A Viewfactor-based Radiative Heat Transfer Model for Telluride

Austin Minnich
UC Berkeley
John Turner
LANL
Michael Hall
LANL

August 7, 2002

Abstract

In this paper we describe an implementation of a viewfactor-based radiative heat transfer model for Telluride. A viewfactor (also known as a form factor or geometric configuration factor) is a dimensionless factor that generally describes how much of an object is visible to another object based solely upon the geometry of the situation. For our purposes this will be defined as the amount of energy leaving an area $A_i$that is directly intercepted by A_j, or:

F_{ij} = energy leaving A_i and directly intercepted by A_j / energy leaving A_i

where F_{ij} is the viewfactor from A_i to A_j.

Our algorithm relies on a method commonly used for graphics applications called *radiosity.* The fundamental principle of this method is that the radiosity, or total energy leaving each face is equal to the emitted energy plus the sum of the reflected energies. This principle leads to a system of linear equations, the solution of which is the final radiosity of each face. Using this technique, the algorithm inputs a list of faces, calculates the viewfactors for every face, solves the resulting system of equations and returns a matrix which contains the radiant flux of each face. The algorithm is used in the Telluride project the calculate the equilibrium state of the mold after the mold has been heated. This equilibrium state is then passed into Telluride as the initial condition of the mold.

1 Introduction

Currently, Telluride models heat flow with the boundary condition:

$$Q_i = \sigma A_i K (T_i^4 - T_{amb}^4) \tag{1}$$

where Q is the rate of radiative heat transfer from a surface j at temperature T_j, T_{amb} is an ambient temperature to which the surface radiates, and K is a constant

which takes into account the effects of emissivity and viewfactors [5]. In fact, the actual interaction in the system is much more complicated, with every face being dependent on the state of every other face. The method our algorithm uses to determine the radiant flux of each face is a technique commonly used in graphics rendering called *radiosity.* The radiosity, or the total energy leaving each face, is equal to the emitted energies plus the reflected energies, or:

$$B_{total} = \sigma e T_i^4 + \rho_i \sum B_j F_{ij} \tag{2}$$

where B_i is the emitted energy, ρ_i is the reflectance of the face i, $\sum B_j F_{ij}$ is the incident flux, and B_{total} is the final radiosity of face i. [5]

How much radiation actually reaches face i from face j is dependent on the viewfactor, F_{ij}. In heat transfer a viewfactor is defined as the amount of energy leaving A_i that is directly intercepted by another area A_j (see Figure 2).

Using this principle and the ideas from radiosity, we see that this leads to a system of equations of the form:

$$\begin{bmatrix} 1 & -\rho F_{12} & -\rho F_{13} & ... & -\rho F_{1n} \\ -\rho F_{21} & 1 & -\rho F_{23} & ... & -\rho F_{2n} \\ -\rho F_{31} & -\rho F_{32} & 1 & ... & -\rho F_{3n} \\ -\rho F_{n1} & ... & ... & ... & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ ... \\ B_n \end{bmatrix} = \begin{bmatrix} \sigma e T_1^4 \\ \sigma e T_2^4 \\ \sigma e T_3^4 \\ ... \\ \sigma e T_n^4 \end{bmatrix}$$

where F_{ij} is the viewfactor from face i to face j. Note that this system is guaranteed to converge by Gauss-Seidel because it is diagonal dominant.

To relate the radiosity back to the net radiant flux, we use:

$$q_i = B_i - I_i \tag{3}$$

$$I_i = \sum B_j F_{ij} \tag{4}$$

where B_i is the outgoing radiant flux, and I_i is the incident radiant flux. [5]

The temperature of each face is determined by the relation

$$\alpha \frac{\delta T}{\delta t} = q_{rad} + q_{conv} + q_{cond} \tag{5}$$

where α is a conversion constant, q_{conv} term is convection, q_{cond} is conduction, and q_{rad} is radiation. The convection and conduction modules are already in Telluride; the radiation term is determined by this algorithm.

2 Viewfactors

2.1 The definition of viewfactors [2]

There are a few ways to arrive at the mathematical definition of a viewfactor, but because our algorithm is used for heat transfer, we define everything in terms of energy and heat, instead of other terms used by those in graphics.

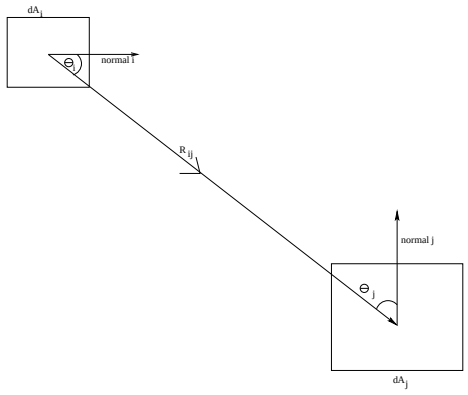


Figure 1: Example situation between two differential areas

To derive the definition of a viewfactor we first define two differential areas, dA_i and dA_j, which are joined by the vector R_{ij} (see Figure 1).

θ_i is thus the angle between R_{ij} and the normal, n_i, to dA_i, with θ_j being defined likewise.

It can be shown that the distribution of flux from a surface varies according to the cosine of the angle from the normal(Lambert's Cosine Law). Using this principle, we can state:

$$q_\theta = \frac{q}{\pi}\cos\theta = \frac{\sigma T^4}{\pi}\cos\theta$$

where q_θ is the energy radiating at an angle θ to the normal.

Therefore, the energy radiating through a solid angle $\sin\theta d\theta d\phi$ at an angle θ is:

$$\frac{\sigma T^4}{\pi}\cos\theta\sin\theta d\theta d\phi$$

Next, we determine the fraction of the solid angle area taken up by dA_j (see figure 2).

This area is equal to the component of the area of dA_j in the plane of the solid angle, $dA_j\cos\theta$, divided by the solid angle a distance R_{ij} from the source, $R_{ij}^2\sin\theta d\theta d\phi$, or:

fraction of area taken up by $dA_j = \frac{dA_j\cos\theta}{R_{ij}^2\sin\theta d\theta d\phi}$

The energy emitted by dA_i and directly intercepted by dA_j is therefore:

$$Q_{ij} = \frac{\sigma T^4\cos\theta_i\cos\theta_j dA_i dA_j}{\pi R_{ij}^2} \tag{6}$$

Since σT^4 represents the energy emitted, the rest of the equation describes the fraction of energy emitted from dA_i and directly intercepted by dA_j, or the viewfactor. Therefore, the viewfactor between two differential areas is defined as:

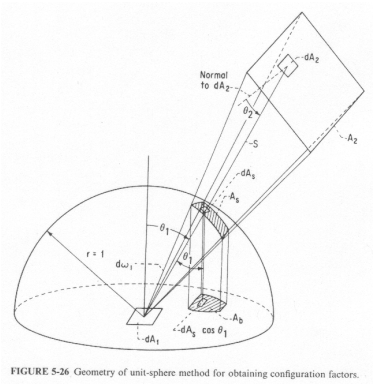


FIGURE 5-26 Geometry of unit-sphere method for obtaining configuration factors.

Figure 2: The Nusselt unit sphere [6]

$$dF_{ij} = \frac{\cos\theta_i \cos\theta_j dA_i dA_j}{\pi R_{ij}^2} \tag{7}$$

We leave the definition in its differential form because the algorithm evaluates it in this form, making the assumption that each mesh face is approximately a differential area. The formal definition of a viewfactor from one finite area to another is simply the double integral of (7) over i and j.

Additionally, we can derive two important relationships. The *reciprocity relationship* states that dF_{ij} and dF_{ji} are related by:

$$dF_{ij} dA_i = dF_{ji} dA_j \tag{8}$$

Secondly, for a closed system,

$$\sum F_{ij} = 1 \tag{9}$$

This shows that when the system is closed to the surroundings, every face must be seen by every other face. The consequences of this relationship with be discussed elsewhere in the paper.

2.2 Evaluating Viewfactors

There are a few methods currently used to evaluate viewfactors; most are expensive to run and are used primarily to construct images for computer graphics. These methods are extremely detailed, as they try to reconstruct a visually realistic scene (see [8]). This level of detail, for our purposes, is unnecessary. To evaluate the viewfactors, we calculate (7) from every face i to every other visible face j. These viewfactors are then stored in a matrix.

The algorithm evaluates (7) by treating all the terms as vectors. To arrive at this alternate definition of viewfactor, we rewrite (7) as follows:

$$dF_{ij} = \frac{\|A_i\|\|A_j\| \cos\theta_i \cos\theta_j}{\pi \|R_{ij}\|^2}$$

where A_i and A_j are now area vectors normal to their respective faces.
The dot product of any two vectors is:

$$AB = \|A\|\|B\| \cos\theta$$

If we let B equal the vector joining the two areas, or R_{ij} we see that:

$$\frac{AR_{ij}}{\|R_{ij}\|} = \|A\| \cos\theta = Au_r$$

where u_r is the unit vector of $R_{ij} = \frac{R_{ij}}{\|R_{ij}\|}$. Our definition of viewfactor thus becomes:

$$dF_{ij} = \frac{A_i u_r A_j u_r}{\pi \|R_{ij}\|^2} \tag{10}$$

This expression is evaluated from every face i to every other visible face j to form the viewfactor matrix.

3 The Algorithm

The primary purpose of the algorithm is to return a matrix containing the final radiant flux of each face. To do this, we first construct the viewfactor matrix by evaluating (10) for every pair of visible faces. This is the most complicated part of the algorithm: we cannot simply calculate (10) for every face. We must first ensure that the faces are visible to each other. We have many tests to exclude those faces that do not participate in the calculation which will be described below. Once the viewfactor matrix is constructed, calculating the final radiosity is fairly simple. The algorithm solves the system of equations formed from the viewfactor matrix and outputs the final radiant flux matrix (see Figure 3).

3.1 The Tests

3.1.1 The Dot Product Tests

Because we evaluate the viewfactor using vectors, we must remember that the area vector for each face does have a direction- it is not merely a scalar area term. Care must be taken to ensure that the area vector points in the proper direction as determined by the problem. Currently, the area vector is determined according to a right-hand coordinate system, meaning in an enclosure radiation problem the area vector would point into the enclosure. Right-hand rules will probably be correct in most, if not all, problems.

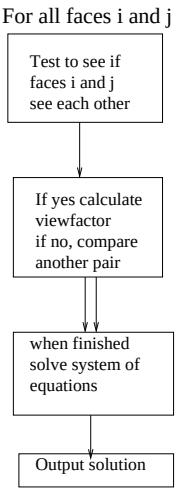


Figure 3: Flow Chart of the algorithm

We can use the direction component of the area vector to our advantage: because of the direction component, we can easily rule out those faces which do not point towards each other. To do this, we simply take the dot product of the area vectors of the two faces in consideration: if the dot product is negative, then the two faces are not pointing in the same direction; positive, they are facing in the same direction and thus do not participate in the calculations. This is the first dot product test(see Figures 4 and 5).

Figure 4: Dot product is negative between the two area vectors, so they face each other

Figure 5: Dot product is positive, so one of the faces is in the wrong direction.

This method does contain one major shortfall that must be remedied before we can be completely sure that two faces participate in the calculation. The problem lies in the evaluation of the dot product. The dot product does not have a direction term; that is, two vectors which are pointing towards each other and two vectors which are pointing in opposite directions have the same dot product. Thus, if we were to use only the first

dot product test, vectors pointing towards each other would correctly pass the test, but vectors pointing away from each other would incorrectly pass the test (see Figure 6).

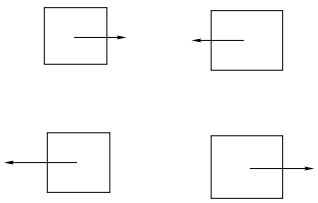


Figure 6: The top example would correctly pass the first dot product test, but the bottom example would incorrectly pass. We solve this using the second dot product test.

To solve this problem, we make use of the second dot product test.Using R_{ij}, the vector between the the faces defined in the previous section, we take the two area vectors, A_i and A_j, and take the dot product of them with R_{ij}. If A_i and A_j are facing each other, then, depending on how R_{ij} was defined, the dot product of A_i with R_{ij} should be positive, and the dot product of A_j with R_{ij} should be negative(Assuming R_{ij} was defined as $s_i - s_j$, where s is the position of each face). Through this test, we are able to differentiate between the faces that are pointing in the same direction and those that are pointing in opposite directions.

3.1.2 The Occlusion Test

By using the two dot product tests, we know that the faces are pointing towards each other. Before we can proceed, however, we must consider one more problem: occlusion. Occlusion is defined as the situation that occurs when one face partly or completely obscures the view of another face. If a face j were occluded by another face k, our current algorithm would incorrectly include face j in the calculation even though face k is in the way and thus prevents face j from participating in the calculation(see Figure 7).

Clearly, the algorithm requires an occlusion test to determine whether the face in question is occluded by another face. Unfortunately, while easy for the human eye to discern occlusions of this type, it is extremely difficult for a computer: the routine

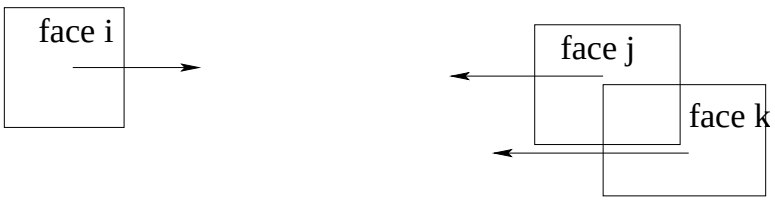


Figure 7: Example of a possible occlusion.

which implements this makes up more than half of the program. There are a few ways to discern whether one face occludes another; most involve further discretizing the differential area into small subelements called pixels and projecting each face onto the pixel grid(see [8], [3]). These methods are mainly used in graphics where a high level of detail is necessary to construct a visually realistic scene. For our purposes, this level of detail is unnecessary. We instead use a modified version of this scheme: rather than discretizing the differential area and projecting each face onto the grid, we simply take two faces, project them onto a two-dimensional plane, and test to see if the polygons overlap.

To understand how we accomplish this, consider three faces: i, j, and k. In this situation, we are testing to see if, from the point of view of face i, there is any face k that occludes face j. First, we rotate the coordinate system so that it is centered around face i. Next, we convert the vertices of each face j and k to the new coordinate system. Now that everything is in terms of the new coordinate system, we convert to spherical coordinates: equivalent to mapping each face j and k onto Nusselt's unit sphere [6] (see Figure 2). Then, to avoid problems with the cyclical nature of spherical coordinates, we convert back into cartesian coordinates, but with the radius term set to unity. This effectively disposes of the distance element of the coordinate system and allows us to consider the faces j and k solely in the xy plane. Finally, we can test to see if face k occludes face j (for a chart of this process see Figure 8).

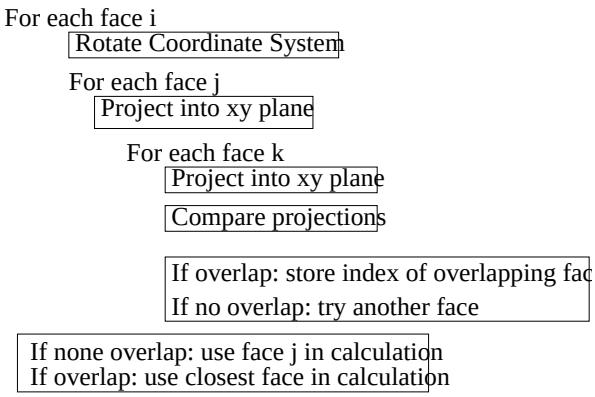
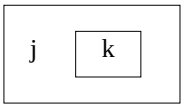


Figure 8: flow chart for the occlusion subroutine.

While it was fairly difficult to get into the position to test whether two faces occlude, it is even more difficult to test whether two faces do in fact occlude. To do this we have several tests which handle all the possible cases in polygon intersection. Here we refer to the faces j and k as polygons, since our tests rely on computational geometry algorithms to determine whether or not two faces occlude.

The first test determines whether the center of one polygon is in the center of the other polygon. We compute the center of one polygon, and use a common crossings test algorithm to test. The crossings algorithm essentially counts the number of intersections of a horizontal ray determined by the center point with all the lines of the

First Occlusion Test Example:



Second Occlusion Test Example

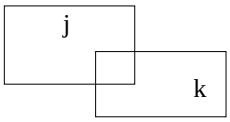


Figure 9: Possible cases for comparison of polygons in the occlusion subroutine.

polygon. If there are an even number of crossings then the center is outside the polygon; odd number, it is inside. This test handles cases where one polygon might be completely inside the other polygon.

The second test checks to see if any of the lines forming the polygons intersect. To do this we simply take each line of one polygon, and check to see if it intersects with any of the lines forming the second polygon. If there are two intersections in different locations, then the faces occlude(see Figure 9).

Finally, if none of these checks have been set, then we know that the faces do not occlude. We perform this for every face i, j, and k that have passed the dot product tests.

3.2 Putting it Together: The Viewfactor Matrix

While it may seem that we have just described many independent tests, all of these tests actually work together in order to form the viewfactor matrix, the basis for the system of linear equation and ultimately the final radiosity of each face. All the tests contribute to the viewfactor matrix. First, the dot product test is applied(taking the dot product of the area vectors). If the dot product is positive, then the faces are pointing in the same direction, are not visible to each other, and the viewfactor matrix element corresponding to face i and face j is set to zero. If the faces pass this test, then the second dot product test is applied(taking the dot product of each area vector with the vector joining the two faces). If this test fails, the element of the viewfactor matrix is set to zero. Otherwise, we check to see if another face occludes the face in consideration. If the face is not occluded, then we calculate the viewfactor and put it in the viewfactor matrix. If the face is occluded, we calculate the viewfactor for the occluding face and set the viewfactor for the occluded face to zero. We are also able to account for multiple occlusions: should multiple faces occlude the same face, the routine calculates the viewfactor for closest face.

Note that this does not allow for partial occlusions: a face is either completely

occluded or not occluded at all. Currently, if a face is only slightly occluded then this is counted as a complete occlusion even though the occluding face may occlude only a small fraction of the occluded face (see Figure 10).

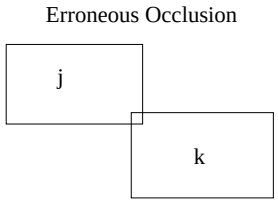


Figure 10: Our occlusion subroutine will register this as an occlusion even though face k occludes very little of face j.

4 Relating Radiosity and Temperature

Once we have the viewfactor matrix, we must solve the linear system of equations. The solution of this system is the final radiosity of each face, or the total energy leaving each face due to the emitted and reflected energies. Once we have this radiosity matrix there is one additional step required to determine the final temperature distribution of the mold. Using the relation:

$$q_{rad} = B_i - I_i \tag{11}$$

where B_i is the radiosity matrix we have just calculated, and I_i is the incident flux equal to:

$$I_i = \sum B_j F_{ij} \tag{12}$$

Once we have determined q_{rad} we can determine the change in temperature by:

$$\alpha \frac{\delta T}{\delta t} = q_{conv} + q_{cond} + q_{rad} \tag{13}$$

where α is a conversion constant, q_{conv} is the convection term, q_{cond} is the conduction term, and q_{rad} is the radiation term determined by our algorithm.

4.1 Conservation of energy

Because this algorithm will be used for heat transfer, conservation of energy must be taken into account. This is accomplished in the viewfactor matrix. When we introduced the definition of viewfactor, we also stated that, for a closed system:

$$\sum F_{ij} = 1 \tag{14}$$

This relation simply states that all faces must see each other. However, for most of the problems for which this algorithm will be used, the system will not be closed: often there will be an opening at an ambient temperature. Thus F_{ij} cannot sum to unity because the system is not closed. In addition, our occlusion algorithm does not conserve energy, as there is no provision for partial occlusions. To solve both these problems, we first sum F_{ij}. We then subtract this sum from unity. This gives a viewfactor which represents the fraction of energy for a particular face that was either not conserved in the occlusion algorithm or goes to an ambient surface. We treat this fraction of energy as if it were all radiating to an ambient temperature. At the end of the calculation we have an ambient energy term which represents the energy radiated to the ambient surface. In this way we ensure that energy is conserved.

5 Runtime

This algorithm is extremely expensive: when most faces are visible to each other the runtime is close to n^3. This is because we necessarily use three loops in order to incorporate occlusion into our algorithm: two loops for faces i and j, and one additional loop to check for occlusion with face k. This algorithm is also expensive in terms of memory because of the viewfactor matrix. Most of the problems for Telluride will contain thousands of faces: not prohibitively large, but definitely expensive. If the geometry of the problem is such that most faces do not see each other then the runtime will approach n^2, because most of the faces will not be involved in the calculations. The viewfactor matrix will thus become more sparse by the same reason, easing memory requirements.

6 Conclusion

In this paper we have demonstrated how we implemented a viewfactor based radiative heat transfer model for Telluride. Our algorithm incorporates ideas from the primarily graphics-oriented radiosity methods, using the original ideas and concepts but changing the exact implementation to meet our needs. Some changes include the calculation of the viewfactors and our algorithm for the occlusion problem. We also relate radiosity back to heat transfer: an idea not used by graphics radiosity. Our algorithm will be used to determine the final temperature distribution of the mold for Telluride, which will then be passed into the main code for calculations. In the future, we hope to make our occlusion test more accurate and include partial occlusions.

References

[1] Ian Ashdown. Eigenvector radiosity. Master's thesis, The University of British Columbia, 2001.

[2] R. B. Bird, W. E. Stewart, and E. N. Lightfoot. *Transport Phenomenon*. John Wiley & Sons, second edition, 2002.

[3] Steven M. Drucker. Radiosity: An illuminating perspective. Technical report, Media Laboratory, Massachusetts Insitute of Technology, 1992.

[4] A. F. Emery, O. Johansson, M. Lobo, and A. Abrous. A comparative study of methods for computing the diffuse radiation viewfactors for complex structures. *Journal of Heat Transfer*, 113:413–422, May 1991.

[5] Kin Lam. An enclosure radiation model for telluride. proposed model implemented by this algorithm, 2001.

[6] A. Mavroulakis and A. Trombe. A new semianalytical algorithm for calculating diffuse plane view factors. *Journal of Heat Transfer*, 120:279–282, February 1998.

[7] Michael F. Modest. *Radiative Heat Transfer*. McGraw-Hill, Inc., 1993.

[8] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.