Proceedings of BS2013:
13th Conference of International Building Performance Simulation Association, Chambéry, France, August 26-28

A FULLY AUTOMATED CALCULATION OF SHADOW CASTING WITH MATRIX-BASED COORDINATE TRANSFORMATIONS AND POLYGON CLIPPING

Matthias Gladt[1] and Thomas Bednar[1]
[1]Vienna University of Technology, Vienna Austria

ABSTRACT

Short wave radiation has a significant influence on a building's thermal behavior. The calculation of the projected sunlit surface fraction is therefore an indispensable prerequisite for running a meaningful simulation. It consists of at least two major steps. One is to solve the visibility problem of all surfaces of the building that is investigated. This means to set up an order in which one surface may cast a shadow on another.

The other step is to calculate the projected sunlit surface fraction based on the order obtained in the first step.

There are basically two technologies to address both problems. One is based on OpenGL and the other one on polygon clipping algorithms. The paper at hand will outline a polygon-based implementation and give the pros and cons compared with an OpenGL-based implementation.

INTRODUCTION

The visibility problem as well as the need for the calculation of the projected sunlit surface fraction (PSSF) usually occurs in the course of a building simulation where short wave radiation shall be taken into account. Both issues have been addressed in papers as early as in the 1970s. Walton (1979) compares two algorithms of calculating the shadow. One, the "discrete element analysis" is the basis for modern GPU-based calculations, the other, "overlapping polygons", focuses on finding a more accurate solution for the calculation of the PSSF. Newell, Newell and Sancha outline a method for solving the hidden surface problem (1972) which is further developed by Weiler and Artherton (1978). Artherton, Weiler and Greenberg (1977) explain how these methods can be integrated with computer-aided design (CAD) rendering at the time. Grau and Johnson (1995) show how Weiler and Artherton's polygon clipping algorithm (1978) can be implemented.

Even with modern computers polygon clipping remains the most consuming part of the calculation if accurate results are required. Therefore efficient polygon clipping algorithms are vital for a fast computation. Vatti (1992) presents a polygon clipping algorithm on which some modern implementations are based (e.g. Murta, 2009). Although Vatti's algorithm can still be slightly improved (Greiner and Hormann, 1998) the consumption of CPU time grows with the average number of intersections of polygons on the order of O(*nm*) where *n* and *m* denote the numbers of edges of the polygons to be intersected.

The calculation of the PSSF consists of several steps. Most of them have already been dealt with in literature but hardly any publications exist outlining the whole process from providing an interface for the input data over processing all steps needed to calculate the PSSF to returning the results in a usable form.

This paper will outline in detail how this goal can be achieved by the means of a modern programming language in general and by the example of the Python programming language in particular.

It will also show how a new implementation can be based on some sophisticated solutions for the individual steps to calculate the PSSF and how an easy integration with existing building simulations can be achieved.

Furthermore it will point out where alternatives to the solution provided for a particular step of the calculation exist and it will elaborate on the pros and cons of these alternatives.

POLYGON-BASED APPROACH

State of the art

Jones, Greenberg and Pratt (2011) define the projected sunlit surface fraction (PSSF) as

$$I_B \left(A_S / A_T \right) \cos\theta \tag{1}$$

where I_B is the intensity of radiation (i.e. the sun-light), A_S the sunlit surface area, A_T the total surface area and θ the angle of incidence of sun's rays with respect to the surface's normal vector. A_S on its part is defined as

$$A_S = A_T - A_D \tag{2}$$

where A_D (D...dark) is the part of A_T that a shadow is cast on.

A_D is the factor to be calculated with the help of the implementation at hand. Since the sunlight is in the focus of interest rather than one or more spot lights all rays can be assumed parallel.

A sample scene (Figure 1), shall be used to illustrate the following approach. There are two faces, one casting a shadow on the other. Although the example at hand is a simple one all considerations are equally valid for more complex scenarios.

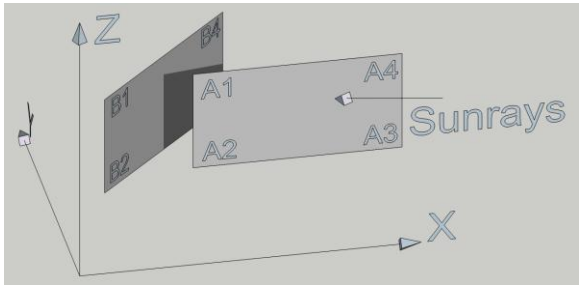

Figure 1 Scene with two surfaces, one casting a shadow on the other

Shadow mapping

Williams (1978) proposes to render the scene from the position of the light and to generate a map with depth values representing the distance of a particular object from the source of light. The values of the so called "depth map" are updated whenever an object is added to the scene whose distance from the source of light is smaller than the current value from the depth map. In the end the depth map holds the values representing the distance(s) of the foremost object(s) seen from the position of the light.

We adopt the first part of Williams' idea to render the scene from the light's point of view (Figure 2) but we modify his approach in which he stores a complete depth map.

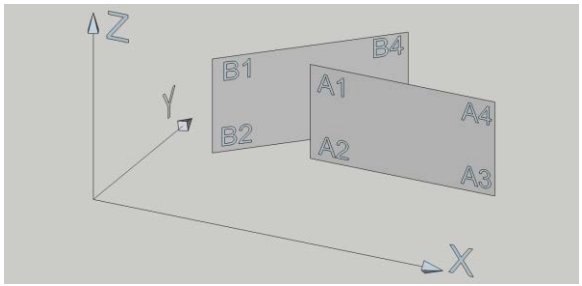

Figure 2 Same scene from light's point of view. Rendering the scene from the light's point of view provides all faces exposed to direct sunlight whereas any other faces, not visible from this point, remain in the shadow

Instead we will only compare depth values at some points and use it as a basis for the depth sort of all objects.

Weiler's and Artherton's hidden surface removal

Weiler and Artherton (1978) present a method to set up an order based upon the depth values of the individual polygons where the polygons represent the enveloping surfaces of an object. For the sake of simplification only flat faces were used in Figure 3 which can also be thought of as detached slim walls.

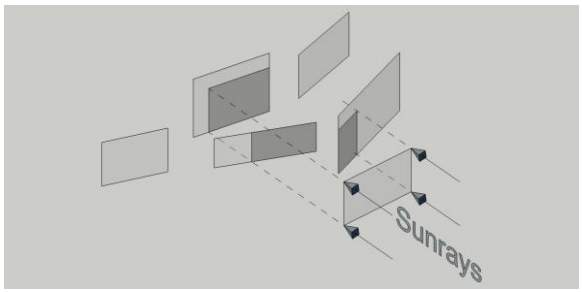

Figure 3 Weiler's and Artherton's algorithm of solving the visibility problem

Weiler and Artherton apply their sorting method only to polygons that are partly or completely covered by the foremost polygon seen from the position of the light. They store these polygons in an "inside list". The inside list has to meet the criterion that none of its polygons must be in front of the polygon that was assumed foremost. If it cannot be met a new inside list has to be set up recursively with the polygon violating the above criterion as the new foremost.

Polygons which are not part of the inside list as well as the parts of the polygons from the inside list that are not covered by the currently foremost polygon are temporarily stored in an "outside list". The outside list will become the new inside list after the original one has been finished processing.

So Weiler and Artherton's algorithm requires that two geometrical problems are solved:

- Determine whether two polygons overlap from the point of view of the light.
- Determine which of two polygons is more to the front.

And both those problems require that

- a 3D scene is projected onto a 2D projection plane (Figure 4) and that
- the 2D projection plane is transformed from the original 3D coordinate system into a 2D coordinate system in order to apply common polygon clipping algorithms.

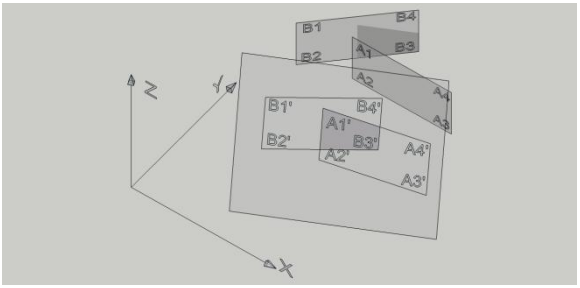

Figure 4 Projection of a 3D scene on a plane orthogonal to the sun. The overlapping part in the projection plane represents the shadow being cast from one polygon onto the other. However, without solving the visibility problem no conclusions can be drawn as to whether polygon A casts a shadow on polygon B or vice versa.

Transformation from 3D into 2D

The sequence in which the two steps are executed is arbitrary. The relevant rotation matrix and projection matrix are not the same in both cases, though. We choose to rotate our coordinate system before and apply the projection afterwards.

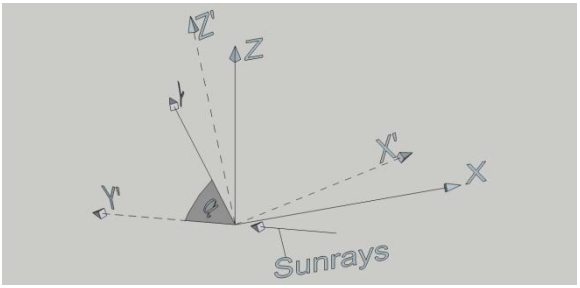

Figure 5 Rotation of the coordinate system to have one axe parallel to the sunrays

The choice of the axis to be aligned with the direction of the light is of no relevance for the transformation into a 2D coordinate system. It can be X as well as Y or Z. In the example illustrated in Figure 5 we chose Y.

The rotation matrix in order to align Y with the direction of the light is set up as

$$\tilde{M}_{rot} = \begin{bmatrix} r_x^2(1-\cos\alpha)+\cos\alpha & r_x r_y(1-\cos\alpha)-r_z\sin\alpha & r_x r_z(1-\cos\alpha)+r_y\sin\alpha \\ r_y r_x(1-\cos\alpha)+r_z\sin\alpha & r_y^2(1-\cos\alpha)+\cos\alpha & r_y r_z(1-\cos\alpha)-r_x\sin\alpha \\ r_z r_x(1-\cos\alpha)-r_y\sin\alpha & r_z r_y(1-\cos\alpha)+r_x\sin\alpha & r_z^2(1-\cos\alpha)+\cos\alpha \end{bmatrix} \quad (3)$$

with $\bar{r}$ as the rotational axis through the origin and α the rotation angle.

$$\bar{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \quad (4)$$

$\bar{r}$ is orthogonal to the Y-axis and the direction of the light $\bar{s}$:

$$\bar{r} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \times \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} \quad (5)$$

and

$$\cos\alpha = s_y \quad (6)$$

The rotation of a single point can be calculated via the matrix multiplication of $\tilde{M}$ with the relevant coordinates:

$$\begin{bmatrix} p_x' \\ p_y' \\ p_z' \end{bmatrix} = \tilde{M} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \quad (7)$$

Calculating the rotation of all *n* vertices with a single matrix multiplication supported by Python's NumPy results to:

$$\begin{bmatrix} p_{1,x}' & p_{2,x}' & \cdots & p_{n,x}' \\ p_{1,y}' & p_{2,y}' & \cdots & p_{n,y}' \\ p_{1,z}' & p_{2,z}' & \cdots & p_{n,z}' \end{bmatrix} = \tilde{M}_{rot} \begin{bmatrix} p_{1,x} & p_{2,x} & \cdots & p_{n,x} \\ p_{1,y} & p_{2,y} & \cdots & p_{n,y} \\ p_{1,z} & p_{2,z} & \cdots & p_{n,z} \end{bmatrix} \quad (8)$$

The new y'-coordinate of a vertex represents its distance from the position of the light reduced by the distance of the origin from the position of the light. Since the second term is constant it can be ignored in the following considerations.

The plane receiving a parallel projection is represented by the equation:

$$p_x + p_y + p_z = 0 \quad (9)$$

The direction of the projection can be represented by the vector:

$$\bar{d} = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \quad (10)$$

Thus the matrix for the parallel projection results to:

$$\tilde{M}_{proj} = \begin{bmatrix} (d_y p_y + d_z p_z)/c & -d_x p_y/c & -d_x p_z/c \\ -d_y p_x/c & (d_x p_x + d_z p_z)/c & -d_y p_z/c \\ -d_z p_x/c & -d_z p_y/c & (d_x p_x + d_y p_y)/c \end{bmatrix} \quad (11)$$

where

$$c = d_x p_x + d_y p_y + d_z p_z \quad (12)$$

If the direction of projection is parallel to the Y'-axis:

$$\bar{d} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad (13)$$

Then the projection matrix obviously defaults to:

$$\tilde{M}_{proj} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (14)$$

So the y'-values of all vertices can be omitted in order to apply a 2D polygon clipping library and to calculate the overlap of the projections of the individual polygons.

The order in which the polygons are overlapped is given by the inside list mentioned above. Nevertheless the order of the inside list has to be checked because there is no guarantee that the original order of the inside list is correct. (Figure 6)

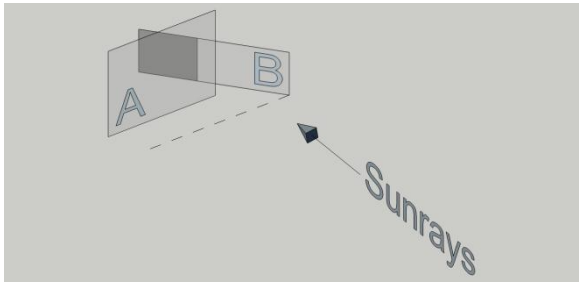

Figure 6 Scenario causing an error in the depth sort based on the lowest y' – coordinate of each polygon. The inside list needs to be re-created recursively.

The check can be made at any point within the overlapping part of the two polygons in the projection plane and consists of comparing the relevant y' – values of the two polygons. If the original order turns out to be erroneous then a new inside list has to be created recursively.

Calculation of shadowed parts

The shadowed part of a particular polygon can easily be calculated from the area of all clipped parts of the projections resulting from the other polygons further at front:

$$A_D \cos\theta = \sum A_{ClipProjPoly} \quad (15)$$

The total area can be obtained from the original projection of a particular polygon.

$$A_T \cos\theta = A_{ProjPoly} \quad (16)$$

The polygon clipping library that was used to figure out the clipped parts of a polygon's projection was Murta's general polygon clipper library (GPC). It is based on Vatti's generic solution to polygon clipping (1992) and can also process convex polygons and polygons with holes which is needed to take windows into account.

Rädler provides a python binding for the GPC and binds clipping operations to the standard operators like +, -, & etc.

OPENGL-BASED ALTERNATIVE

An alternative way of calculating shadows and the PSSF respectively is based on OpenGL. (Jones, Greenberg and Pratt, 2011) OpenGL is intended for accessing the graphics processing unit (GPU) of a computer in order to manipulate the picture sent from the computer to the screen and to make use of the parallel computing potential of the GPU as efficiently and as directly as possible. The idea of exploiting GPUs for scientific calculations that can be parallelized has come up only during the recent years. However, the GPU-centric design of OpenGL as well as its original purpose makes it often difficult to port scientific implementations to OpenGL.

Although shadow calculation appears to be an explicitly graphic problem also seemingly peripheral tasks such as figuring out the coordinates of a shadow polygon from the signal of the GPU or counting pixels representing the shadowed part of a model can become troublesome.

Another problem with OpenGL is related to the diversity of GPUs and to the unreliable compatibility of GPU drivers with current programming specifications. Where some manufactures comply well with agreed standards others don't. The OpenGL wiki states that a recent driver is needed for the compatibility with modern OpenGL versions and recommends to also try out beta versions of drivers in order to get access to bugfixes.

Therefore problems related to the driver for the GPU which is often needed in a particular version may occur as well as problems with the installation of the programming framework needed to develop OpenGL applications. Prior to the attempt to implement a complete shadow mapping calculation with OpenGL the installation of the OpenGL utility toolkit (GLUT) had to be completed. The installation process was also troubled by inconsistencies of particular versions which produced error messages of little value.

Baker (2009) runs a website on which he not only explains the theory of a GPU-based implementation of shadow mapping but also gives a concrete example in C++ programming language. His example was modified to create a scene representing the facades of a virtual assembly of buildings.

Drawing objects in OpenGL can be done with the OpenGL utility toolkit (GLUT). GLUT disposes of a few functions to draw simple geometric forms, such as triangles, quads or polygons. These functions can be used to create also complex structures such as a building envelope.

Rendering a scene with shadows requires the relevant transformation matrices in OpenGL. A transformation matrix can be obtained via the *glGetFloat* command after the desired operations have been applied to the matrix currently on top of the OpenGL matrix stack. The object that is intended to hold the relevant matrix is passed as an argument. There are two types of matrices which are needed to implement shadow mapping in OpenGL:

- Matrices defining a perspective projection from the position of the light and from the position of the camera respectively.
- Matrices defining a view transformation to render the scene from a particular point, in this case the position of the light and the position of the camera respectively.

OpenGL-based shadow mapping builds upon Williams' approach (1978): A scene needs to be rendered three times in order to implement shadow mapping in OpenGL. In the first pass the scene is drawn from the position of the light. This is done by applying the light-projection matrix in the GL_PROJECTION mode of OpenGL and by applying the light-view matrix in the GL_MODELVIEW mode. The depth values of all objects within the current field of view are stored in a shadow map texture which will be used in a successive step to identify the lit-up parts of the scene.

In the second pass the whole scene is rendered in dim light from the position of the camera by applying the camera-projection matrix and the camera-view matrix and setting the required properties to set up dim light.

The third pass is where the depth map comparison is set up. Therefore the depth values from the shadow map texture of the first pass are compared with the depth values of a newly generated texture representing the objects of the scene in sunlight. OpenGL provides a function (*glAlphaFunc*) that can be used to compare the depth values of two textures and to discard all fragments of the texture disposing of a higher depth value than the shadow map texture. This way only those parts of the newly generated texture are rendered which are in direct light.

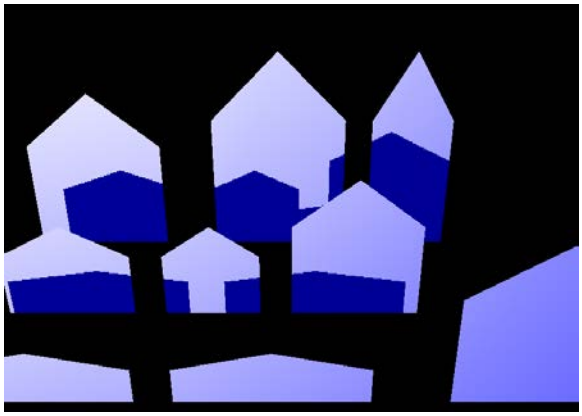

Figure 7 Image of correct OpenGL-based calculation of shadow casting

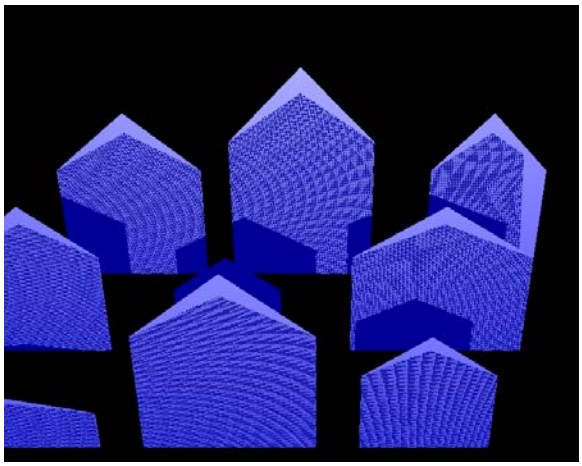

Figure 8 Image of OpenGL-based calculation of shadow casting with unwanted artifacts

The attempt to develop an OpenGL based shadow mapping implementation was partly successful (Figure 7) but partly the results rendered had notable artifacts (Figure 8) which might be due to the fact that the quality of the results of different GPUs may vary, depending on the manufacturer and the model. The GPU used for the attempts at hand was a rather simple onboard one.

Rendering a scene with shadows is usually not enough for determining the PSSF for each surface of interest. Therefore the number of pixels has to be determined. Jones et al. (2011) present a method based on Yezioro's and Shaviv's approach (1994) where they render a particular scene or a portion thereof to two image buffers, once with shading devices and once without. Then the number of pixels held in the two image buffers is tallied, and the fraction of each surface which is shadowed can be calculated. Bartz et al. present a method of OpenGL-assisted occlusion culling for large polygonal models where they use a stencil buffer to run an occlusion test.

Since 2003 OpenGL provides an extension, the ARB_occlusion_query, whose functionality is demonstrated by Harris (2005). Occlusion queries can be used to track the number of fragments or samples that pass the depth test. For this purpose all relevant objects have to be wrapped into an occlusion query by calling *glBeginQueryARB* and *glEndQueryARB* respectively. The command *glGetQueryObjectuivARB* writes the number of pixels of the relevant object or texture into a variable which is passed as an argument. A pixel count of the texture representing the parts that are in direct light and of the whole scene can be used to determine the PSSF.

EVALUATION

The polygon-based implementation

The implementations were tested with a notebook with an Intel P6100 CPU with 2.0 GHz. The polygon-based implementation was applied to an

arbitrary number of randomly generated polygons regarding their size and orientation. The polygons were placed in rows of five with the rows at a randomly generated distance from each other. Intersections between polygons were avoided. The light source was assumed horizontal in the direction of where the rows of polygons were aligned. Each polygon had four vertices and three windows of different size.

Table 1 Calculation time for scenarios with an arbitrary number of randomly generated polygons

NUMBER OF POLYGONS	NUMBER OF VERTICES	CALCULATION TIME IN SECONDS
200	800	0.1
400	1,600	0.3
800	6,400	1.1
1,600	12,800	4.2
3,200	25,600	15.9
6,400	51,200	67.5
9,600	76,800	151.6
12,800	102,400	264.7

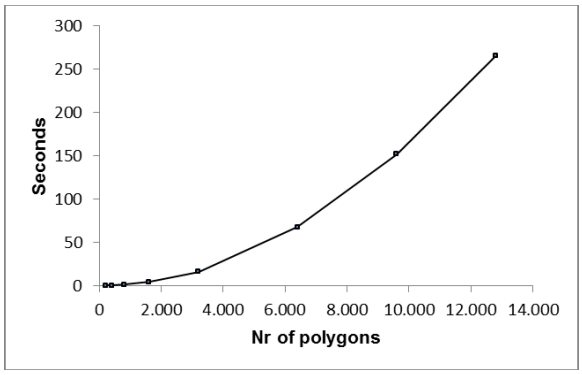

Figure 9 Performance of GPC-based implementation. Processing time rises in a ratio of the power of two of the number of input polygons

Comparing OpenGL with polygon clipping

The OpenGL-based implementation as well as the polygon-based one has been tested with a scenario where the shadow that a treetop casts on a flat wall is calculated. The leaves of the treetop were modeled as triangles whose edge lengths were reduced between the individual simulation runs in order to raise the number of leaves.

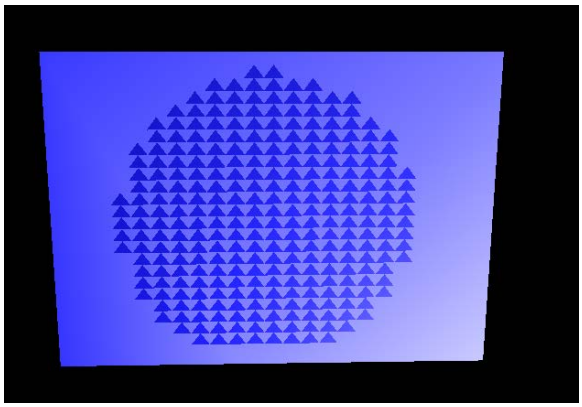

Figure 10 Image of OpenGL-based simulation of a treetop casting a shadow on a flat wall.

With the OpenGL implementation the number of leaves was set to about 500 at the first simulation run. It was doubled in both directions after each run ending up at about 500.000 leaves after five runs. The shadow map buffer was set to a size of 768^2. There was practically no delay between the start of the simulation run and the rendering of the scene to the screen. However already at the third simulation run (8.000 leaves) it was not possible to make out a meaningful result with the naked eye. With the chosen shadow map buffer about 74 pixel can be allocated per leaf presumed the leaves do not cover each other. So no exact result can be expected if the number of details comes close to the size of the shadow map buffer.

Jones et al. (2011) propose to set the viewport to the part (e.g. a surface) of a relevant scenario that the PSSF shall be calculated for. Thus the accuracy of the result can be augmented.

Whereas in general the exactness of the result suffers and the performance is excellent with OpenGL for scenarios containing a high number of details the opposite is the case with implementations based on polygon clipping. The scenario from Figure 10 was also calculated with the polygon-based implementation. A simulation run with 2.500 leaves took about 45 seconds, one with 10.000 leaves about 550 seconds. The difference from the figures in Table 1, illustrated in Figure 9 is due to differences of the geometric circumstances of the test scenarios. In the first test scenario some polygons were covering others whereas with the treetop simulation all leaves were considered to be at the same depth. The result for the treetop simulation included not only the exact PSSF for the wall in the background but theoretically also the PSSF for every single leaf.

Accuracy

The accuracy of the results obtained by an OpenGL calculation depends on the size of the shadow map buffer and the geometry of the objects within the relevant viewport. Jones et al. (2011) show that the error for the calculation of a single surface is usually negligible. However, depending on the geometry of

the scene and the position of the viewport, more than one surface needs to be taken into account in a single calculation run. This is when the calculation error may rise depending on the number of obstructing objects in the scene.

Several runs with different viewports in order to cover the whole scene may be required to obtain good accuracy.

The error of a polygon-based calculation depends on the accuracy of the intersections of the projections of the individual edges of two surfaces which is usually also negligible.

CONCLUSION

The calculation of shadowed areas is indispensable for a running comprehensive building simulation. There are substantially two different types of calculations: Polygon-based ones and GPU-based ones.

The paper at hand demonstrates how a complete polygon-based implementation can be set up, starting with a list of coordinates representing the envelopes of buildings and ending with the calculation of the PSSF. Shadow calculation is theoretically often reduced to clipping polygons. Many publications ignore that the prerequisites for applying a polygon clipping algorithm need to be fulfilled before. These prerequisites comprise the projection of 3D objects onto a 2D projection plane as well as solving the visibility problem. Transformations of 3D forms into a 2D coordinate system require the relevant matrices to be set up and the appropriate programming tools or languages to perform the associated mathematical operations. A matrix-based transformation from 3D into 2D implies a significant performance gain compared to vector-based calculations of intersections for all points.

Python is one well known programming language written in C that disposes of sophisticated libraries such as NumPy supporting these operations. Also some polygon clipping libraries, such as the GPC, dispose of a python binding which eases a homogenous implementation regarding the programming language. Weiler and Artherton do not only present a polygon clipping algorithm in their paper (1977) but also a way to solve the hidden surface problem. Whereas the hidden surface removal algorithm is not critical regarding the overall performance of the implementation a polygon clipping library can be. Jörg Rädler's binding of the GPC was used in the implementation at hand. The GPC itself is based on Vatti's solution to polygon clipping. It supports concave polygons (which is not the case with Weiler's and Artherton's algorithm) and holes which is needed in order to take windows into account.

GPU-based approaches of shadow calculation are usually based on shadow mapping. The installation and programming effort is bigger compared with the polygon-clipping-approach. However its performance can well be used in real time calculations and with complicated structures because the approach makes use of the available GPU the best possible way. Since GPU-based implementations return a result per pixel the accuracy of the calculation depends on the size of the chosen viewport. Jones et al. (2011) have shown that with a viewport set in a sensible way the solution is accurate enough for most requirements.

With complex scenarios the full advantage of GPU driven calculations can be used most efficiently.

The effort of setting up the programming environment as well as preparations for rendering the scene such as building up the model in OpenGL and setting the right field of view, etc. is substantial.

Thus based on the current state of programming technology and scientific knowledge the authors of the paper at hand favor the polygon-based approach to calculate shadowing.

Acknowledgements

Research funded by the Zentrum für Innovation und Technologie GmbH (ZIT) and the Federal Ministry for Transport, Innovation and Technology.

The authors would also like to thank Nathaniel Jones for providing some publications relevant for the this paper.

REFERENCES

Artherton P., Weiler K., Greenberg D., 1978. Polygon shadow generation. Proceedings of the 5th annual conference on Computer graphics and interactive techniques, p.275-281, August 23-25.

Baker P., 2009. Shadow Mapping [online]. Available from http://www.paulsprojects.net/tutorials/smt/smt.html [Accessed 4 November 2012]

Bartz D., Meißner M., Hüttner T., 1999. OpenGL-assisted occlusion culling for large polygonal models, Computer & Graphics, vol. 23, no. 5, pp. 667-679

Grau, K. and Johnsen, K., 1995. General shading model for solar building design. ASHRAE Transactions, 101 (2), 1298–1310.

Greiner G., Hormann K., 1998 Efficient clipping of arbitrary polygons, ACM Transactions on Graphics, 17 (2), pp. 71–83

Harris K., 2005. Occlusion Query [online]. Available from http://www.codesampler.com/oglsrc/oglsrc_7.htm [Accessed 4 November 2012]

Jones N. L., Greenberg D. P. and Pratt K. B., 2011. Fast computer graphics techniques for calculating direct solar radiation on complex building surfaces. Journal of Building

Performance Simulation Vol. 5, No. 5, September 2012, 300–312.

Khronos Group, 2010. OpenGL, Open Graphics Library [online]. Available from http://www.opengl.org/ [Accessed 4 November 2012]

Khronos Group, 2010. GLUT, The OpenGL Utility Toolkit [online] http://www.opengl.org/resources/libraries/glut/ [Accessed 4 November 2012]

Khronos Group, 2010. ARB_occlusion_query [online] http://www.opengl.org/registry/specs/ARB/occlusion_query.txt [Accessed 4 November 2012]

Murta, A., 2009. General polygon clipper library (GPC) [online]. Available from: http://www.cs.man.ac.uk/~toby/gpc/ [Accessed 4 November 2012].

Newell M. E. , Newell R. G., Sancha T. L., 1972. A solution to the hidden surface problem. Proceedings of the ACM annual conference, August 01-01, 1972, Boston, Massachusetts, United States

NumPy, package for scientific computing with Python [online] http://numpy.scipy.org/ [Accessed 4 November 2012]

Python Programming Language – Official Website, http://www.python.org/

Rädler J., Polygon [online]. Available from http://www.j-raedler.de/projects/polygon/ [Accessed 4 November 2012]

Vatti, B.R., 1992. A generic solution to polygon clipping. Communications of the ACM, 35 (7), 56–63.

Walton, G.N., 1979. The application of homogeneous coordinates to shadowing calculations. ASHRAE Transactions, 84 (1), 174–180

Weiler K., Artherton P., 1977. Hidden surface removal using polygon area sorting, ACM SIGGRAPH Computer Graphics, 11 (2), 214–222

Williams L., 1978. Casting curved shadows on curved surfaces, Proceedings of the 5th annual conference on Computer graphics and interactive techniques, p.270-274, August 23-25, 1978

Yezioro, A. and Shaviv, E., 1994. Shading: a design tool for analyzing mutual shading between buildings. Solar Energy, 52 (1), 27–37