Team 20 -
Deepak Jeevan Kumar
Gene Shuman
Michael A. Brown

The SunSaver:
An OpenGL Visualization of the Sun's Surface

Abstract:

The sun is a source of immense power, giving rise to all life on earth. But most people don't realize that it is of great beauty in addition to this. The forces which govern the existence of the sun are subject to constant fluctuation, giving rise to field lines carrying charged particles, radiation winds, and perpetual flows of plasma churning up to the surface. The purpose of this project is to create a simple model of the sun's surface, with its changing characteristics and particle flows, and render it in realtime on consumer-level hardware. This last constraint prevents accurate scientific modeling of the system, but a visually-similar model based on a set of characteristics of the sun's surface is certainly within reach. The SunSaver accomplishes just this - it creates a discretely defined surface representing the sun, with a set of varying potentials governing the release of particles, and formations of arcs representing solar flares.

Introduction:

In presenting our visualization of the sun's dynamic surface, our group was intending to create an interesting scientific visualization which would be both interesting and readily available to the average computer user. Using C/C++ with GLUT and OpenGL, we maximized our portability with regards to platform, allowing development on both Windows machines and MacOS X. The visualization is intended to be of interest to any audience, as it is not an accurate scientific model. The ultimate goal of creating the SunSaver is to provide a unique multi-platform real-time visualization of a spectacular natural phenomenon which most people have never been exposed to.

With regards to the material presented in class, this project heavily utilizes the many aspects of object modeling, transformations and translations of object spaces with regards to world space, animation and interpolation, as well as simulation of a physical system by means of particles. As the project was done using OpenGL through the GL Utility Toolkit, the rendering pipeline is taken care of on the hardware side. However, knowledge of its function with regards to drawing polygons, coloring vertices, and visualization enhancements like polygon smoothing, were very necessary in creation of the final product.

Goals:

The ultimate goal of this project was to create a simple simulation of the sun's surface as governed by the physical properties of the sun itself. To do this, we separated the visualization into two categories - the surface, and the particles.

The surface of the sun is a triangular tessellation of a sphere, calculated in such a fashion that the distance to each vertex from the worldspace origin is equal. Each vertex on this surface has a set of properties which change over time, depending on its previous state, and influenced by its neighbors. In reality, the sun's surface changes depending on quite a few variables, such as temperature, magnetic field fluctuations, and charged plasmas flowing on and above the surface. In our representation, we wished to represent the forces governing these as a vector flow field across the surface of the sphere, with a magnitude and direction determining the next state. The visual representation of this, such as colors, particles, and arcs, is based upon both the current state of these vertices and their previous states.

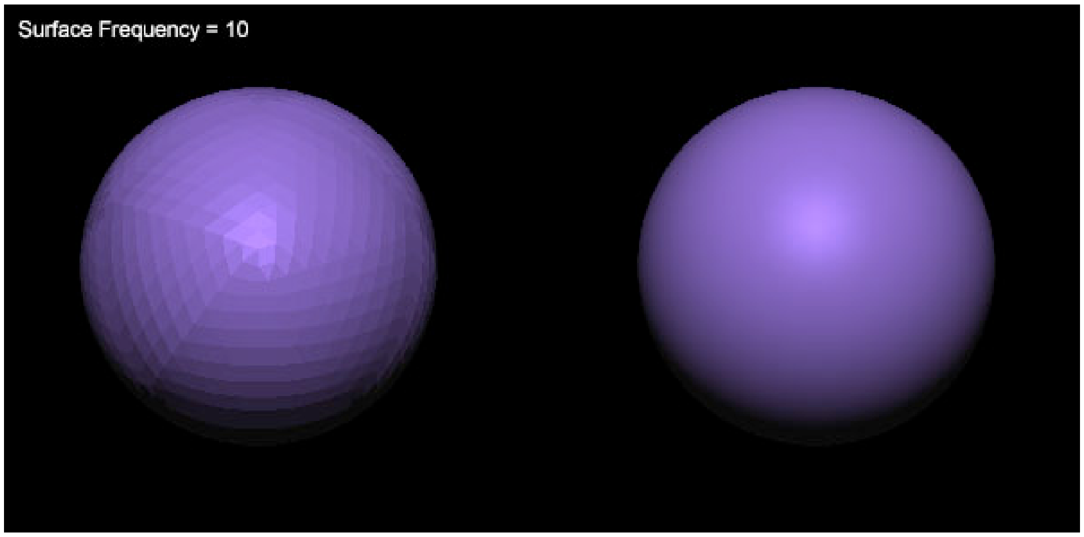
The particle aspect of this, as already stated, is determined by the current state of the surface. The particles exist in two forms - general prominence particles(constantly emitted), and arc particles. The former is a steady release of particles from the vertices whose quantity, color, and velocity are determined by the states of the vertex and its neighbors. As the magnitude of the potential at a vertex is increased, the number of particles allowed for this vertex is increased. Also, the magnitude of the velocity, as well as color, are altered accordingly to attempt to convey that the particles are highly charged. At the other end, vertices of low potentials emit very few particles at low velocities, with a darker color to represent their state.

In addition to the prominence particles, huge flows of particles can form over great distances on the sphere's surface. These are the solar flare arcs. An arc can be created when a vertex at its maximum potential, and a vertex at its minimum potential, are within a threshold distance. This does not necessarily mean that an arc is assured to form, but this is the only condition which allows for an arc to come into existence. If an arc forms, the particles are mapped to a dynamic path across the surface. Depending on the changes exhibited by the surface, the arc may or may not complete its path - it can fizzle out, blow off into space, or otherwise disperse.

One of the main things we wished to incorporate into this project was scalability. The desire to make a simulation which runs at 30 frames/second was of top priority, but the option to have a very finely meshed surface, and many particles, was of top priority. The sphere tessellator is very capable of representing surfaces ranging from a 20-triangle model to many, many thousands of triangles. The limitation on this detail is the complexity of the data, as each vertex has a defined set of neighbors which is pre-calculated, and each vertex likewise has its own particles to deal with. Instead of locking ourselves into a few options and building around them, we approached the design from the perspective of being able to set whatever details you wish to attain a balance of speed and quality.

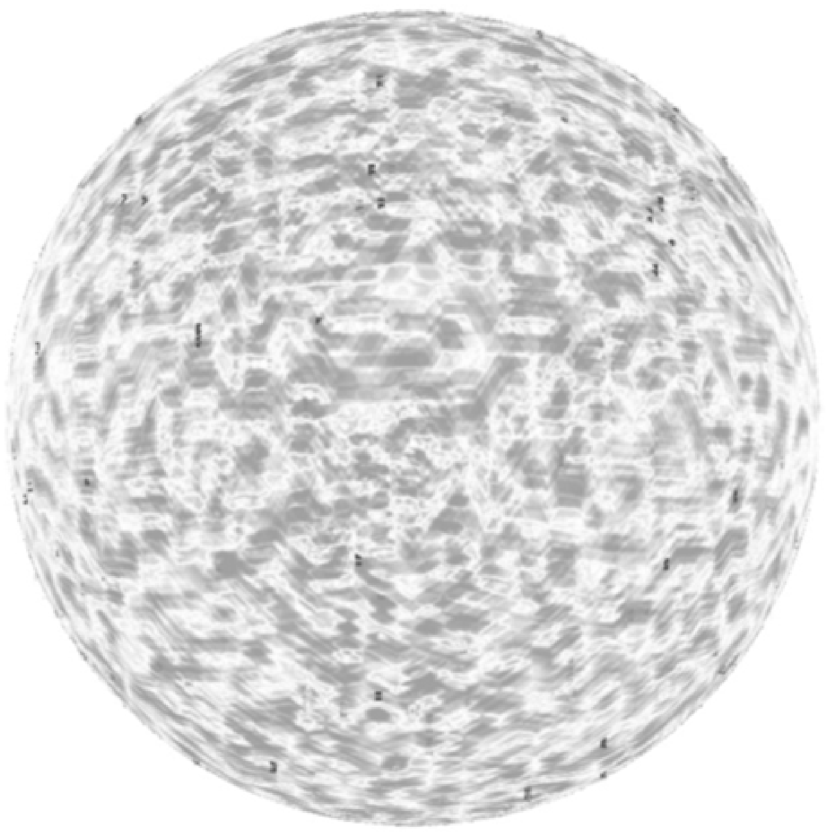
Achievements:

The tessellation of the sphere into a variable number of triangles works flawlessly. After struggling with some strange bugs early on involving improper grouping of vertices into their respective triangles, as well as some surface deforming due to artifacts from the tessellation, the end product works at any specified detail level, and will calculate neighbors of a vertex to a specified depth. Due to the fact that this is massively scalable, CPU limitations become very evident at high frequencies, mainly due to pre-processing of vertex coordinates, and initial structure creation(in the creation of our Surface() object, a detail frequency of 25 takes approximately 30 seconds to generate on a 1GHz G4 PowerMac).



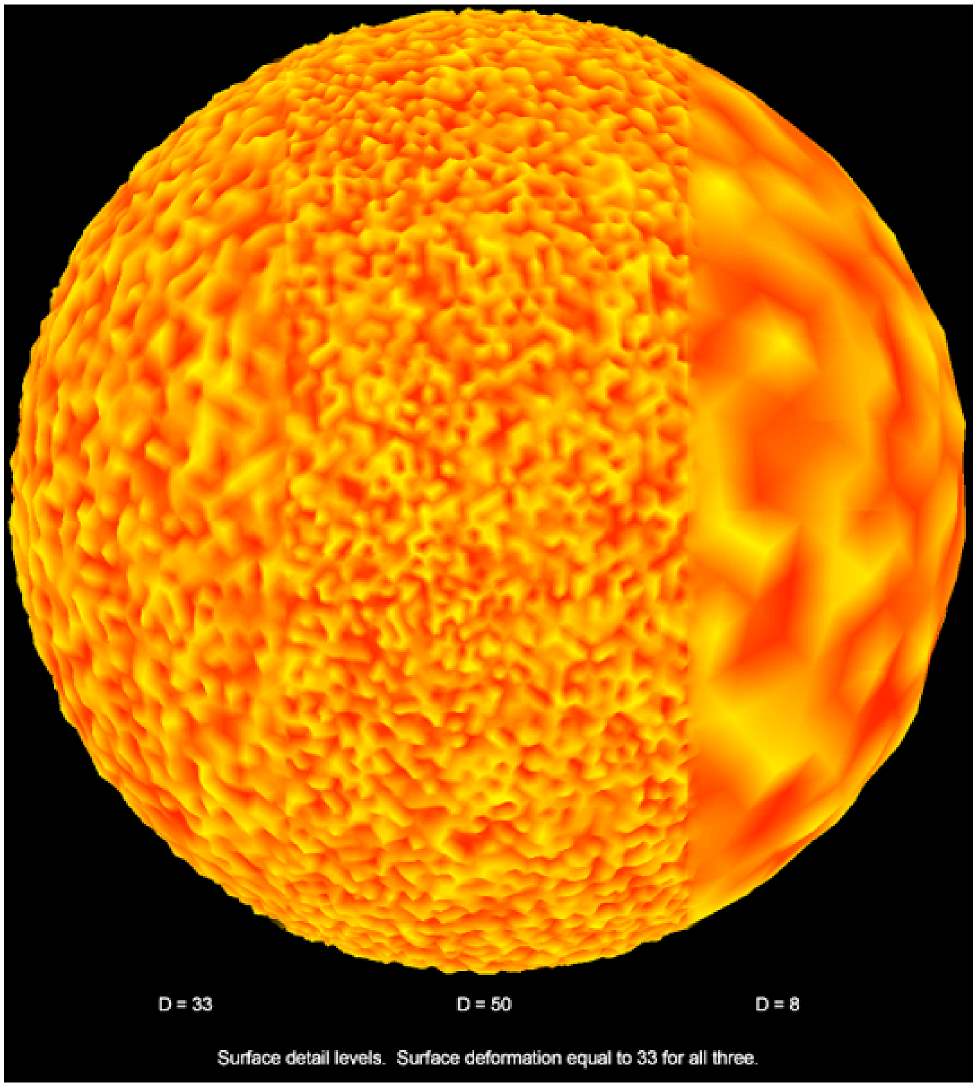
As coding began on the variables defining the characteristics of the sphere, our initial approach was specifying a vector tangent to the surface for each vertex. The magnitude and direction of this vector represented a potential flow on the surface, such that the potential of a specific vertex would be distributed proportionally over its neighbors. However, two major issues came up with this method. First, for reasonable detail levels of the surface, the triangulation of the surface prevented realistic looking flows across the surface. An analogy related to the course would be the issue of aliasing: mapping our continuous function to a discrete surface resulted in a very artificial-looking surface with completely unrealistic flows. The next issue was along similar lines, in that the triangulation of the surface was done such that the vertices were actually positioned such that they would be on a sphere's surface, not such that they would be equidistant from each other, or have the same number of neighbors. This likewise spawned interesting artifacts making the tessellation quite obvious. The result was a very geometric-looking flow, which didn't really accomplish what we had intended. At a finer mesh, these problems were less evident, but as this flow method was already quite computationally intensive, it didn't allow for us to achieve our real-time goal. Another peculiar issue with this method was a platform-related bug, in that the algorithm behaved erratically under MacOS X. Rather than debug and tweak this so early on in the project, which would have been quite extensive, we opted to trash it and make a visually-appealing replacement.

The resulting means of updating the sphere's properties was reduced to one variable, representing an outward flow potential for each vertex, plain and simple. The potential for each vertex is "semi-random," in that it's current state is a random fluctuation from a previous state, possibly influenced by a neighbor. This certainly doesn't show any sort of vector flow, but the resulting visual effects are much more true to the actual sun, especially at high frequencies.

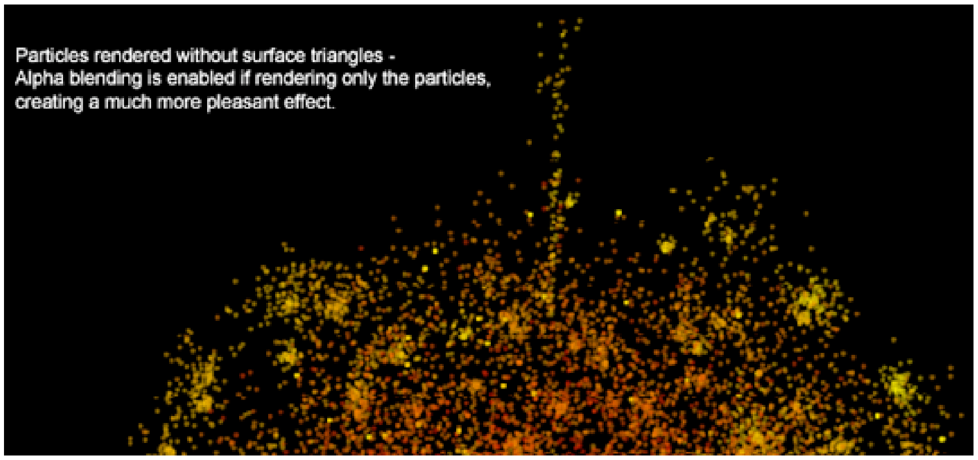


Composite image showing changes across six updates of the sphere surface.
Dots represent vertices at a minimum or maximum potential.

With the simplification of the potential algorithm, doing other updates to the vertices of the surface became less computationally taxing. This allowed for an impromptu feature addition of a deforming surface. When enabled, the potential of each vertex determines its distance from the origin within a specified range. In essence, it is just a little tweak to make the surface more interesting, but it is accurate to the actual sun's surface. Another interesting side-effect of this, straying away from the accuracy, is the ability to turn the sphere into a spikey, pulsating blob. This feature is amazingly interesting, but completely useless aside from being eye candy.


D = 33
D = 50
D = 8
Surface detail levels. Surface deformation equal to 33 for all three.

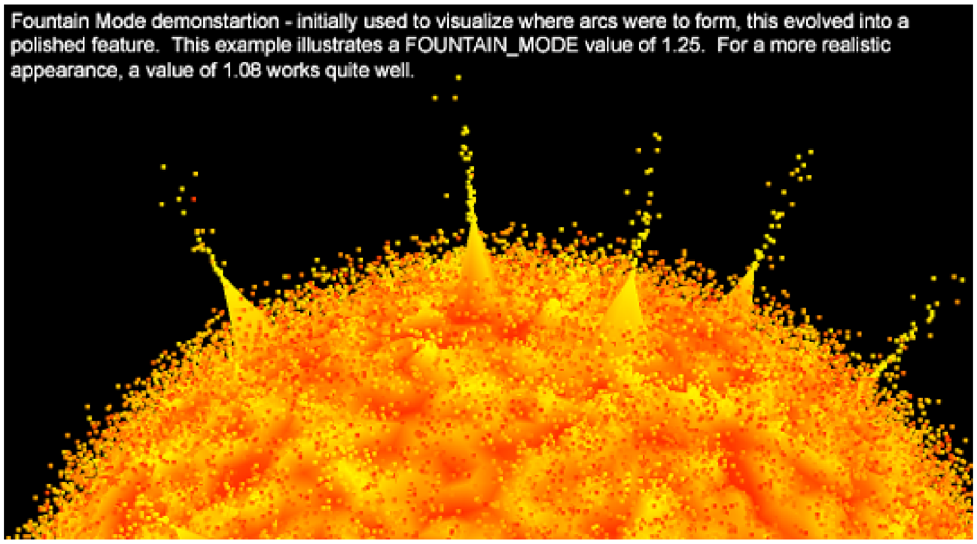
Also dependent on the vertex potential is the general prominence particle system. These particles are essentially for "surface fuzz" - they are created with a specified velocity and life span, such that they either fade away at a low height, or return to the surface and disappear. Only one major issue came up with this, and that is with regards to alpha blending. It was our original intent to have these particles be semi-transparent, to make the surface look very hazy, yet dynamic. The issue which stemmed from this was with regards to alpha blending our particles with the polygons on the surface. As the scene isn't rendered back to front, or front to back, alpha blending in combination with the depth test sometimes results in groups of pixels becoming completely transparent - i.e. they show through the whole scene, not just to the colors behind them. With large particles, this is completely unacceptable, and with small particles, the colors become muddy. Therefore, the only time alpha blending is performed is when ONLY particles are rendered, which can be set as a flag. Additionally, these local particles have one small shortcoming, in that the collision detection with the sphere is not a function of surface height - particles are nullified when they fall within the sphere's original radius.

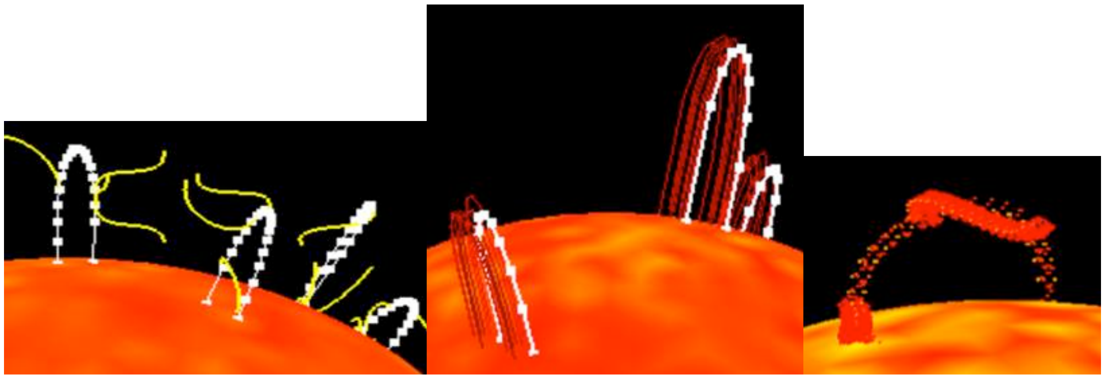

Particles rendered without surface triangles -
Alpha blending is enabled if rendering only the particles,
creating a much more pleasant effect.

Another shortcoming of the surface particle system is that it is completely CPU based as far as updating its state. Due to hardware and platform related issues, it wasn't really feasible to have a hardware-based particle system. The reason for this is that particle systems are built into DirectX for Windows, but MacOS X cannot use this, and its own particle extensions are vendor-specific.

The final part of our system is the solar flares - the arcs. As this feature happens to be quite complicated, our accomplishments for this have come in stages.

Initially, the arc starting points were implemented as a pronounced stream from an emission point. This was mostly for testing and demonstration purposes, but remains in the code as the "fountain mode" feature. Following this, we represented the actual paths that the arc would take over the sphere using a bezier curve, drawing the path and control points so as to visualize how the particles actually needed to be updated. Currently, we have not reached the points of having a steady stream of particles follow this line. We do have particles which follow it, but uniquely defining each particle's path such that it looks like an eruption has not yet been accomplished.


Fountain Mode demonstartion - initially used to visualize where arcs were to form, this evolved into a polished feature. This example illustrates a FOUNTAIN_MODE value of 1.25. For a more realistic appearance, a value of 1.08 works quite well.



Another quick addition made to the code, for the purpose of rendering to disk, was a quick function to grab the color buffer from the GPU, and dump it to a file. Specifically, the intent of this was to render frames of an animation to disk, and lump them into a video. This works fine, but with the one issue that it is an unformatted file - it's just an array of RGB values written into a file. These files are readable by some programs, but a batch conversion needed to be performed before rendering a video file from them. Implementing a movie capture function in this program was not high on the priority list.

The final part of our project is a simple menu system in the application itself, to set variables and toggle the current easter-egg type features. A simple version of this was written up in AppleScript, but not included because it isn't platform independent. A GLUT-based menu system should be fairly straightforward to implement, but likely will not be implemented in the included version.

In the timeframe allocated, we accomplished many of our goals, and produced a visually-appealing model of the sun's surface which runs in real time. The code is by no means polished, and could certainly be made more efficient. To get the project working, we wanted bug-free code, however mess it was. The GPU is unquestionably underutilized, and using hardware for the particle systems would certainly speed things up significantly, With more time, these issues would certainly be approached.

Also, the program is not ready for a public release by any means. As our intent is to make this into a screensaver on our respective platforms, the authors have every intent of implementing a GUI for setting properties of the visualization, and toggling of the features.

Contributions of Members:

Gene -
Gene's primary focus was the Surface() object, which tessellates the sphere's surface into the triangles. It also sets the appropriate neighbors for each SunPoint - the objects which define the vertices of the surface. He also implemented the vector-based flows of the surface. In essence, Gene is responsible for the underlying data structure for the system. Also, he has been working with arc creation and destruction through the Surface() object, as well as debugging the arc code to get the desired flow. Contributed to: Surface(), SunPoint(), SunTriangle(), SunArc(), main.cpp.

Mike -
Mike's primary focus was the drawing of the system, and visual effects. He maintained the drawing methods, OpenGL and GLUT settings, the coloring of the sphere's surface and particles, and the deformation of the surface. Also, he implemented the surface particle system, as well as the mentioned "fountain mode" for arc simulation. He's also responsible for the quick and dirty disk rendering. Contributed to: SunPoint(), SurfaceParticle(), main.cpp.

Deepak -
Deepak's primary focus was the implementation of the arc's paths, and particles associated with this. This was done through use of bezier curves between specified vertices, with control points to warp the path over time. Contributed to: SunArc(), main.cpp.

Lessons Learned:

Each member gained experience in using OpenGL, to some degree. Given the timeframe of our project, as well as the fact that we were learning how to use OpenGL as we went along, our introduction was certainly not thorough. Online documentation, from http://www.opengl.org and many other sites provide excellent descriptions of OpenGL and GLUT related functions. The authors also made extensive use of the OpenGL "Red Book," which is fairly dated at this point, but valuable for the basics. The only unfortunate part of OpenGL is its inconsistency across hardware and platforms. Extensions may only be implemented by a specific vendor, or a function may only be available for specific video hardware. This is fairly unavoidable, as the same can be said about implementations of c-compilers.

The most important lesson which we learned was probably in time management. The greatest problem with this project is that it wasn't a smooth progression towards a finished program, partially due to changing implementations upon realization of a problem. Given our knowledge of the problem at hand, in retrospect we could have made a cleaner, more efficient implementation. The bottom-up approach to this problem seemed like the best way to structure it, but this introduced some inefficiencies as far as drawing and manipulating the objects. Also, given the timeframe for completion of the project, we usually had no choice but to plow forward when we encountered a problem. Restructuring a central part of the code would have been a huge setback, although the end result would most likely be better.

Description of Deliverables:

For your reviewing, two sets of source code are provided. The first is without the arcs implemented, but everything else is as mentioned. This is being submitted as the code is very well tested, and stable across our machines. This source is available at:
http://static-soul.mit.edu/~oranje/Solar/SunSaver-sourcecode.tgz

The second has the arc paths implemented, but is not yet in its completed form. However, it shows how we were approaching the problem. For our presentation, we hope to have these issues worked out. This code is located at:
http://static-soul.mit.edu/~oranje/Solar/SunSaver-sourcecode.zip

Additionally, three rendered sequences are available at:
http://static-soul.mit.edu/~oranje/Solar/Examples/

detail50-withParticles.mov - This is rendered at detail level 50, which takes several minutes of pre-processing to begin. It is not reasonable to run the detail at this level for normal use. Additionally, fountain mode is enabled at 1.08, and fountains only is enabled as well so as to keep clutter of the surface down. Each vertex has a maximum of 100 particles.

detail23-particlesOnly.mov - This file shows the particle system on its own at detail level 23, with fountain mode enabled at 1.1. Fountains only is disabled. Alpha blending is also enabled for this, and is enabled automatically if you're rendering only the particles.

perpetual-implosion.mov - This is an interesting capture created to show the versatility of the particle system, and the representation as a whole. The application was set to detail 13, particles only, 300 particles per vertex, deformable surface with height ratio of 0.1, and fountain mode enabled at 1.1. The result is best described as a perpetual falling of particles towards the origin. The effect is quite neat, but represents nothing in terms of the project.

Quicktime is required for playback. As the files are rather large, you may choose to wait for a physical copy of the movies on a DVD, to be shown and handed in at the time of our presentation.

Acknowledgments:

Much of the inspiration for this project, as well as idea searching, came from the Solar and Heliospheric Observatory's website, located at http://sohowww.nascom.nasa.gov/. This has a great deal of data, and learning materials about theories of the sun's function.

All source code for this project is original. Countless websites were referred to in learning how to use OpenGL, but code was based upon nothing from these websites. The only thing not original is lighting and material information, which was only used for initial testing purposes for correctness of normals and debugging the Surface() object. This unused code was most likely taken from The OpenGL Programming Guide, third edition (the red book).

Appendix:

About the state variables -
As a platform-independent GUI was not a part of this release, changing the settings of the program must be done before compile-time through state variables. All of these were implemented in this fashion because most we wanted constant for development, and others just became convenient. Setting these as default-values in a GUI would be quite simple.

In main.cpp ->

FRAMERATE - Misnamed. Actually represents frame duration in milliseconds.

DETAIL - Sets detail level of Surface() tessellation. Use carefully. Values below 10 are very low detail. Somewhere between 15 and 25, with particles on, renders quickly and looks quite good. When you go above 25, the preprocessing time starts to get quite long, and the program likewise gets much slower. Must be an integer greater than or equal to 1.

NEIGHBOR_INFLUENCE - Sets depth of exploring neighbors for each vertex. Any integer >= 0 is valid. Setting to 0 results in particles shooting straight out from the sphere, i.e. they aren't influenced by neighboring vertices.

PARTICLE_SIZE - Particle size in pixels. I like 2.33 for this. When using PARTICLES_ONLY mode, 5.0 or so looks good.

PARTICLES_ONLY - Boolean. If enabled, the triangles defining the surface aren't drawn.

WRITE_IMAGE_SEQUENCE - Boolean. If enabled, a series of RAW images will be written to the executable's directory. Use at your own risk.

WRITE_IMAGE_SEQUENCE_NUM - Specify frames to write to disk. Integer values greater than zero are accepted.

WIDTH - Width of display window, positive integer.

HEIGHT - Height of display window, looks best if equal to WIDTH

PIXEL_FRAME - Defines width of border to be removed for writing images to disk. Must be an integer value greater than or equal to zero. This has no effect on the rendering.

In SunPoint.h ->
NUM_PARTICLES - Integer value greater than or equal to 0. This defines the maximum number of particles defined for each vertex of the sphere.

DEFORM_SURFACE - Boolean. If set, the surface will deform according to the potentials of each vertex. Very neat, not computationally expensive. Should leave at 1.

HEIGHT_RATIO - Best described as the deforming scale. Larger numbers correspond to less deforming. Distance from (0,0,0) = 1 +/- 1/HEIGHT_RATIO. For representing the sun, a value of 33.0 is good. It accepts any non-zero number. Negative numbers make the deformation backwards. Numbers from -1 to +1 result in REALLY, REALLY NEAT effects, such as a spikeball, and an inverted spikeball. Useless but neat.

FOUNTAIN_MODE - Enabled if value is greater than 1.0. This number also represents the scalar factor for height and particles of vertices at maximum potential. For example, setting to 1.5 would cause vertices at maximum potential to extend to 1.5 times their normal distance from the sphere. Causes a little eruption of particles, if enabled. A good value is 1.07. If Arcs are enabled, this should be set to 0 to disable it.

FOUNTAINS_ONLY - Boolean. If enabled, surface particle numbers are sharply decreased everywhere but at vertices with maximum potential. If using a high detail level for the sphere surface, using fountain mode with this enabled speeds up the program slightly, as well as making it more visually appealing.