each particle by increasing the red component of colour with growing temperature.



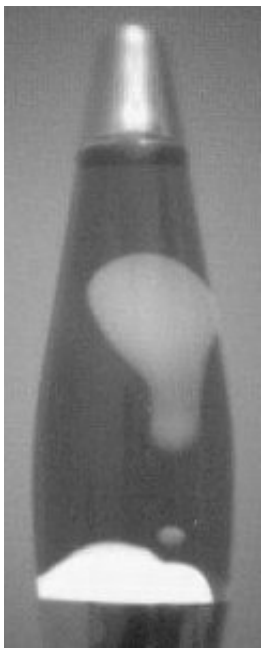
The iso-surface implementation in diagnostic mode

Evaluation

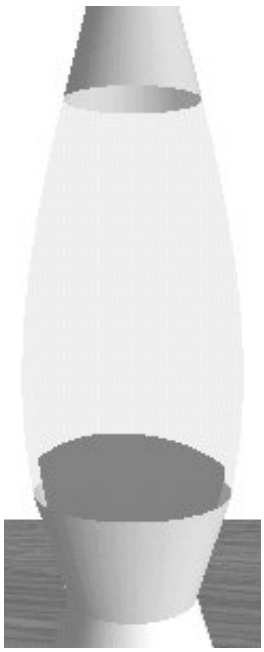
Strengths

In the overview it is stated that for the program to be successful, it should emulate the following phenomena:

- **Pooling -** The wax collects in a pool at the bottom of the bottle. This pool is not as flat as a similarly sized pool of water in the bottom of a bottle; it bulges towards the center and slowly ripples as wax enters and leaves it.
  This was successfully simulated by the iso-surface implementation, which, when provided with enough particles, creates a pool at the bottom which bulges gently in the middle.

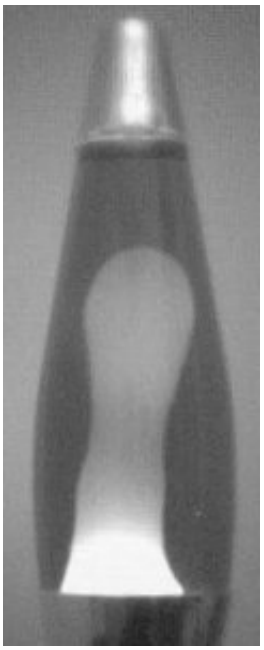


The real pool of wax

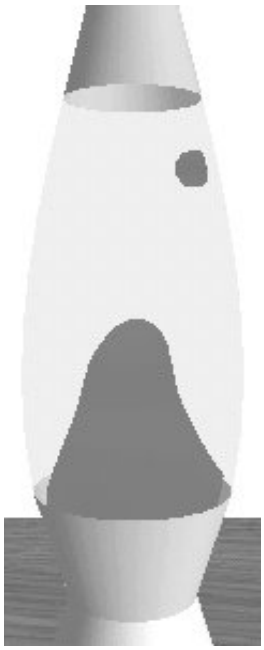


The simulated pool of wax

- **Expansion and ascent** - The wax is heated by the the lightbulb, and expands accordingly. Once the wax has a low enough density, it rises from the pool, forming an unsteady column. This is close to central, due to the position of the light-bulb; the wax towards the perimeter is not heated as much and does not rise. The column writhes back and forth as the hot wax flows up and down inside it. The diameter of the column is different at different heights and is constantly changing. *This was partially achieved by the iso-surface renderer. As particles rise up from the large blob at the bottom, a short, thick column forms. Unfortunately, the bonds break too early, and it rarely forms the long thin column observed in the real lamp. This may be fixed by changing the bond constants, but this will have to be determined experimentally. Both implementations have the heated blobs rising.*

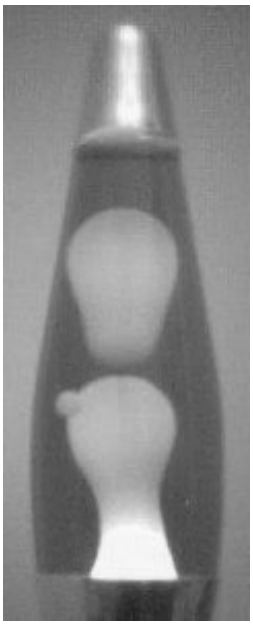


The real column of wax



The simulated column of wax

- **Splitting** - Eventually the column breaks off into one or more distinct blobs. The top of the column forms one large blob, and sometimes the rest of the column can break off into smaller ones. These blobs are approximately spherical. In a similar fashion, the larger blobs formed by this process can stretch out vertically and breaks into more blobs. First they stretch into a vertical hourglass shape, with the 'neck' in the middle gradually thinning to a cone, with the large end at the top and the small at the bottom. When the piece at the bottom breaks off, the point of the cone may also generate small blobs.
  This was also partially achieved by the iso-surface version. Blobs split apart smoothly, sometimes into several pieces, with a small 'neck' joining them as they part. However, unlike the real blobs, they do not always split vertically.



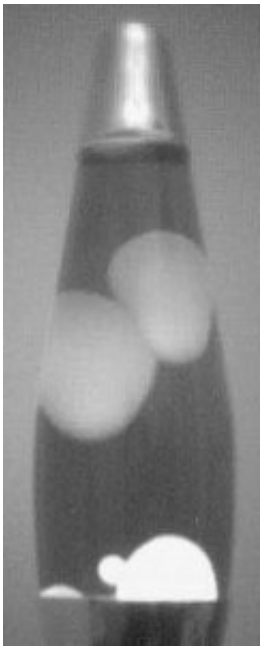
The real column breaking

The simulated column breaking

The real blob splitting

The simulated blob splitting

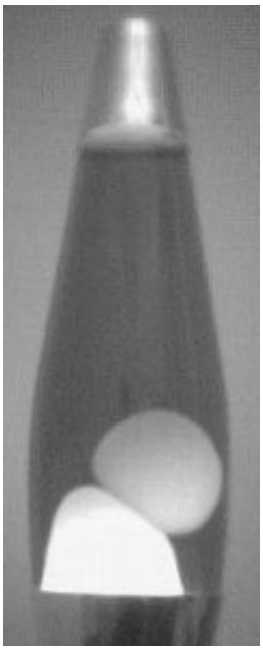
- **Collision** - Once splitting has occurred, surface tension ensures that the wax does not re-combine until it reaches the pool at the bottom again. Instead, blobs deform when they collide with each other, with the impact propogating as a low frequency wave across the surface. It slowly settles back into a flattened near-spherical shape afterwards. The collisions are close to frictionless, and the kinetic enery of the blobs is converted to deformation.
  This is where the mesh implementation excels. Deformation due to collisions and subsequent rippling is highly realistic, and the energy of the collision is visibly converted absorbed into the blobs.



The real blobs colliding

The simulated blobs colliding

- **Cooling and descent** - As the wax rises, it cools, and when it becomes dense enough it falls back to the pool. It has usually reached nearly if not all the way to the top of the bottle by the time its velocity changes direction.
  Both implementations have blobs with temperatures affecting their vertical acceleration.



The real blob returning to the pool at the bottom

The simulated blob returning to the pool at the bottom

Weaknesses

The biggest weakness in the project is that the realistic behaviour of the blobs is split across two implementations of the lamp. Merging these two implementations is non-trivial, although it should possible.

Due to the approximate nature of the collision detection for the mesh based implementation, problems can occur. The worst is when the blob turns convex, which usually happens when it is crushed beneath another at the bottom of the lamp. This contradicts the assumed structure of the icosahedron, and essentially turns the collision detection inside out. The result of the error is that the blob which has crushed the convex one ends up being inexorably sucked into the other.

The marching cubes renderer is glitchy, and could be optimised further. The glitches stem from the normals generated by the tetrahedron polygoniser, which doesn't know which way the polygon should be facing. To resolve this, a short ray is cast in the direction of the normal it has chosen, and if the ray ends up inside the threshold, then the normal is reversed. This doesn't work 100% of the time, and sometimes it ends up with polygons facing in the wrong direction, giving serious lighting artifacts.

There are noticeable jumps in animation when a blob is split from or merged back into the pool at the bottom. Due to the large mass of the pool and resulting powerful influence over the overall function, any addition or subtraction results in a large change in the surface rendered. This could be solved using a gradual merge and split algorithm, although it would require a much more complex representation of each blob. Furthermore, the meshes can intersect before they merge, causing clipping problems and discontinuities.

A lot of the physical constants plucked out of the air for the simulation. It would be very difficult to measure the true values of these, especially given the macroscopic nature of the particles, and instead they have to be determined by experimentation (see future work).

Future work

More accurate physics

While I am confident that the simulation can produce realistic results, it relies on many user defined constants: the distance at which blobs split, the bond length, the bond energy, the iso-surface threshold, the particle masses, the rate of heating and cooling, and so on. Experimental evaluation should determine the optimal values, but this may take a long time due to the number of variables. Furthermore, liquid in which the wax floats was not considered. This will also have convection currents, and the hydrodynamics will play a part in the motion of the wax. However, this is much harder to observe than the behaviour noted already.

A more complex approach to hydrodynamics than the approximate Leonard Jones potential which was used is the Smoothed Particle Hydrodynamics approach. Due to the difficulty of understanding and implementing the system, it was not included in the project, but it may provide the key to creating more realistic animation. It also uses particles following the fluid flow to encode the information, and include factors such as artificial viscosity [12].

Active contours

The marching cube algorithm is only one way of rendering an isosurface. Another method in development is active contours. These have already be explored for 2D computer vision applications (sometimes called ’snakes’). In the 2D case, a function is optimised so that it fits around a certain section of an image. It can be restricted for elasticity and smoothness. The technique is particularly good for following objects which change by small increments from frame to frame. This has now been extended to 3D, using adaptive techniques which can create extra polygons for difficult sections and merge large areas for efficiency. It would be particularly suitable for this application because the blobs are slow moving objects which could be tracked well by the contours [7].

Ray-tracer

A much slower method of rendering isosurfaces is ray-tracing [10]. For every pixel on the screen, a ray, simulating a ray of light, is cast from the viewer into the 3D space being observed. When it hits something, the colour there is calculated. This can take into account secondary rays, which are rays cast from the point hit in the direction of the original ray reflected by the normal at that point. For highly specular surfaces, such as the wax blobs in consideration here, this can give much more realistic results. Seconrdary rays can also come from refraction; both the glass bottle and the liquid inside it have a high refractive index compared to air, and this will distort the image behind and inside the bottle. Calculating all these rays is very expensive, and rather than running a real-time simulation it would pre-render a series of frames of animation to be played back as a video.

Merge mesh-based with iso-surface implementation

Both the mesh based and the iso-surface implementations of the lamp have their advantages and disadvantages. The mesh based creates beautifully animated single blobs, but cannot perform splitting and merging. The iso-surface simulates splitting and merging properly, but individual blobs do not look as realistic as the mesh based ones. Merging the two implementations successfully should achieve the best of both worlds. This could be probably be achieved by using the meshes as active contours, which would be wrapped around the particle blobs. Certainly single particle blobs could be rendered as mesh blobs; the case for multiple particles would be harder but not impossible.

Radiosity, both within lamp and the lighting the lamp gives to its surroundings

A lava lamp is, as well as a decoration, a light source. The movement of the blobs subtly affects the light the lamp gives off. When a large amount of wax is covering the bulb, the light is darker, but when most of it is higher, it is much brighter. This could be simulated crudely by just making the light given off proportional to the average height of the particles, but to simulate the lighting properly, each blob would have to be a separate secondary light-source. The partial translucency of liquid wax further complicates the lighting.

Conclusions

Hoping for both real-time speed and photo-realistic quality was over-ambitious; fluid dynamics are highly complex and require a lot of power to compute. However, I believe I have created some very visually pleasing results. The computer games industry is constantly pushing forward the boundaries of real-time simulation, and this project shows that even such complex systems can be approximated quickly, given a combination of observation of macroscopic behaviour and an understanding of the true molecular physics in the system, although it does take a lot of experimentation with the physical constants. While the immediate applications of the technology are not obvious, it is a demonstration of how the combination of a number of existing techniques can give impressive new results.

Bibliography

OpenGL
[1] Neider, Jackie, Tom Davis, and Mason Woo. OpenGL Programming Guide (HTML text): Addison-Wesley Publishing Company, 1994

Existing Programs
[2] Walken, Craven. "Lava Lite Lamp" Lava i-Mate Jan. 2002. i-Mates.de
[3] Heidemann, John. "LavaPS"
[4] Miaw, Wesley, Tobin Fricke, and Kalman Lau. "Lava Lamp project"
[5] Rudd, Robert. "Lava Lamp" The Mathematica Journal Vol. 5 Issue 4 (Fall 1995), p. 12-16
[6] Gerken, Till, Olaf Roß. "The Lava Lamp"

Graphics
[7] Bowden, R, T. A Mitchell, and M. Sahardi. "Real-time dynamic deformable meshes for volumetric segmentation and visualisation"
[8] Desbrun, Mathieu. "Active Implicit Surface for Animation"
[9] Bourke, Paul. "Polygonising a scalar field"
[10] Geiss, Ryan. "Metaballs"

Physics
[11] Allen and Tildesley, "Computer Simulation of Liquids": Oxford Press
[12] Hultman, John. "A course in Smoothed Particle Hydrodynamics"
[13] van den Heuvel, Joe, Miles Jackson. "Pool Hall Lessons" Gamasutra.com Jan. 2002

User guide

Source code and both Windows and Linux executables are available at
http://www.doc.ic.ac.uk/~kor98/project/code/

Command line options for iso-surface implementation

-B(number)	Floating point strength of bonds between molecules
-D(number)	Floating point distance at which particles in a blob will split
-G(number)	Floating point size of the grid cells on which to render the iso-surface
-P(number)	Integer number of particles which will be rendered
-T(number)	Floating point threshold for the iso-surface

Controls for both implementations

Hold the left mouse button and drag to rotate around the lamp. The equivalent keyboard controls are the arrow keys.
Hold the right mouse button and drag up and down to zoon in and out.
Press ’P’ to pause the simulation.
Press ’Q’ or the escape key to quit.
Press ’R’ to toggle the diagnostic rendering style.

Controls for iso-surface implementation

Use ’A’ and ’Z’ to increase and decrease the threshold respectively.
Use ’S’ and ’X’ to increase and decrease the grid cell size respectively.
Use ’G’ to toggle the grid display on and off.

Both implementations are in constant development, and the controls may change; the online user guide will be updated to keep track of these changes.

A 3D Lava Lamp Simulation

Table of Contents