

## Overview

The primary motivation for this project was to implement a physics-based simulation showing off real-time, game-suitable techniques for soft-body dynamics and animation. To that effect, I focused on the specific soft body dynamics implementation, as well as on creating an environment that could be found in a common game and testing the soft body dynamics in such an environment (as opposed to a “safe” box-world where everything is known or hard-coded). I thus spent time working on side-effects of the soft body physics, including the rendering techniques used both in surface detail on the soft-bodies, as well as on overall shadow techniques.

In addition to the notes here, I invite the reader to visit my webpage at [www.emeyex.com](http://www.emeyex.com) where there are screenshots and additional information pertaining to the project.

## Things That Went Right

### Spring-Based Deformable Meshes:

Though not ultimately as suitable to arbitrary triangle render meshes as I had hoped, the spring mesh implementation that I ended up with was fairly resilient for simple shapes and provided a fun and pleasing soft effect. I have included a full tutorial-style write-up outlining this method and included it as an appendix.

### Cloth:

I was also happy with the outcome of the cloth implementation. There were some issues with the cloth collision, but I consider it as part of the larger problems I faced with collision. I discuss these problems below. The overall cloth technique is also outlined in the tutorial-style write up on spring-based techniques in the appendix.

### Loose Quadtree:

I ended up implementing what is known as a “loose” quadtree for space-partitioning and quick object culling in collision detection. The loose aspect of the quadtree has both advantages and disadvantages. On the plus side, it allows for constant-time object insertion and removal, as an object’s level and cell in the quadtree can be directly computed based on its bounding sphere radius and world position; also, objects will not have a tendency to float up into larger cells just because they straddle the boundary between two cells, as is common with regular quad/octrees. On the bad side, when queries are made on the quadtree for potentially colliding objects, more cells will be tested, potentially resulting in more objects received in the query (however, this effect is mitigated by the fact that objects can actually “fit” at the level they should). Overall, I found that the technique allowed my application to scale much better, in the sense that I could add dramatically more objects to the scene while maintaining reasonable frame rates.

## Stencil-Buffer Shadow Volumes on the GPU:

Though I started the project with an implementation of depth-buffer-based shadow mapping, I was unhappy with the visible aliasing which is common to that technique. I thus spent some time researching alternatives, and I was very happy when I came upon hints towards GPU-based implementations of shadow volumes. I was familiar with the overall technique, but not with the ingenious trick that allows all shadow-volume computation and extrusion to take place inside the vertex shader itself on the graphics card (as opposed to performing silhouette detection on the CPU). This effect proved very efficient and was fairly straightforward to implement for static meshes as well as for skinned, animated meshes.

To make the technique work for the soft body meshes, I had to add extra information to the soft body vertex format that allowed each vertex to dynamically compute its triangle's polygon normal on the shader at run-time. This technique proved very successful, making the soft body shadows as convincing as the other shadows found in the scene.

## Things That Went Wrong

### Collision:

Collision handling is obviously a large undertaking, and given that this was my first wrestling match with it in 3D, I was satisfied with it as an initial project. However, I now have gained a better perspective, and know what I would do differently the next time around. The most obvious problem with the collision that can be seen in the demo itself is the fact that interpenetration constraints are not maintained. There are two primary reasons for this, as far as I can tell: *i)* I did not give enough consideration to what is commonly referred to as “resting contact” collision, which is a surprisingly separate matter from “colliding” collision resolution; *ii)* My collision detection routines were not thorough enough, and specifically did not perform collision detection on edges

Before further discussing what went wrong with my collision, I will give a basic overview of the physics update loop. Essentially, it involved iterating over each moving object in a single loop. For each moving object:

- I remove that object from the quadtree, then integrate the forces currently acting on it (force  $\rightarrow$  acceleration  $\rightarrow$  velocity  $\rightarrow$  position, using Euler integration and velocity clamping)
- I then query the quadtree for any objects potentially colliding with the object at its new position
- I resolve all actual collisions by modifying the position and velocity of the moving object (not the objects against which it is being tested)
- I consider the object static and move on to the next object

Looking at it now, this algorithm seems to make some basic sense, but there are many flaws with it, most notably perhaps the lack of any post-processing for resting contact.

Not providing for resting contact collision from the outset seemed reasonable at the time; it's somewhat more involved than handling colliding contact, and I figured I didn't need stacking in this demo. But in fact it led to objects that were barely moving slowly overlapping. I believe this could be mitigated by changing the overall physics loop and adding in a phase for post-processing, after colliding contacts have been handled.

In addition, the manner in which I iterate over each object, treating only it as moving, seems a bit too much of a departure from a physically-based solution; by treating only the one object as moving and adjusting only its velocity and/or position, the collision cannot be fully inelastic, and hence resting contact cannot occur. In addition, I often resorted to *ad hoc* procedures within the collision resolution function, tweaking constants and at times modifying objects that should have been considered "inactive", in order to achieve visually pleasing results. I feel this could have been minimized with a more carefully planned update loop.

Though I did not have time to implement an improved version, I do have an idea of what I will try next, based on researching different methods toward the end of the semester. The new physics and collision resolution phase would look like:

- Temporarily move each object to new predicted position (integrate)
- Determine all colliding pairs of objects
- Resolve such collisions on objects with impulses based on their old velocity (velocity at beginning of frame); potentially loop multiple times
- Now actually move each object using new forces and velocities (integrate)
- Resolve remaining collisions as resting contact collisions, using completely inelastic impulses

Future work will hopefully test my theories about how successful this approach will be.

# Appendix A: Tutorial on Spring-Based Physics Techniques

## The Basics

### Hooke's Law

Springs can be useful in implementing a variety of phenomena in a game-like simulation environment, including deformable bodies, cloth, 3<sup>rd</sup>-person cameras, collision resolution, and even certain aspects of AI.

Specifically, let's define a spring to be an "object" connected to two masses which applies an equal and opposite force to each mass based primarily on the distance separating the two masses. You are probably familiar with the famous equation known as Hooke's law,  $\mathcal{F} = -k\chi$ , where the force applied on an object is proportional to the distance of that object from some initial, "at rest" position, and where  $k$  is the (positive) constant of proportionality that governs how strong the force is ( $k$  will be large for a stiff spring, and small for a loose spring). Hooke's law gives rise to the phenomenon known as Simple Harmonic Motion, where the motion of an object on a spring is sinusoidal. This can be seen by manipulating Hooke's law:

$$\begin{aligned}\mathcal{F} &= -k\chi \\ a &= -mk\chi\end{aligned}$$

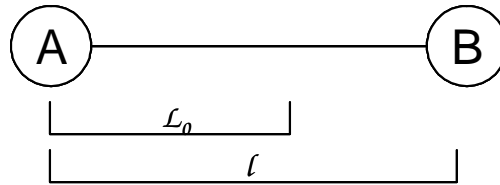
This simple manipulation explicitly demonstrates the relation between an object's acceleration and its position; notably, that, aside from a constant scaling, they are equal. Knowing that acceleration is the second derivative of position suggests that the sine or cosine functions could represent such a relation. If we were pursuing a symbolic analysis of the spring equations, we might go further with the sinusoidal representation; however, for the purposes of numerical methods of integration as implemented in a computer simulation, we prefer to have our quantities in terms of forces.

Hence, we will keep on with Hooke's law as the underlying representation of our spring forces. However, we want to have a few more parameters to allow for more fine-tuned control of the spring. So let's get a little more precise.

### Natural Spring Length

Suppose we want to hook a camera to a player, as in the case of a 3<sup>rd</sup>-person camera. We clearly want the camera to come to rest some distance away from the player, as opposed to constantly trying to move to the player's exact position. That is, there is some natural

distance away from the player, at which the camera should remain at rest. For the camera to remain at rest, we need the spring connecting the camera to the player to not apply any force.



Hence, letting

$\mathcal{L}_0$  stand for the natural length of the spring;

$l$  stand for the current length of the spring (distance between A and B);

$\boldsymbol{\chi}$  be the unit vector in the direction from A to B;

we can redefine the spring forces acting on object A and object B as:

$$-\mathcal{F}_A = \mathcal{F}_B = -k(l - \mathcal{L}_0) \boldsymbol{\chi}$$

Notice that if  $l$  is equal to  $\mathcal{L}_0$ , then the force will be zero as desired. Likewise, the spring will push object B away from object A if  $l$  becomes smaller than  $\mathcal{L}_0$ . If for some reason this behavior were not desired, the quantity  $(l - \mathcal{L}_0)$  could always be clamped to zero.

### Damping Proportional to Velocity

In the absence of any frictional or damping forces, the spring forces as modeled above would cause endless oscillatory motion. While such forces can be modeled elsewhere in the simulation, it useful to add a damping term directly into the spring force equation. This allows more control over the behavior of a spring, and specifically keeps a spring from looking so “springy”. This allows a spring to act more like a strut or a shock, as in a car or architectural support system.

To dampen an object’s motion means that we want to apply a force to the object against it’s velocity. Linear damping generally takes the form of:

$$\mathcal{F} = -b\boldsymbol{v}$$

where  $b$  is the (positive) damping constant and  $\boldsymbol{v}$  is the object’s velocity. More complicated damping is also possible, including damping proportional to an object’s squared velocity, as well logarithmic functions. I will focus here on simple linear damping.

To incorporate damping into the spring equation, we first find the relative velocity between the two connected object A and B. Keeping with the above description and diagram, we define relative velocity as

$$\mathbf{v}_{rel} = \mathbf{v}_B - \mathbf{v}_A$$

This represents the velocity of object B with respect to object A. Further, we only want to dampen motion along the axis of the spring. That is, we only care about the component of  $\mathbf{v}_{rel}$  which lies in the direction of  $\boldsymbol{\chi}$ . To find this quantity, we must project the vector  $\mathbf{v}_{rel}$  onto  $\boldsymbol{\chi}$ ; because we have defined  $\boldsymbol{\chi}$  as a unit vector, the projection amounts to a dot product. Our final damping term acting on object B then is

$$\mathcal{F}_B = -b(\mathbf{v}_{rel} \cdot \boldsymbol{\chi}) \boldsymbol{\chi}$$

Combining the damping with the spring force gives us

$$\begin{aligned} -\mathcal{F}_A = \mathcal{F}_B &= -k(l - L_0) \boldsymbol{\chi} - b(\mathbf{v}_{rel} \cdot \boldsymbol{\chi}) \boldsymbol{\chi} \\ &= -(k(l - L_0) + b(\mathbf{v}_{rel} \cdot \boldsymbol{\chi})) \boldsymbol{\chi} \end{aligned}$$

Care must be used when selecting a value for  $b$ , as too large a value may provoke more drastic oscillations against the normal spring motion. Generally speaking, a value of  $b$  greater than one will push the object back by more than the length of the relative velocity, thus we usually select a value between zero and one.

### Empirical Damping for Stability

It is also possible to apply a more severe form of damping, more along the lines of clamping. It is possible that the stability of a large system of springs will degrade, given too large a time step, too poor an integration method, or too stiff of spring constants ( $k$ ). Hence, it can be useful to incorporate a quick empiric model to help maintain the stability of the system.

One such solution involves clamping the velocities of the objects in the direction of the spring force should the length  $l$  become too large (or too small). To this end, we can incorporate another parameter into our spring model, called the spring length tolerance,  $t$ . This value should represent a maximum allowed percentage change with respect to the natural length  $L_0$ . We can now perform a check on the current length  $l$ , using the following boolean

$$l < (1 - t)L_0 \mid \mid l > (1 + t)L_0$$

If the result of the above expression is true, then we clamp the velocities of the objects in the direction of the spring force. Again, this involves projection of each object's velocity onto the unit vector  $\boldsymbol{\chi}$ . Each object's new velocity can be computed as

$$\begin{aligned} \mathbf{v}_A' &= \mathbf{v}_A - (\mathbf{v}_A \cdot \boldsymbol{\chi}) \boldsymbol{\chi} \\ \mathbf{v}_B' &= \mathbf{v}_B - (\mathbf{v}_B \cdot \boldsymbol{\chi}) \boldsymbol{\chi} \end{aligned}$$

This clamping can prove useful in larger systems of springs where we need some extra help to maintain stability in a real-time simulation.

## Systems of Springs

The above-defined equations for spring forces can be used for a simple pair of moving objects, as well as for a more complicated arrangement, such as a regular grid of springs or even an arbitrary spring mesh. The equations are very flexible in that regard, and as such extremely useful.

There are other concerns which creep in however when dealing with larger systems of springs. I will attempt to address some of those issues here, at least those that I have encountered myself.

### Efficiency

In general, when we integrate the equations of motion for objects in a physics simulation, we compute all the forces acting on an object, and then integrate upward from acceleration to position. If we were to follow this scheme for a system of springs, treating each mass as an individual object accumulating forces, then we would redundantly perform each spring force computation twice. This is because, as shown above, the spring force acting on two objects is equal but opposite. Therefore, if we iterate over each spring, instead of each object in the system, we can cut down by half the number of times we compute the spring force.

Thus, we should add the forces for each spring to each object in the system before performing integration. This process will also help improve stability in the system; if forces for one object in the system are computed and then integrated before moving onto the next object, the system will not compute all forces on each object as they were at the beginning of the time step. This is an important factor to be aware of.

### Arbitrary Systems

While a grid of objects connected by springs can be represented in a straight-forward manner using a 2D array, under the assumption that adjacent cells are connected by springs, it is useful to develop data structures that will handle more complex systems. Two complementary data structures can be used to manage such a system. One should contain the list of masses contained in the system (and their associated attributes, including force, velocity, position, etc.), and the other a list of index pairs representing the springs, where the index pairs represent the masses that are connected. Therefore, we can easily iterate either over individual masses or over the springs, as necessary. I call the data structure that contains these lists the *spring mesh*.

## Applications: 3<sup>rd</sup>-Person Camera

As mentioned above, one very simple application of springs comes in implementing a 3<sup>rd</sup>-Person camera.

First, we represent the camera as we would any other physics object, giving it a position, velocity, and force. In addition, we will want to include collision information, so that we can prevent the camera object from going through objects we don't want it to (i.e., the terrain, or walls).

We then associate a spring between the player and the camera. However, we don't want the spring to act on the player object (it would be a pretty annoying camera if it hindered the player's movement). Therefore, it is important that the spring object is capable of recognizing whether it should apply forces to both objects or just one.

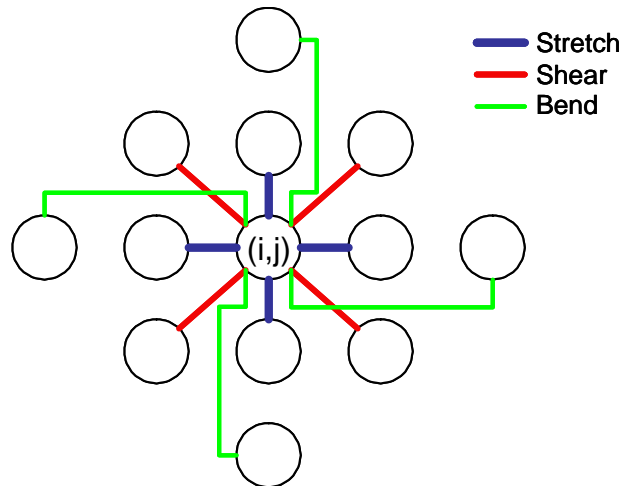
With such physics objects added to the list of objects to update, we simply step through the regular physics loop, accumulating forces and integrating them, and the camera will move smoothly and intuitively, following the player within a reasonable distance. It is important to play with the various parameters we defined above, including the natural spring length, the spring constant, and the spring damping. In addition, you will want to play with whether you add gravity to your camera, and also whether it has friction with the ground. I've found these latter two forces to be useful in making the camera more stable.

After each physics update loop, you will then need to change the view matrix by computing a standard "look-at" matrix using the camera-to-player vector as your look vector.

Pretty simple! No crazy AI or logic required, just spring-physics...

## Applications: Cloth

A standard approach for implementing cloth is to use a rectangular grid of point masses. However, instead of just assuming springs connecting adjacent elements in the 2D array, we add extra springs to connect diagonal elements, as well as elements one-element away from each other. The diagram below demonstrates these connections on the  $(i,j)^{th}$  spring in a 2D array. Note that the diagram does not demonstrate the corresponding set of springs on the other elements, for simplicity. Each point mass in the system would be connected as shown in the diagram.



We associate a different semantic to each of these different types of springs. In most of the literature that I've read pertaining to such techniques, these different spring forces are referred to as *stretch*, *shear*, and *bend*, for intuitive and obvious reasons (stretch springs act against stretching, shear springs act against shearing, and bend springs act against bending). Usually we will use a different set of parameters for each spring type, but keep all springs of a given type the same; i.e., the spring constant, damping, and natural length will be the same for all stretch-springs in the system, but the shear-springs will have a different set of these parameters, and likewise for the bend-springs.

## Applications: Deformable Bodies

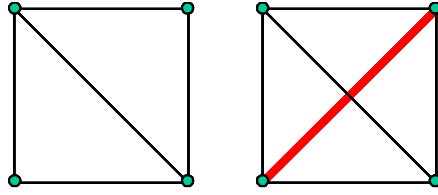
One of the most exciting applications of using springs is in simulating real-time deformable meshes. As mentioned in the section describing arbitrary systems of springs, a spring mesh should be used to aid in the implementation of a volume-based deformable body.

There is no one way to implement deformable volumes, but I will propose a fairly straightforward approach that is fairly simple to implement and successful for basic shapes such as cubes and spheres.

We start by taking as input a triangle mesh, as would be exported by some modeling package. All we are interested in is the actual vertex position data (i.e., we don't need vertex normals, UV data, etc.). From this data we will generate two sets of springs, which we will call the *surface springs* and the *volume springs*.

### Surface Springs

As the name would imply, the surface springs represent the skin of the mesh. We iterate over each triangle of the input triangle mesh, inserting a spring for each edge of each triangle. However, by observing a simple quad, we can see that such a scheme will not lead to a balanced set of springs (left); a balanced system would insert a spring along the other diagonal (right).



example of inserting springs along triangle edges (left)  
 example of inserting an extra spring along unconnected diagonal (right)

We want to be able to detect such unconnected diagonals for an arbitrary mesh, as this will prevent our mesh from caving in and shearing in the direction of the arbitrarily selected triangulation.

We assume that the input triangle mesh contains a vertex list  $V$  and an index list  $I$ , where the vertices of the  $i^{\text{th}}$  triangle are defined as  $V[I[i*3]]$ ,  $V[I[i*3+1]]$ , and  $V[I[i*3+2]]$ . Then two triangles share an edge (as in the left diagram above) if two of the indices for the two triangles are equal. Some pseudocode for detecting whether the  $i^{\text{th}}$  and  $j^{\text{th}}$  triangles are in such a situation is:

```

int shared=0;
if ( I[i*3] == I[j*3]           ||
    I[i*3] == I[j*3+1]        ||
    I[i*3] == I[j*3+2] )
    ++shared;

if ( I[i*3+1] == I[j*3]        ||
    I[i*3+1] == I[j*3+1]      ||
    I[i*3+1] == I[j*3+2] )
    ++shared;

if ( I[i*3+2] == I[j*3]        ||
    I[i*3+2] == I[j*3+1]      ||
    I[i*3+2] == I[j*3+2] )
    ++shared;

if ( shared == 3 ) return error;//two identical triangles
if ( shared == 2 ) return shared_edge;
return no_shared_edge;

```

Computing which indices of each triangle are unshared can be combined with the above pseudocode, and then we can insert a spring into our surface spring mesh, representing a connection between the two previously disconnected vertices.

After we have computed all the surface springs, we will want to compute the natural spring length for each spring. We do so by iterating over each spring and computing the initial distance between the two vertices connected by each spring; this distance represents the length of the springs at rest, which we assume to be the distance between the vertices when the triangle mesh was authored. Be sure that, if the model needs to be scaled, this scaling has been performed to the vertices prior to setting up the springs.

Normally, we will just use the same spring force and damping constants for all the surface springs, hence we would store the variables exterior to the springs (as we only need one copy for all of them). However, one could certainly store separate variables on each spring, if there were some reason to have different values for each spring.

### **Volume Springs**

Again, as the name implies, volume springs represent the internal volume of a body. In many ways, this is the hardest part to represent; however, if we ignore trying to model the volume, our deformable meshes will simply wither, wilt, and quite simply look like crap.

One of the simplest and most “cost-effective” methods for modeling the volume of a deformable body is by connecting each vertex on the body’s surface to a “center” vertex by a spring. This requires adding new mass to our spring mesh (call it the center vertex). We then add new springs for each vertex on the surface of the mesh to this new center vertex, computing the natural spring lengths as above for the surface springs. We will usually want to store new values of the spring force constant and damping for the volume springs (separate from the copy we have for the surface springs), as this will allow for better tuning (i.e., some meshes may work better if we turn up the force for the volume springs, but keep the surface springs turned down, or vice versa, etc.).

# Appendix B:

## Tutorial on Vertex Normal Calculations

### Vertex Normals vs. Surface Normals

We all know what a surface normal is (if not, it's the normal to the plane that contains the surface). So how can a vertex (i.e., a point), have a normal? Strictly speaking, it can't. What vertex normals provide is a means of simulating smoother surfaces during lighting calculations when using procedures such as Phong or Gouraud shading (sometimes just called Smooth shading). Imagine a polygonal mesh of a human: technically, this mesh is just a bunch of flat polygons. But really, this mesh is simulating the smooth surface of a human body. If all pixels within a polygon were lit identically, the "flatness" of each polygon would be starkly obvious; but by using vertex normals, we can light each vertex on a triangle differently, thus causing a smoother appearance. The trick is to generate these vertex normals so that they actually enhance this smooth appearance.

### The Intuition behind Vertex Normals

Let's take a close look at the lighting process involved in traditional polygonal mesh, scan-line rasterization techniques (such as those employed on graphics cards and in software renderers for real-time applications). Lighting will be applied per-vertex, where the dot product of the surface normal and the light's direction will be used to modulate the intensity of the light's color (this aspect of the light model is called the Diffuse lighting term). This modulated light color will then be added to a pre-computed, constant vertex color. Other terms are also possible (specular, ambient) as well. Thus, if we are using triangles, all three vertices of a triangle will have an identical dot product between the light direction and surface normal vectors<sup>1</sup>. Using the same normal for all three vertices assumes that the actual surface we are simulating is flat, but we know this is not the case. It's as though we are using a single sampling point to compute the normal for all three vertices. But imagine we could actually sample the "true" surface at the vertices themselves; then we would surely get more variation amongst the vertex normals, in turn creating a (smooth) variation in the diffuse lighting terms. But how can we sample the "true" surface?

Unfortunately, we can't. However, we can come close, using the observation that a vertex lies at the intersection of multiple triangles. Hence, instead of using the surface normal of the one triangle of which the vertex is a part, we can detect the adjacent triangles (i.e., the other triangles that share the same vertex), and perform a kind of

---

<sup>1</sup> Alternatively, if point light sources are being used, there will be a small amount of variation in the dot product, as the light's direction will be computed per vertex as the difference between the vertex's position and the light's position. Nonetheless, given small triangles, the diffuse term will remain nearly identical for all three vertices.

“averaging” of all adjacent surface normals. This provides the intuition behind the notion of a vertex normal, as there is no strict definition (there are multiple techniques for computing vertex normals, and no one is “correct”).

## The Algorithm

Now that we have an idea of what the vertex normal, let’s devise an algorithm to compute them. After devising a standard algorithm, I will discuss a few variations that can be incorporated for a nicer appearance.

The idea is that we want to average the normals of all polygons that are adjacent to a given vertex. Recall that in standard rasterization techniques, we have access to a list of triangles; this list will necessarily contain duplicate vertices, as a given vertex is included for each triangle of which it is a part. This duplication is often alleviated using indexed triangle lists, but to keep the argument simple, let us assume we are using an un-indexed list, where the  $i^{\text{th}}$  triangle in our vertex list is defined using the three vertices’ positions  $v[i*3].p$ ,  $v[i*3+1].p$ , and  $v[i*3+2].p$ . Then our task is to find, for each vertex, all the triangles that share that vertex. Each time we find a triangle that shares the current vertex, we compute the triangle’s surface normal, and add it into a running tally. When we are done, we normalize the vertex normal, and store it with the vertex.

In pseudocode:

```
struct Vertex { Position p, Normal n }
VertexList v

for each vertex i in VertexList v
    n ← Zero Vector
    for each triangle j that shares ith vertex
        n ← n + Normalize(Normal(v, j))
    end for
    v[i].n ← Normalize(n)
end for
```

The routine `Normal(v, j)` is assumed to return the outwardly facing normal to the triangle using the index scheme described above.

This pseudocode is pretty simple, though not particularly fast for a large number of vertices ( $O(n^2)$ ).

One problem with the existing approach is that, for surfaces that were actually *not* meant to be smooth (i.e., a mesh of a cube), the result will appear strange; at the corners of a sharp crease, the lighting will appear smooth, which is clearly not what we want.

One approach to combat this is to use a threshold when deciding whether to include a triangle’s normal in the averaging. Specifically, when examining the  $i^{\text{th}}$  vertex, we cache

the surface normal of the triangle to which the vertex technically belongs (as opposed to the triangles which share a copy of the vertex elsewhere in the triangle list). Then, when upon finding another triangle that shares the vertex, we compute the dot product between the cached normal and the current triangle's normal; if this dot product is less than some threshold (call it  $\epsilon$ , usually a little above 0), then we do not include the current triangle's normal in the average. How does this work? Recall that the dot product corresponds to the cosine of the angle between two vectors. What we are effectively saying is, if the angle between two surfaces is greater than  $\epsilon$ , don't average them.

The new pseudocode:

```

struct Vertex { Position p, Normal n }
VertexList v
epsilon e

for each vertex i in VertexList v
    n ← Zero Vector
    m ← Normalize(Normal(v, i%3))
    for each triangle j that shares ith vertex
        q ← Normalize(Normal(v, j))
        if DotProduct(q, m) > e
            n ← n + q
        end if
    end for
    v[i].n ← Normalize(n)
end for

```

Another possible improvement to our algorithm comes from examining the fact that, while a vertex lies at the intersection of multiple triangles, these triangles are not necessarily of equal size. Some people suggest using the area of the triangle as a weighting factor when including a surface normal in the averaging process. The pseudocode is hardly changed to incorporate this; we just require a new routine to compute the area of a triangle, given the three vertices.

The final version looks like:

```

struct Vertex { Position p, Normal n }
VertexList v
epsilon e

for each vertex i in VertexList v
    n ← Zero Vector
    m ← Normalize(Normal(v, i%3))
    for each triangle j that shares ith vertex
        q ← Normalize(Normal(v, j))
        w ← Area(v, j)
        if DotProduct(q, m) > e
            n ← n + w*q
        end if
    end for
    v[i].n ← Normalize(n)
end for

```

## Applications to Real-Time Generation

So what about a deformable mesh? For a rigid body of course, where the vertices *do not* move with respect to each other, we can place a vertex normal in the world using the inverse transpose of the world transformation matrix. But this won't work for vertices whose positions change with respect to the other vertices on the mesh. This is because the planes governing the generation of the normals change with respect to each other.

We clearly can't just use the above algorithm *as is*, because it's too slow for a mesh with many polygons. We can however do a pre-process which will allow us to more quickly update the normals. Consider that generating a vertex normal for a given vertex requires knowledge of the adjacent polygons; finding these adjacent polygons in an arbitrary triangle list requires looping through the entire list; this is slooow ( $O(n^2)$ ). But if we do this part once, as a pre-process, and associate with each vertex a list of adjacent polygons, then computing all the vertex normals can be performed in linear time. You simply walk your vertex list, then iterate through the (pre-computed) list of adjacent polygons, summing their normals as above, before finally normalizing the normal.

However, in newer architectures utilizing vertex shaders, often we want to initialize our vertex buffer once, and perform all vertex modification through shaders (i.e., on the GPU, not the CPU). Re-computing the vertex normals as described above using an adjacency list requires relative addressing (array indexing) and access to a large amount of data (all the vertices at once), two features that aren't readily supported by most vertex shader languages (though even this is changing as shaders become more flexible). Hence, this computation would happen on the CPU in your normal C/C++ code, and then you would feed these dynamically updated normals to your graphics card by locking down your vertex buffer and copying the new data in (potentially causing a stall in the pipeline).

### Surface Approximation

So how can we get around this? The answer is highly dependent on the needs and context of the application. However, if you're willing to sacrifice some accuracy in exchange for an extremely rapid and easy-to-implement alternative, then I propose a very simple approximation that is well suited to simple objects and vertex shader implementations.

If we can find an analytic function that sufficiently approximates our surface, then taking the partial derivative of this function will give us the normal of the surface at a given point, i.e.:

$$\nabla f(x,y,z) = (\delta f / \delta x, \delta f / \delta y, \delta f / \delta z) = \text{surface normal at the point on the surface } (x,y,z)$$

where it is assumed that  $f(x,y,z)=0$  for a point on the surface.

An analytic function is easy to express in a vertex shader; notice that the only dependence of the function is the point on the surface, i.e. the vertex position; but this is the natural input of a vertex shader. Of course, the reason we use polygonal meshes is because there is rarely an analytic function that can describe the sorts of shapes we have in games or other applications. Again, I stress the word *approximation*.

At the extreme, we can say that our object is represented by a sphere. Taking the partial derivative of the equation for a sphere centered at an arbitrary position yields:

$$f = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r^2 = 0$$

$$\nabla f = (\delta f / \delta x, \delta f / \delta y, \delta f / \delta z) = (2(x - C_x), 2(y - C_y), 2(z - C_z))$$

Since we are going to normalize the normal anyway, we can scale the above result by 0.5; notice that this yields just the point on the surface minus the center of the sphere! Of course, this is what we would expect intuitively, that the surface normal of a sphere would be in the direction along the line from the point on the sphere to the center. In our vertex shader, now all we need as extra input is the “center” of our object; calculating the normal then involves a vector subtraction and normalization.

An extension of this method could account for concave objects by performing a pre-process of the vertex list and associating with each vertex a sign of negative or positive one. This sign represents whether the vertex normal should point toward or away from the center. Computing this sign would depend on some heuristic, such as performing ray casts from the point to the center of the object and determining how many times the ray enters or exits the surface.

### **Surfaces using Regular Grids of Vertices**

Other times, we may have surfaces defined using a regular grid of vertices, such as terrain or a piece of cloth. In this case, we can use the original “correct” algorithm without an adjacency list. This is because we can generate in constant time the neighbors of a given face, simply using the  $(i,j)$ -th index of a given vertex. To minimize redundant polygon normal computations, first we would run through the entire grid and generate each polygon normal; then we would run through all the vertices of the grid, and access the adjacent polygons, summing their normals, then normalizing. This technique is not suitable for shaders, but it is fast and very accurate; in cases where the accuracy is necessary (such as a piece of cloth, or a deformable terrain), a surface approximation will likely not provide an adequate visualization; thus we resort to this grid based approach.