Geometric Modeling and Optimization in 3D Solar Cells: Implementation and Algorithms

by

Jin Hao Wan

Double B.S. in Math and Computer Science, M.I.T., 2012

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014
[June 2014]

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
February 26, 2014

Signature redacted

Certified by .....
Jeffrey C. Grossman
Associate Professor
Thesis Supervisor

Signature redacted

Accepted by .....
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Geometric Modeling and Optimization in 3D Solar Cells: Implementation and Algorithms

by

Jin Hao Wan

Submitted to the Department of Electrical Engineering and Computer Science
on February 26, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

Conversion of solar energy in three-dimensional (3D) devices has been essentially untapped. In this thesis, I design and implement a C++ program that models and optimizes a 3D solar cell ensemble embedded in a given landscape. The goal is to find the optimum arrangement of these solar cells with respect to the landscape buildings so as to maximize the total energy collected. On the modeling side, in order to calculate the energies generated from both direct and reflected sunlight, I store all the geometric inputs in a binary space partition tree; this data structure in turn efficiently supports a crucial polygon clipping algorithm. On the optimization side, I deploy simulated annealing (SA). Both advantages and limitation of SA lead me to restrict the solar cell docking sites to orthogonal grids imposed on the building surfaces. The resulting program is an elegant trade-off between accuracy and efficiency.

Thesis Supervisor: Jeffrey C. Grossman
Title: Associate Professor

Acknowledgments

I would like to thank my supervisor, Prof. Grossman, for introducing me to 3D solar cells and for agreeing to supervise this thesis when I was in need. He encouraged me to take ownership of my project and to take roles in learning. His innovative and multi-disciplinary approach to research is inspiring. There has been tough times while working on this thesis. I would like to thank him for always being supportive.

I would like to thank my original collaborator, Dr. Marco Bernardi, for helping me get started on this project and for providing constant feedback on my progress. His warm personality inside and outside of lab always uplifted me. I would like to thank my current collaborators, Dr. Nicola Ferralis and Anna Cheimets, for bearing with my rough schedule last semester and for providing practical perspectives on the direction of this project.

I would like to thank my parents for always being there for me and distilling a can-do attitude in me. I would like to thank all my friends who talked to me during the ups and downs of this project; in particular, I would like to thank Taylor Han for taking time out of his busy schedule to deliver this thesis for me.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, we provide the background on 3DPV. We discuss the original motivation for this new paradigm along with recent development in this direction. Finally, we go over the system pipeline of our C++ program, which naturally leads to an organization chart for the rest of this thesis.

1.1 Motivation and Literature Review

Converting the abundant flow of solar power to the Earth ($87PW$) into affordable electricity is an enormous challenge, limited only by human ingenuity. Photovoltaic (PV) conversion has emerged as a rapidly expanding technology capable of reaching GW-scale electric power generation [2] with the highest energy densities among renewable sources of $2040W/m$. Deployment of Solar PV is therefore essential for the reduction of greenhouse gas (GHG) emissions, at the global scale as well as at state level. Currently, photovoltaic (PV) solar power generation is achieved using a flat panel form factor (2DPV). The biggest advantage of 2DPV consists in straightforward installation on rooftops, leaving no shadow on neighboring cells, has numerous manufacturing advantages, and is materials efficient (e.g., the active layer requires simple materials deposition on flat substrates). The main approach applied so far to alleviate these problems has been the search for cheaper active layers with higher power conversion efficiency [3].

Despite the efficiency improvements, the main barriers to widespread adoption of 2DPV technology include system costs of the order of $35/W$, of which 60% is due to several factors, such as installation costs, and the requirement of a minimum threshold power density for cheaper thin-film technologies to become feasible for residential and commercial rooftop installations. From a technical perspective, additional limitations for widespread PV adoption are the reduced "real life" energy collection due to adverse weather conditions, and the limited number of peak insolation hours available, especially in high-latitude states like Alberta. Since the latter is related to the coupling of the solar cells to the Suns trajectory, it is most sensitive to changes in the PV system design, the typical one consisting of flat panels arranged on a flat surface commonly a rooftop imposing further geometrical constraints that yields far-from-optimal coupling with the Suns trajectory. Sun-tracking systems can extend the range of useful peak hours, but are not well suited for domestic installations as they use expensive and bulky movable parts. Such limitations (weather and non-homogenous power generation, mostly at off-peak hours) imposed by the use of conventional 2DPV design can severely limit the theoretical benefits brought by solar PV to the reduction of GHG emissions. In fact, under these conditions, the reliance on conventional GHG emitting energy sources remains significant. The technology presented in this thesis (three dimensional photovoltaics 3DPV) aims at providing a more stable and homogeneous power generation, less reliant on conventional power generation sources under non-optimal insolation conditions, therefore providing concrete benefits in the reduction of GHG emissions.

Limiting solar PV designs to two dimensions has made sense since it is the most efficient use of photoactive material per square foot of land. Given the historically dominant cost of the active layer material in PV [2], this single cost metric has been important enough to render anything other than flat architectures far too impractical. And yet, times are changing. The silicon learning curve alone (Figure 1-1) shows the dramatic reduction in active layer materials cost compared to balance of systems (BOS), by a factor of > 20 compared to several decades ago. Beyond silicon, many new materials such as those based on plastics, quantum dots, or small molecules,

show promise for even more substantial lowering of the cost of the active materials due to low-temperature processing and solution-based synthesis [4] [5].

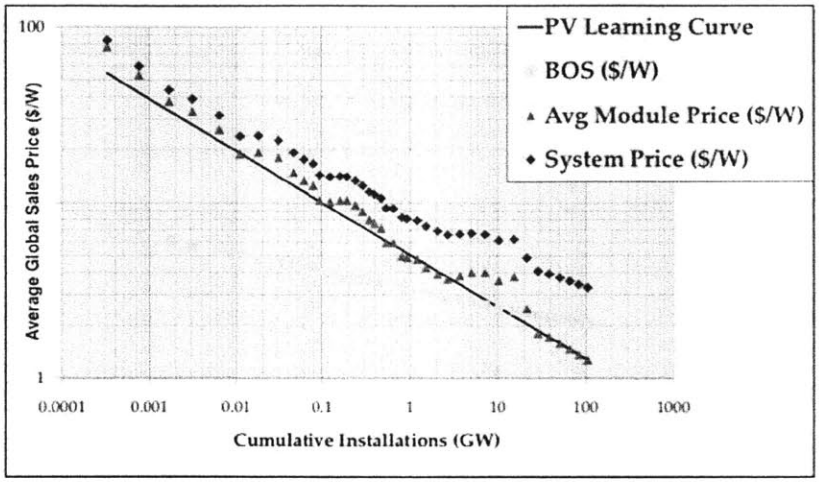

Figure 1-1: Cost and efficiency benefits of 3DPV are increasing as the average module cost is rapidly decreasing compared to BOS costs.

While the development of novel technologies for 2DPV devices is extensive, the potential for scientific and technological investigations of the collection and conversion of solar energy in three-dimensional (3D) devices has been essentially untapped. Yet, it is our firm belief that the impact of 3DPV technology and its innovative potential is enormous, especially in high latitude, climate adverse locations like Alberta. In particular, our initial investigations [6] [7] show 3DPV technology is particularly suited for power generation in adverse weather conditions (with a reduced loss due to clouding) and for a more uniform power generation across the day and at off-peak hours at high latitudes. The innovative and potentially game-changing nature of this technology can be applied at multiple dimensional scales, from small 3DPV towers acting as compact foldable power generators, to optimized integration of 3DPV towers in urban settings. Ultimately, however the major innovation paradigm for this technology will be in the integration of 3DPV with architectural and urban design for improved efficiency in energy generation at grid scale. For example, why should PV systems be deployed only as rooftop installations? Buildings and urban settings

provides a natural 3D framework for which the benefits of 3DPV technology could be maximized through the use of a solar grid. Just as a unified sewage or electrical grid provide clear benefits at community and city levels, a solar grid (where individual 3DPV installations are optimized to work in concert through the day towards the maximization of the power production at a city level) could provide significant advantages in terms of increased and more homogeneous power generation (at the city level), as well as optimization of backup power storage (Figure 1-2). Ultimately this would result in a net reduction of GHG emissions by the reduced reliance on backup conventional power generation sources. Therefore, the tremendous potential for the exploitation of 3DPV at different scales can be achieved only through extensive fundamental and technological, computational and experimental investigations of 3DPV technology. For example, questions regarding the role of light reflection, incident angle, and position with respect to the sun, panel arrangements with respect to one another, and a number of other factors define only a part of a complicated optimization problem.

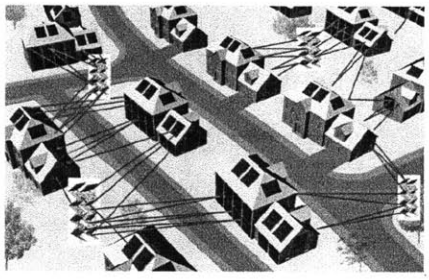

Figure 1-2: The "solar grid": To what extent is it possible to optimize solar energy collection on a city level, by integrating 3DPV systems to maximize the collection of indirect, reflected and diffuse insolation?

1.2 Problem Statement

Conventional rooftop installations are essentially pure 2DPV systems. In considering 3DPV designs, a building where solar PV panels and mirrors are strategically installed not just on the roof, but also around its surface can be considered as a

3DPV system. The energy generation follows the same trends as any 3DPV system, namely, an increased and homogeneous power generation across the day and increased collection of diffused and reflected light. The benefits of individual 3DPV systems that are isolated from the surrounding environment are significant. However, the role of reflected, diffuse and indirect illumination in energy generation in 3DPV systems (which consists of up to 30% of the total energy generated) suggests that even more significant improvements can be potentially achieved by optimizing, by design, how the light can be reflected and collected among several 3DPV systems.

From an urban and architectural standpoint, such approach consists in optimizing how PV modules and mirrors can be installed using 3DPV design schemes on building or urban infrastructures (electric posts, traffic lights, communication towers) to maximize the collection of indirect and reflected light from and to each of these 3DPV systems. Ultimately, the goal is for each building to be interconnected with each other from a solar collection point of view (the solar grid).

The question of how much such interaction between 3DPV systems through reflection of light can be optimized, as an additional factor in power generation, remains open. As the central part of this proposal, we intend to apply the simulation and optimization tools to particular 3DPV systems embedded in an urban environment to explore this potential.

1.3 Program System Pipeline

The entire 3DPV program is divided into five modules: 1. SketchUp Reader/Writer, 2. Solar Tracker, 3. Input Processor, 4. Solar Calculator and 5. Annealing Simulator. An input file goes through these modules sequentially from 1 to 5. In particular, the Solar Calculator is the most complicated and computational involved one; it can be further divided into three procedural parts: a. binary space partition tree generation, b. polygon clipping and c. straightforward application of formulas from optics. The rest of the thesis will follow this division and sub-division to delve into the algorithms and implementation of the 3DPV program.

Chapter 2

SketchUp Reader and Writer

SketchUp is a third party 3D modeling program. It facilitates fast architectural design. A user can easily design a landscape input file in SketchUp and then port it to the 3DPV program. In this chapter, we briefly go over features of SketchUp that are relevant to the modeling part of 3DPV and how the SketchUp C API is being used to port the input file.

2.1 Useful Features of SketchUp

Every 3D object in SketchUp is defined by its set of enclosing surfaces. While SketchUp does support curved and non-polygonal (e.g. circular) surfaces, we restrict our inputs to 3D objects made up of only flat and polygonal surfaces. This means each 3D object in our input can be specified by a list of 2D polygons. This representation is convenient and not far removed from the reality.

Moreover, once a 2D polygon becomes part of a 3D object in SketchUp, its vertices will be arranged in the counter-clockwise order with respect to the polygon's *outward-pointing* normal. This implies that we can use the orientation of a polygon's vertices to infer which side of the polygon is facing outward. This comes in handy when, for example, the program tries to mount a panel on a wall: It should never mount it on the inward side of the wall!

Lastly, SketchUp allows the user to add colors and textures to each surface. We

take advantage of this feature to use colors to encode material properties of the underlying surfaces.

2.2 Reading From and Writing to SketchUp Files

Since all the necessary information, coordinates of vertices and material properties, is stored in the polygonal faces, it suffices if we can access the faces of each object in a SketchUp file. Conveniently, SketchUp provides a C library through which we can write a short program to access an object's faces. Conversely, we can output the results of 3DPV, in terms of vertex coordinates, colors and textures, to SketchUp via the same library. This provides an easy way to visualize the quality of our models and optimized results.

This porting approach, however, is not cross-platform. The C library for Mac is currently broken and SketchUp does not plan to fix and maintain it in the long run. For now, we first read in the SketchUp file on a Windows machine and then carry out the modeling and optimization steps on a Mac. An cross-platform alternative to this is writing a SketchUp plugin in Ruby.

Chapter 3

Solar Position Tracker

We use a third party C++ solar position tracker written by Afshin Michael Andreas. The underlying solar tracking algorithm was developed by the National Renewable Energy Lab (NREL). There only two relevant key features about this tracker:

1. it divides the day time into equal intervals;
2. for each interval, we can query the tracker to get the current average position of the sun.

Later in the energy calculation step, we will need the solar position for each time interval to derive the direction of the sun light.

Chapter 4

Input Pre-Processor: Generating Orthogonal Grids

Knowing that we are going to optimize the placement of the solar cells by simulated annealing, we discretize the search space by restricting the docking sites of the solar cells to predefined grids. These grids are naturally imposed on top of each polygonal surface in the input. We will assume that each polygon is convex.

4.1 Locating the Pivot

We first locate the left end point of the *lowest edge* in the polygon. Here, the lowest edge stands for the edge whose end points have the smallest z-coordinate among all vertices. We can find the lowest edge by a linear exhaustive search. Once the left end point of this edge is identified, we take the two adjacent vertices of the end point. These three points together form the *pivot*.

The reason these three points, which make up an angle, is called pivot is that we are going to transform the polygon so that the lower side of the pivot lies exactly along the positive x-axis, starting at the origin. This transformation will reduce our 3D problem to the same problem on 2D – on the $x - y$ plane. This is what we call the *canonical framework*.

4.2 Transformation onto the Canonical Framework

The transformation onto the canonical framework involves three steps. First we translate the polygon so that the middle point of the pivot is moved to the original. Second. we rotate the resulting polygon so that its normal aligns with the positive z-axis. This is a straightforward application of Rodrigues's rotation formula. Lastly, we need to make sure that the lower side of pivot is along the positive x-axis. This amounts to another rotation.

4.3 Generating Orthogonal Grids in the Canonical Framework

We will use the middle point of the pivot as the *anchor* point. First we repeatedly generate grids from left to right, one after another. Once we've reach the outside of the polygon, we move the current anchor point up by one unit and keep going from left to right. This is fairly easy to implement if we can efficiently decide when a point (e.g., a corner of a current grid) is inside the polygon. As an illustration, the following is the set of orthogonal grids generated for two identical houses by this algorithm:

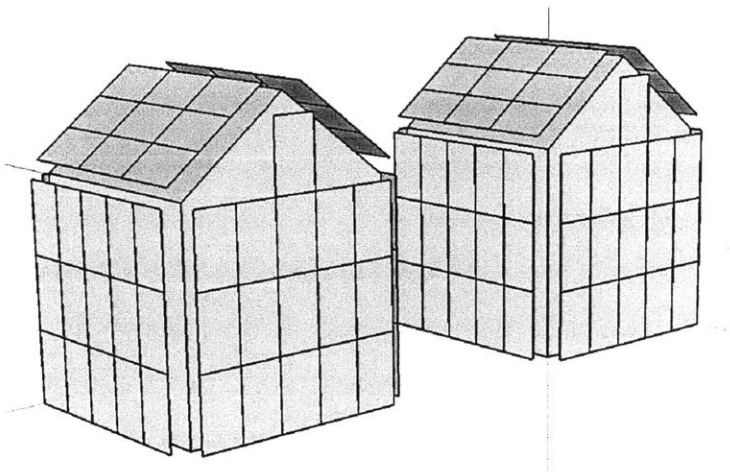

Figure 4-1: Orthogonal grids on two houses.

4.4 The Point in Polygon Problem

It turns out that the point in polygon (PIP) problem is well-known in computation geometry and has a very clever solution. The following three-step algorithm does the trick:

1. Draw a horizontal line to the right of each point and extend it to infinity.
2. Count the number of times the line intersects with polygon edges.
3. A point is inside the polygon if either count of intersections is odd of point lies on an edge of polygon. If none of the conditions is true, then point lies outside.

The correctness of this algorithm is a direct result of Jordan's Curve Theorem.

4.5 Convex Partitioning of Simple Polygons

What if our polygon is concave. In that case the algorithm described above will not work. Although this is not implemented in the current 3DPV code yet, an easy fix is to convex partition the concave polygon first. This partitioning problem is well-known and there are efficient algorithms which can keep the number of resulting convex polygonal fragments as small as possible. After the convex partition, we can apply the above algorithm on each convex fragments to generate separate grids.

Chapter 5

Solar Calculator Part I: Binary Space Partition Tree

In this chapter, we describe binary space partition tree, or BSP tree, the most important and involved data structure used in the solar calculator program. We show that BSP tree is integral in pre-processing a set of polygons in 3-space so that multiple visibility queries can be answered efficiently. These visibility queries are directly related to the hidden surface removal problem and will be invoked in the polygon clipping step, the focus of next chapter. Lastly, we will present a randomized algorithm for generating small-sized BSP tree. The performance guarantee of this algorithm is derived in Appendix B.

5.1 The Hidden Surface Removal Problem

Once the program selects a subset of the grids to be the docking sites for the panels, we have a fixed configuration of surfaces. As will be explained fully in the next chapter, the problem of calculating the total amount of energy generated from first and second absorptions can be modeled as a series of polygon clipping steps, where in each step, we either take union or intersection of a set of polygons. These steps could be greatly simplified if the program knows which polygons or which parts of each polygons are visible from the viewing perspective. Such queries are commonly

known as *visibility queries* in computer graphics world and they are mirror version of the *hidden surface removal problem.*

Hidden surface removal has many applications in visualization, one of which is in flight simulator. Flight simulators must perform many different tasks to make the pilot's experience as realistic as possible. An important task is $3D$ rendering: pilots must be able to see the landscape above which they are flying, or the runway on which they are landing. This involves both modeling landscapes and rendering the models. To render a scene we must determine for each pixel on the screen the object that is visible at that pixel; this is where hidden surface removal comes in. We must also perform shading calculations, that is, we must compute the intensity of the light that the visible object emits in the direction of the view point. The latter task is very time-consuming if highly realistic images are desired. In flight simulators rendering must be performed in real-time, so there is no time for accurate shading calculations. Therefore a fast and simple shading technique is employed and hidden surface removal becomes an important factor in the rendering time.

The *z-buffer algorithm* is a very simple method for hidden surface removal. This method works as follows. First, the scene is transformed such that the viewing perspective is the positive z-direction. Then the objects in the scene are *scan-converted* in arbitrary order. Scan-converting an object amounts to determining which pixels it covers in the projection; these are the pixels where the object is potentially visible. The algorithm maintains information about the already processed objects in two buffers: a *frame buffer* and a *z-buffer.* The frame buffer stores for each pixel the intensity of the currently visible object, that is, the object that is visible among those already processed. The z-buffer stores for each pixel the z-coordinate of the currently visible object. (More precisely, it stores the z-coordinate of the point on the object that is visible at the pixel.) Now suppose that we select a pixel when scan-converting an object. If the z-coordinate of the object at that pixel is smaller than the z-coordinate stored in the z-buffer, then the new object lies in front of the currently visible object. So we write the intensity of the new object to the frame buffer, and its z-coordinate to the z-buffer. If the z-coordinate of the object at that

pixel is larger than the z-coordinate stored in the z-buffer, then the new object is not visible, and the frame buffer and z-buffer remain unchanged.

5.2 The Painter's Algorithm

The z-buffer algorithm is easily implemented in hardware and quite fast in practice. Hence, this is the most popular hidden surface removal method. Nevertheless, the algorithm has some disadvantages: a large amount of extra storage is needed for the z-buffer, and an extra test on the z-coordinate is required for every pixel covered by an object. The *painters algorithm* avoids these extra costs by first sorting the objects according to their distance to the view point. Then the objects are scan-converted in this so-called *depth order*, starting with the object farthest from the view point. When an object is scan-converted we do not need to perform any test on its z-coordinate, we always write its intensity to the frame buffer. Entries in the frame buffer that have been filled before are simply overwritten. This approach is correct because we scan-convert the objects in back-to-front order: for each pixel the last object written to the corresponding entry in the frame buffer will be the one closest to the viewpoint, resulting in a correct view of the scene. The process resembles the way painters work when they put layers of paint on top of each other, hence the name of the algorithm.

To apply this method successfully we must be able to sort the objects quickly. Unfortunately this is not so easy. Even worse, a depth order may not always exist: the in-front-of relation among the objects can contain cycles. When such a *cyclic overlap* occurs, no ordering will produce a correct view of this scene. In this case we must break the cycles by splitting one or more of the objects, and hope that a depth order exists for the pieces that result from the splitting. For example, in Figure 5-1 there is a cycle of three triangles. In this case, we can always split one of them into a triangular piece and a quadrilateral piece, such that a correct displaying order exists for the resulting set of four polygons. Computing which objects to split, where to split them, and then sorting the object fragments is an expensive process. Because the order depends on the position of the view point, we must recompute the order every

time the view point moves. If we want to use the painters algorithm in a real-time environment such as flight simulation, we should pre-process the scene such that a correct displaying order can be found quickly for any view point. An elegant data structure that makes this possible is the *binary space partition tree*, or BSP tree for short.

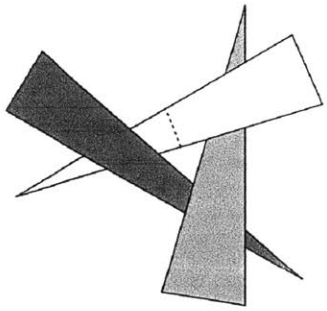

Figure 5-1: Three cyclically overlapping triangles in 3-space that do not admit a depth order.

5.3 BSP Tree in 2D with Auto-Partition

To get a feeling for what a BSP tree is, take a look at Figure 5-2. This figure shows a binary space partition for a set of objects in the plane, together with the tree that corresponds to the BSP. As you can see, the binary space partition is obtained by recursively splitting the plane with a line: first we split the entire plane with l_1, then we split the half-plane above l_1 with l_2 and the half-plane below l_1 with l_3, and soon. The splitting lines not only partition the plane, they may also cut objects into fragments. The splitting continues until there is only one fragment left in the interior of each region. This process is naturally modeled as a binary tree. Each leaf of this tree corresponds to a face of the final subdivision; the object fragment that lies in the face is stored at the leaf. Each internal node corresponds to a splitting line; this line is stored at the node. When there are 1-dimensional objects (line segments) in the scene then objects could be contained in a splitting line; in that case the corresponding internal node stores these objects in a list.

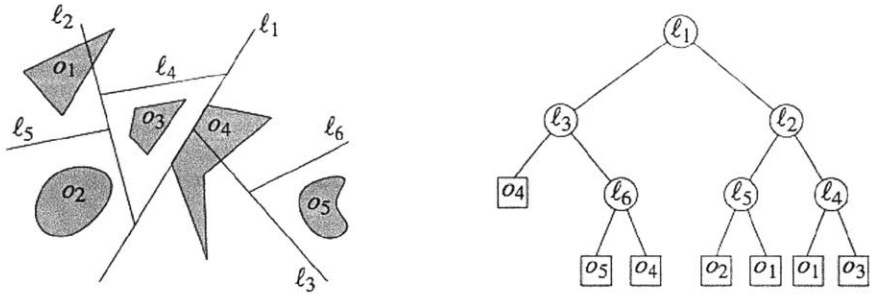

Figure 5-2: The left side shows a binary partion of a set of objects in 2-space; the right side shows the corresponding BSP tree.

The size of a BSP tree is the total size of the sets $S(v)$ over all nodes v of the BSP tree. In other words, the size of a BSP tree is the total number of object fragments that are generated. If the BSP does not contain useless splitting lineslines that split off an empty subspacethen the number of nodes of the tree is at most linear in the size of the BSP tree. Strictly speaking, the size of the BSP tree does not say anything about the amount of storage needed to store it, because it says nothing about the amount of storage needed for a single fragment. Nevertheless, the size of a BSP tree as we defined it is a good measure to compare the quality of different BSP trees for a given set of objects.

The leaves in a BSP tree represent the faces in the subdivision that the BSP induces. More generally, we can identify a convex region with each node v in a BSP tree T: this region is the intersection of the half-spaces h_μ^b , where μ is an ancestor of v and $b = -$ when v is in the left subtree of μ, and $b = +$ when it is in the right subtree. The region corresponding to the root of T is the whole space.

The splitting hyperplanes used in a BSP can be arbitrary. For computational purposes, however, it can be convenient to restrict the set of allowable splitting hyperplanes. A usual restriction is the following. Suppose we want to construct a BSP for a set of line segments in the plane. An obvious set of candidates for the splitting lines is the set of extensions of the input segments. A BSP that only uses such splitting lines is called an *auto-partition*. For a set of planar polygons in 3-space, an auto-partition is a BSP that only uses planes through the input polygons as splitting

planes. It seems that the restriction to auto-partitions is a severe one. But, although auto-partitions cannot always produce minimum-size BSP trees, we shall see that they can produce reasonably small ones. Another great feature of auto-partition is that it naturally resolves the cyclic overlap problem.

5.4 BSP Tree and the Painter's Algorithm

Suppose we have built a BSP tree T on a set S of objects in 3-dimensional space. How can we use T to get the depth order we need to display the set S with the painters algorithm? Let p be the view point and suppose that p lies above the splitting plane stored at the root of T. Then clearly none of the objects below the splitting plane can obscure any of the objects above it. Hence, we can safely display all the objects (more precisely, object fragments) in the subtree T^- before displaying those in T^+. The order for the object fragments in the two subtrees T^+ and T is obtained recursively in the same way. This is summarized in the following algorithm.

Algorithm 1 Painter's Algorithm(T, p)

```
Let v be the root of T
if v is a leaf then
    Scan-convert the object fragment in S(v).
else
    if p ∈ h_v^+ then
        Painter's Algorithm(T^-, p)
        Scan-convert the object fragments in S(v)
        Painter's Algorithm(T^+, p)
    else
        if p ∈ h_v^- then
            Painter's Algorithm(T^+, p)
            Scan-convert the object fragments in S(v)
            Painter's Algorithm(T^-, p)
        else
            Painter's Algorithm(T^+, p)
            Painter's Algorithm(T^-, p)
        end if
    end if
end if
```

Note that the last case in the algorithm corresponds to when p lies on the splitting

plane h_v . In this case, we do not draw the polygons in $S(v)$ because polygons are flat 2-dimensional objects and therefore not visible from points that lie in the plane containing them.

The efficiency of this algorithmindeed, of any algorithm that uses BSP trees depends largely on the size of the BSP tree. So we must choose the splitting planes in such a way that fragmentation of the objects is kept to a minimum. Before we can develop splitting strategies that produce small BSP trees, we must decide on which types of objects we allow. In the 3DPV environment, all objects are $2D$ polygons in the 3-sapce. Moreover, because speed is our main concern, we should keep the type of polygons in the scene simple: we assume that all the polygons have been triangulated (the code for polygon triangulation can be found in Appendix A). So we want to construct a BSP tree of small size for a given set of triangles in 3-dimensional space.

5.5 Constructing a BSP Tree in 3D

When you want to solve a 3-dimensional problem, it is usually not a bad idea to gain some insight by first studying the planar version of the problem. This is also what we do in this section. Let $S = \{s_1, s_2, ..., s_n\}$ be a set of n non-intersecting line segments in the plane. We will restrict our attention to auto-partitions, that is, we only consider lines containing one of the segments in S as candidate splitting lines. The following recursive algorithm for constructing a BSP immediately suggests itself. Let $l(s)$ denote the line that contains a segment s.

Algorithm 2 2D-BSP(S)

if $|S| \leq 1$ **then**
  Create a tree T consisting of a single leaf node, where the set S is stored explicitly.
  return T
else
  Use $l(s_1)$ as the splitting line.
end if

The algorithm clearly constructs a BSP tree for the set S. But it is not necessarily small. Instead of blindly taking the first segment, s_1, we should take the segment

$s \in S$ such that $l(s)$ cuts as few segments as possible. But this is too greedy: there are configurations of segments where this approach does not work well. Furthermore, finding this segment would be time consuming. One solution is simply make a random choice. That is to say, we use a random segment to do the splitting. As we shall see later, the resulting BSP is expected to be fairly small.

To implement this, we put the segments in random order before we start the construction:

Algorithm 3 2D-Random-BSP(S)

Generate a random permutation S' of the set S
$T \leftarrow$ 2D-BSP(S')
return T

We will show in Appendix B that the expected number of fragments generated by the algorithm 2D-Random-BSP is $O(n \log n)$. As a consequence, we can prove that a BSP tree of size $O(n \log n)$ exists for any set of n segments. Furthermore, at least half of all permutations lead to a BSP of size $O(n \log n)$. We can use this to find a BSP of that size: After running 2D-Random-BSP we test the size of the tree, and if it exceeds that bound, we simply start the algorithm again with a fresh random permutation. The expected number of trials in this randomized algorithm is two.

This randomized algorithm for generating a BSP tree easily generalizes to 3-space. Let S be a set of n non-intersecting triangles. Again we restrict ourselves to auto-partitions, that is, we only use partition planes containing a triangle of S. For a triangle t, we denote the plane containing it by $h(t)$.

Algorithm 4 3D-Random-BSP(S)

Generate a random permutation $\{t_1, t_2, ..., t_n\}$ of the set S.
for $i \leftarrow 1$ to n **do**
    Use $h(t_i)$ to split every cell where the split generates new fragments.
    Make all possible free splits.
end for

Chapter 6

Solar Calculator Part II: Polygon Clipping

Now we have generated a BSP tree for all the surfaces in our input, we can leverage this useful data structure to do efficient visibility queries by calling Algorithm 1. As will be explained in this chapter, this allows us to process the surfaces in the right order when doing polygon clipping.

6.1 Problem Definition and Vatti's Algorithm

In this section, we introduce the problem of polygon clipping and briefly go over Vatti's algorithm, the most commonly used algorithm for this problem.

Clipping is defined as the interaction of subject and clip polygons. While clipping usually involves finding the intersections (regions of overlap) of subject and clip polygons, clipping algorithms can also be applied with other boolean clipping operations: difference, where the clipping polygons remove overlapping regions from the subject; union, where clipping returns the regions covered by either subject or clip polygons, and; xor, where clipping returns the regions cover by either subject or clip polygons except where they are covered by both subject and clip polygons.

Quite a few polygon clipping algorithms have been published. Some algorithms, such as the Liang-Barsky and Maillot algorithms, only clip polygons against simple

rectangles. This is not adequate for our application, where surfaces can be non-rectangular. On the other hand, the Sutherland-Hodgman and Cyrus-Beck algorithms are more general and allow clipping against any convex polygon. However, this is still not adequate in 3DPV. In many cases, we allow surfaces in the input to be even concave and even non-simple (e.g., a wall with windows on it). This is why we decide on Vatti's algorithm, which allows clipping against any simple or non-simple polygons.

Call an edge of a polygon a *left* or *right edge* if the interior of the polygon is to the right or left, respectively. Horizontal edges are considered to be both left and right edges. A key fact that is used by Vatti's algorithm is that polygons can be represented via a set of left and right bounds which are connected lists of left and right edges, respectively, that come in pairs. Each of these bounds starts at a local minimum of the polygon and ends at a local maximum. Consider the polygon with vertices $\{p_1, p_2, ..., p_8\}$ shown in Figure 6-1. The two left bounds have vertices p_0, p_8, p_7, p_6 and p_4, p_3, p_2, respectively. The two right bounds have vertices p_0, p_1, p_2 and p_4, p_5, p_6.

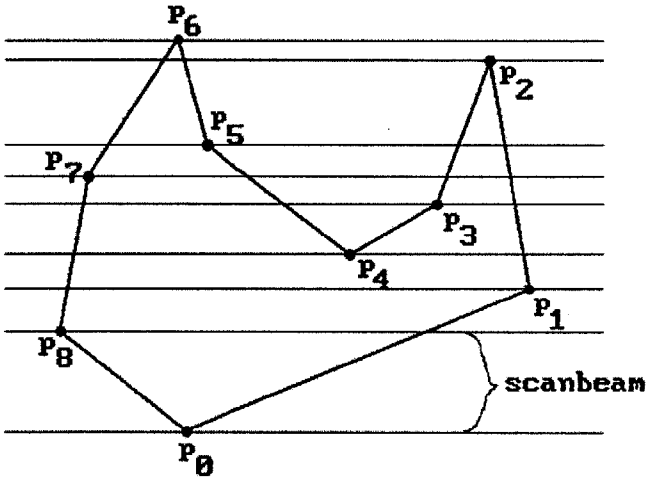

Figure 6-1: Example polygon bounds

Here is an overview of Vatti's algorithm. The first step of the algorithm is to determine the left and right bounds of the clip and subject polygons and to store this information in a *local minima list*, or LML. This list consists of a list of matching

pairs of left-right bounds and is sorted in ascending order by the y-coordinate of the corresponding local minimum. It does not matter if initial horizontal edges are put into a left or right bound. Constructing the LML can be done with a single pass of the clip and subject polygons.

The bounds on the LML were specified to have the property that their edges are either all left edges or all right edges. However, it is convenient to have a more general notion of a left or right bound. Therefore, from now on, a left or right bound will denote any connected sequence of edges only whose first edge is required to be a left or right edge, respectively. We still assume that a bound starts at a local minimum and ends at a local maximum. For example, we shall allow the polygon in Figure 6-1 to be described by one left bound with vertices p_0, p_8, p_7, p_6, p_5, p_4, p_3, p_2 and one right bound with vertices p_0, p_1 and p_2.

The clipped or output polygons we are after will be built in stages from sequences of "partial" polygons, each of which is a "V-shaped" list of vertices with the vertices on the left side coming from a left bound and those on the right side coming from a right bound and where there is one vertex in common, namely, the one at the bottom of the "V" which is at a local minimum. Let us use the notation $P[p_0p_1...p_n]$ to denote the partial polygon with vertices p_0, p_1, ... , p_n, where p_0 is the first point and p_n the last. The points p_0 and p_n are the top of the partial left and right bound, respectively. Some vertex p_m will be the vertex at a local minimum which connects the two bounds but since it will not be used for anything there is no need to indicate this index m in the notation. For example, one way to represent the polygon in Figure 6-1 would be as $P[p_6p_7p_8p_0p_1p_2p_3p_4p_5p_6]$ (with m being 3 in this case). Notice how the edges in the left and right bounds are not always to the right or left of the interior of the polygon here. In the case of a "completed" polygon, p_0 and p_n will be the same vertex at a local maximum, but at all the other intermediate stages in the construction of a polygon the vertices p_0 and p_n may not be equal. However, p_0 and p_n will always correspond to top vertices of the current left and right partial bounds, respectively. For example, $P[p_7p_8p_0p_1]$ (with $m = 2$) is a legitimate expression describing partial left and right bounds for the polygon in Figure 6-1. A good way to implement these

partial polygons is via a circularly linked list, or cycle, and a pointer that points to the last element of the list.

The algorithm now computes the bounds of the output polygons from the LML by scanning the world from the bottom to the top using what are called *scanbeams*. A scanbeam is a horizontal section between two scan lines (not necessarily adjacent), so that each of these scan lines contains at least one vertex from the polygons bus there are no vertices in between them. Figure 6-1 shows the scanbeams and the scan lines that determine them for that particular polygon. The scanbeams are the regions between the horizontal lines. It should be noted here that the scan lines that determine the scanbeams are not computed all at once but incrementally in a bottom-up fashion. The information about the scanbeams is kept in a scanbeam list (SBL) which is an ordered list of the y-coordinates of all the scan lines that define the scanbeams. This list of increasing values will be thought of as a stack. As we scan the world, we also maintain an active edge list (AEL) which is an ordered list consisting of all the edges intersected by the current scanbeam.

When we begin processing a scanbeam, the first thing we do is to check the LML to see if any of its bound pairs start at the bottom of the scanbeam. These bounds correspond to local minima and may start a new output polygon or break one into two, depending on whether the local minimum starts with a left-right or right-left edge pair. After any new edges from the LML are added to the AEL, we need to check for intersections of edges within a scanbeam. These intersections affect the output polygons and are dealt with separately first. Finally, we process the edges on a AEL.

6.2 Calculating the First Energy Absorption From Sunlight

Calculating the first energy absorption from sunlight is equivalent to solving a polygon clipping problem where the viewing perspective is the direction of the sun-ray. We

can get the sun-ray direction, p_s, easily from the solar position tracker.

Recall that we have constructed a BSP tree T of all the surfaces in our input, so now we can simply invoke Algorithm 1 on T and p_s. This returns a depth order of all the surface with respective to p_s. Let this order be $S = s_1, s_2, ..., s_n$, where s_1 is the surface closest to the sun (or the one fully visible with respective to p_s), and n is the number of surfaces. Let T be an empty list of surfaces and add s_0 to it. To get the planar surface that is invisible to the sun (and therefore capable of generating energy for the first absorption), we repeatedly clip the next surface in S against T and add the result to T. At the end, T stores all the planar, visible polygonal fragments with their respective material properties.

Of course, some polygon fragments in T come from non-panel surfaces, i.e, surfaces that are part of the buildings or landscape. This, however, can be easily detected by checking an indicator variable for each fragment. So we can simply go through all the panel-related polygonal fragments in T and sum up their energy absorption due to direct light light. However, computing each panel fragment's energy absorption cannot be solved exactly. There are various ways to approximate this value and they will be discussed in next chapter.

6.3 Calculating the Second Energy Absorption From Reflection

Calculating the second energy absorption from reflection is analogous to calculating the first energy absorption. For each panel P, we replace the direction of run ray with the directions of the reflected sun rays from all the panels visible to P (these can be determined via BSP tree and Algorithm 1). If there are k such panels, we carry out k corresponding polygon clipping calculations (just as the previous section). The sum of all the absorbed energy from these k calculations is the second energy absorption from reflection for panel P. To get the total second energy absorption, we repeat this procedure for all other panels.

Chapter 7

Solar Calculator Part III: Optics

The last part of solar calculator is to sum up all the energy generated from each panel fragment and over all time steps. In this chapter, we summarize all the physics and material-related features involved in this calculation process.

The use of Air Mass (AM) correction for solar flux allows for simulations during different seasons and at different latitudes, with reliable calculation of power curves and total energy. The computation has been generalized to account for cells of different efficiency and reflectivity within the structure, thus expanding the design opportunities to systems such as solar energy concentrators, where the mirrors are added as cells with zero efficiency and 100%. The Fresnel equations employed assume unpolarized sunlight.

Our algorithm considers a $3D$ assembly of N panels of arbitrary efficiency and optical properties, where the lth panel has refractive index n_l and conversion efficiency η_l ($l = 1, 2, ..., N$, and $\eta_l = 0$ for mirrors). The energy E (kWh/day) generated in a given day and location can be computed as:

$$E = \sum_{k=1}^{24/(\delta t)} P_k \dot{\delta} t$$

where δt is a time-step (in hours) in the solar trajectory allowing for converged energy, and P_k is the total power generated at the kth solar time step. The total energy over a period of time is obtained by looping over the days composing the period and

summing the energies generated during each day.

The key quantity P_k can be expanded perturbatively:

$$P_k = P_k^0 + P_k^1 + ... + P_k^m + ...$$

where the mth term accounts for m reflections of a light ray that initially hits the structure – and is thus of order R^m, where R is the average reflectivity of the absorbers – so that for most cases of practical interest, an expansion up to $m = 1$ suffices.

Explicitly, P_k^0 and P_k^1 can be written as:

$$P_k^0 = \sum_{l=1}^{N} l_k \times \eta_l \times A_{l,eff}[1 - R_l(\theta_{l,k})] \cos(\theta_{l,k}) \tag{7.1}$$

$$P_k^1 = \sum_{l=1}^{N} \{[I_k \times \eta_l \times A_{l,eff} R_l(\theta_{l,k}) \cos(\theta_{l,k})] \times \eta_s[1 - R_s(\alpha_{l_s,k})]\} \tag{7.2}$$

where l_k is the incident energy flux from the sun at the kth time-step (and includes a correction for Air-Mass effects), $A_{l,eff}$ is the unshaded area of the lth cell, and $R_l(\theta_{l,k})$ is the reflectivity of the lth cell fro an incidence angle (from the local normal) $\theta_{l,k}$ at the kth time-step, that is calculated through the Fresnel equations for unpolarized incident light. In eq. 7.2 the residual power after the absorption of direct sunlight (first square bracket) is transmitted from the lth cell and redirected with specular reflection to the sth cell that gets first hit by the reflected ray. The sth cell absorbs it according to its efficiency η_s and to its reflectivity calculated using the angle of incidence $\alpha_{l_s,k}$ with the reflected ray. How this is computed in practice was covered in previous chapter.

The empirical expression used to calculate the intensity of incident light on the Earth surface I_k (W/m^2) with AM correction for the Sun's position at the kth time-step is:

$$I_k \equiv I(\beta_k) = 1.1 \times 1353 \times 0.7^{[\frac{1}{\cos(\beta_k)}]^{0.678}}$$

where β_k is the Zenith angle of the sun ray with the Earth's local normal at the kth solar time step, $1353W/m^2$ is the solar constant, the factor 1.1 takes into account

(though in an elementary way) diffuse radiation, and the third factor contains the absorption from the atmosphere and an empirical AM correction. The angle β_k is calculated at each step of the Sun's trajectory for the particular day and location using a solar vector obtained from the solar position tracker.

Dispersion effects (dependence of optical properties on radiation wavelength) and weather conditions are not taken into account and are the main approximations of our model. Dispersion effects are fairly difficult to include, and would increase the computation time by a factor of $10 - 100$. Weather effects require reliable weather information (e.g., from satellites), and seem interesting to explore in light of recent work on optimization of PV output based on weather by [1].

Chapter 8

Optimization with Simulated Annealing

Simulated annealing (SA) is a local search algorithm that appear to be quite successful when applied to a broad range of practical problems. Note that, by restricting the panel docking sites to be the orthogonal grids, we have essentially discretize the search space for our problem. Now the panels only have a limited number of positions available to them. SA is particularly effective in this kind of discrete search problem.

We choose trial moves that preserves the optical properties of the single cells to favor the convergence of the optimization process. Our SA algorithm uses a standard Metropolis scheme for acceptance/rejection of trial moves, and a fictitious temperature T that is decreased during the simulation according to a specified cooling Schedule. Trial moves consisted in the change of a randomly chosen set of coordinates of the candidate structure. A number of coordinates varying between 1 and $9N$ (where N is the number of panels) are translated randomly within a cubic box of side $1 - 100\%$ the simulation box side (with periodic boundary conditions), thus determining a change δE in the total energy. When $\delta E > 0$, the move is accepted, whereas when $\delta E < 0$, the move is accepted with probability

$$P = e^{-|\delta E|/T}.$$

When a move is accepted, the structure is updated and a new random move is performed. Most SA simulation consisted of 100,000 steps with a converged value of the final energy. The code implement both power law and inverse-log cooling schedules; in most simulations we use the inverse-log cooling schedule

$$T(t) = \frac{c}{a + \log t}.$$

Average δA values for the given trial move were determined prior to running the optimization with a short simulation (1000 steps). Parameters for the cooling schedule were determined by imposing a temperature value such that the initial acceptance rate is $P = 0.99$ and the final acceptance is $P = 10^{-5}$.

As a comparison, we also try optimization with genetic algorithm (GA). Both algorithms give consistent results for optimization of $1, 2, 3, 4, 10, 20, 50$ cells, suggesting that both algorithms are capable of finding energy values near the global maximum using less than $100,000$ steps. Since the main cost of the simulation is the energy computation routine , the cost is comparable for both algorithms. For example, a $100,000$ steps long simulation with 20 cells is completed in $1 - 2$ days on a single processor. Parallelization of the code using standard MPI library is in the agenda, and could cut the computation time by a factor linear in the number of processors.

Chapter 9

Conclusion and Future Work

In this thesis, we present a algorithmic framework to model and optimize a 3D solar cell ensemble embedded in a given landscape. Because of the arbitrary constraints and the multi-dimensional factors involved in this optimization problem, solving it exactly is unrealistic. We strike an elegant trade-off by discretizing the search space and leveraging power data structures, e.g., binary space partition tree, in the pre-processing step. As a result, we are able to utilize the speed and accuracy of polygon clipping algorithm and simulated annealing to give the program good performance.

This is only the first step towards solving the grand "solar grid" problem. As a starter, the current 3DPV program has not been tested extensively. So far, all the positive results have been produced on simple input files with less than five buildings. A systematic test is in agenda.

Secondly, even though restricting the panel docking sites to pre-defined grids significantly simplifies the problem, it remains unknown whether too much accuracy is sacrificed. A simple (but computational expensive) test that runs on extremely small panel sizes can reveal how sub-optimal this simplification is. Essentially, this amounts to approximating the theoretical best result by going to the limit of the discrete case. Along the same line, it would be interesting to see if an alternative, grid-free approach could be possible.

Lastly, we chose simulated annealing as our optimization algorithm because we have discretized the search space. SA might not perform satisfactorily once we find a

grid-free approach to model our problem. Other approaches, such as gradient descent and convex optimization might become more suitable in that domain.

Appendix A

C++ Triangulation Code

```
#ifndef TRIANGULATE_H

#define TRIANGULATE_H

/********************************************************************/
/** Static class to triangulate any contour/polygon efficiently **/
/** You should replace Vector3d with whatever your own Vector
**/
/** class might be.  Does not support polygons with holes.
**/
/** Uses STL vectors to represent a dynamic array of vertices.
**/
/** I did not write the original code/algorithm for this
**/
/** this triangulator, in fact, I can't even remember where I
**/
/** found it in the first place.  However, I did rework it into **/
/** the following black-box static class so you can make easy
**/
```

```
/** use of it in your own code.  Simply replace Vector3d with
**/
/** whatever your own Vector implementation might be.
**/
/*************************************************************/


#include <vector>  // Include STL vector class.

class Vector3d
{
public:
        Vector3d() : mX(0), mY(0), mZ(0) {};
        Vector3d(float x, float y, float z)
        {
                Set(x, y, z);
        };

        float GetX(void) const { return mX; };

        float GetY(void) const { return mY; };

        float GetZ(void) const { return mZ; };

        float Modulus() const {
                return sqrt(mX*mX + mY*mY + mZ*mZ);
        };

        Vector3d operator*(const float scalar) const {
                return Vector3d(mX*scalar, mY*scalar, mZ*scalar);
```

```
};

Vector3d operator/(const float scalar) const {
        return Vector3d(mX/scalar, mY/scalar, mZ/scalar);
};

Vector3d operator+(const Vector3d &p) const {
        return Vector3d(mX + p.GetX(), mY + p.GetY(), mZ + p.GetZ(
};

Vector3d operator-(const Vector3d &p) const {
        return Vector3d(mX - p.GetX(), mY - p.GetY(), mZ - p.GetZ(
};

Vector3d &operator=(const Vector3d &p) {
        if (this != &p) {
                this->mX = p.GetX();
                this->mY = p.GetY();
                this->mZ = p.GetZ();
        }
        return *this;
};

Vector3d &operator*=(const float scalar) {
        this->mX *= scalar;
        this->mY *= scalar;
        this->mZ *= scalar;
        return *this;
}
```

```
Vector3d &operator/=(const float scalar) {
        this->mX /= scalar;
        this->mY /= scalar;
        this->mZ /= scalar;
        return *this;
}

Vector3d &operator+=(const Vector3d &p) {
        this->mX += p.GetX();
        this->mY += p.GetY();
        this->mZ += p.GetZ();
        return *this;
};

Vector3d &operator-=(const Vector3d &p) {
        this->mX -= p.GetX();
        this->mY -= p.GetY();
        this->mZ -= p.GetZ();
        return *this;
};

float Dot(const Vector3d &p) const {
        return (mX*p.GetX() + mY*p.GetY(), mZ*p.GetZ());
};

Vector3d Cross(const Vector3d &p) const {
        float x = mY*p.GetZ() - mZ*p.GetY(),
                y = mZ*p.GetX() - mX*p.GetZ(),
                z = mX*p.GetY() - mY*p.GetX();
        return Vector3d(x, y, z);
```

```
        };

        Vector3d &Normalize() {
                float norm = Modulus();
                if (norm > 0) {
                        (*this) /= norm;
                }
                return *this;
        };

        void  Set(float x, float y, float z)
        {
                mX = x;
                mY = y;
                mZ = z;
        };

private:
        float mX;
        float mY;
        float mZ;
};

// Typedef an STL vector of vertices which are used to represent
// a polygon/contour and a series of triangles.
typedef std::vector< Vector3d > Vector3dVector;


class Triangulate
{
```

```
public:

        // triangulate a contour/polygon, places results in STL vector
        // as series of triangles.
        static bool Process(const Vector3dVector &contour,
                Vector3dVector &result);

        // compute area of a contour/polygon
        //static float Area(const Vector3dVector &contour);

        // compute the signed are of a contour/polygon
        static float SignedArea(const Vector3dVector &contour, Vector3d *n

        // decide if point Px/Py is inside triangle defined by
        // (Ax,Ay) (Bx,By) (Cx,Cy)
        static bool InsideTriangle(const Vector3d &a, const Vector3d &b, c
                const Vector3d &p);

private:
        static float CalcAngleSum(const Vector3d &q, const Vector3dVector &
        static bool Snip(const Vector3dVector &contour, const Vector3d &no
                int u, int v, int w, int n, int *V);

};


#endif

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <assert.h>
#include <math.h>

#include "triangulate.h"

// constants
#define EPSILON 0.0000000001f
#define TWOPI 6.28318530717958647692528 7

//float Triangulate::Area(const Vector3dVector &contour)
//{
//
//      int n = contour.size();
//
//      float A = 0.0f;
//
//      for (int p = n - 1, q = 0; q<n; p = q++)
//      {
//              A += contour[p].GetX()*contour[q].GetY() - contour[q].GetX
//      }
//      return A*0.5f;
//}

// only works for convex polygons
float Triangulate::CalcAngleSum(const Vector3d &q, const Vector3dVector &p
{
        int n = polygon.size();
        float m1, m2;
        float anglesum = 0, costheta;
```

```
        Vector3d p1, p2;

        for (int i = 0; i<n; i++) {
                p1 = polygon[i] - q;
                p2 = polygon[(i + 1) % n] - q;

                m1 = p1.Modulus();
                m2 = p2.Modulus();
                if (m1*m2 <= EPSILON)
                        return(TWOPI); /* We are on a node, consider this
                else
                        costheta = p1.Dot(p2) / (m1*m2);

                anglesum += acos(costheta);
        }
        return(anglesum);
}

float Triangulate::SignedArea(const Vector3dVector &contour, Vector3d *norı
        int n = contour.size();
        if (n < 3) return 0;

        Vector3d sum;
        Vector3d v1 = contour[1] - contour[0], v2 = contour[2] - contour[1
        *normal = (v1.Cross(v2)).Normalize();

        for (int i = 0; i < n; i++) {
                sum += (contour[i].Cross(contour[(i + 1) % n]));
        }
```

```
        return ((*normal).Dot(sum)/2);
}

/*
InsideTriangle decides if a point P is Inside of the triangle
defined by A, B, C.
*/
bool Triangulate::InsideTriangle(const Vector3d &a, const Vector3d &b, con
        const Vector3d &p) {
        Vector3dVector triangle;
        triangle.push_back(a);
        triangle.push_back(b);
        triangle.push_back(c);

        if (CalcAngleSum(p, triangle) == TWOPI) return true;
        else return false;
}

//bool Triangulate::InsideTriangle(float Ax, float Ay, float Az,
//      float Bx, float By, float Bz,
//      float Cx, float Cy, float Cz,
//      float Px, float Py, float Pz)
//
//{
//      float ax, ay, bx, by, cx, cy, apx, apy, bpx, bpy, cpx, cpy;
//      float cCROSSap, bCROSScp, aCROSSbp;
//
//      ax = Cx - Bx;  ay = Cy - By;
//      bx = Ax - Cx;  by = Ay - Cy;
//      cx = Bx - Ax;  cy = By - Ay;
```

```
//        apx = Px - Ax;   apy = Py - Ay;
//        bpx = Px - Bx;   bpy = Py - By;
//        cpx = Px - Cx;   cpy = Py - Cy;
//
//        aCROSSbp = ax*bpy - ay*bpx;
//        cCROSSap = cx*apy - cy*apx;
//        bCROSScp = bx*cpy - by*cpx;
//
//        return ((aCROSSbp >= 0.0f) && (bCROSScp >= 0.0f) && (cCROSSap >= 0
//};

bool Triangulate::Snip(const Vector3dVector &contour, const Vector3d &norm
        int u, int v, int w, int n, int *V)
{
        Vector3d p0 = contour[V[u]], p1 = contour[V[v]], p2 = contour[V[w]
        Vector3d localNormal = (p1 - p0).Cross(p2 - p1);
        if (normal.Dot(localNormal) < 0) return false;

        //if (EPSILON > (((Bx - Ax)*(Cy - Ay)) - ((By - Ay)*(Cx - Ax)))) r

        // check if any other point of the polygon is inside the triangle;
        // in concave polygons.
        for (int p = 0; p<n; p++)
        {
                if ((p == u) || (p == v) || (p == w)) continue;
                if (InsideTriangle(p0, p1, p2, contour[V[p]])) return fals
        }

        return true;
}
```

```
bool Triangulate::Process(const Vector3dVector &contour, Vector3dVector &r
{
        /* allocate and initialize list of Vertices in polygon */

        int nv = contour.size();
        if (nv < 3) return false;

        int *V = new int[nv];
        for (int v = 0; v<nv; v++) V[v] = v;

        ///* we want a counter-clockwise polygon in V */

        //if (0.0f < Area(contour))
        //for (int v = 0; v<n; v++) V[v] = v;
        //else
        //for (int v = 0; v<n; v++) V[v] = (n - 1) - v;

        /* the canonical normal, i.e., the one based on the cross product
        of two consecutive vectors at a convex vertex*/
        Vector3d normal;
        float area = SignedArea(contour, &normal);
        if (area < 0) normal *= -1;

        /*  remove nv-2 Vertices, creating 1 triangle every time */
        int count = 2 * nv;   /* error detection */

        for (int m = 0, v = nv - 1; nv>2;)
        {
                /* if we loop, it is probably a non-simple polygon */
```

```
            if (0 >= (count--))
            {
                    //** Triangulate: ERROR - probable bad polygon!
                    return false;
            }

            /* three consecutive vertices in current polygon, <u,v,w> :
            int u = v; if (nv <= u) u = 0;      /* previous */
            v = u + 1; if (nv <= v) v = 0;      /* new v
*/
            int w = v + 1; if (nv <= w) w = 0;      /* next
*/

            if (Snip(contour, normal, u, v, w, nv, V))
            {
                    int a, b, c, s, t;

                    /* true names of the vertices */
                    a = V[u]; b = V[v]; c = V[w];

                    /* output Triangle */
                    result.push_back(contour[a]);
                    result.push_back(contour[b]);
                    result.push_back(contour[c]);

                    m++;

                    /* remove v from remaining polygon */
                    for (s = v, t = v + 1; t<nv; s++, t++) V[s] = V[t]
```

```
                    /* resest error detection counter */
                    count = 2 * nv;
                }
            }



            delete V;

            return true;
}
```

Appendix B

BSP Tree Size Analysis

Theorem 1. *The expected number of fragments generated by the algorithm 2D-Random-BSP is* $O(n \log n)$.

Proof. Let s_i be the fixed segment in S. We shall analyze the expected number of other segments that are cut when $l(s_i)$ is added by the algorithm as the next splitting line.

When the line through a segment is used before $l(s_i)$, then it shields s_j from s_i. This leads us to define the distance of a segment with respect to the fixed segment s_i: $dist_{s_i}(s_j)$ = the number of segments intersecting $l(s_i)$ in between s_i and s_j if $l(s_i)$ intersects s_j; or $+\infty$, otherwise. For any finite distance, there are at most two segments at that distance, one on either side of s_i.

Let $k = dist_{s_i}(s_j)$, and let $s_{j_1}, s_{j_2}, ..., s_{j_k}$ be the segments in between s_i and s_j. We get that

$$Pr[l(s_i) cuts s_j] \leq \frac{1}{dist_{s_i}(s_j) + 2}.$$

Notice that there can be segments that are not cut by $l(s_i)$ but whose extension shields s_j. This explains why the expression is not an equality.

We can now bound the expected number of cuts generated by s_i:

$$E[number of cuts generated by s_i] \leq \sum_{j \neq i} \frac{1}{dist_{s_i}(s_j) + 2} \leq 2 \sum_{k=0}^{n-2} \frac{1}{k+2} \leq 2 \ln n.$$

□

Bibliography

[1] Lave, M., Kleissl, J. *Solar variability over various timescales using the top hat wavelet.* American Solar Energy Society Conference, in press, 2011.

[2] REN21, *Renewables 2010 Global Status Report, 19.* See `http://www.ren21.net`, 2010.

[3] S.H. Park, S. Roy, S. Beaupre, S. Cho, N. Coates, J.S. Moon, D. Moses, M. Leclerc, K. Lee, and A.J. Heeger, *Nature Photonics.* 3, 297, 2009.

[4] N.S. Lewis, *Science.* 315, 798, 2007.

[5] M. Helgesen, R. Sondergaard, F.C. Krebs, *J. Mater. Chem.* 20, 36, 2010.

[6] B. Myers, M. Bernardi, J.C. Grossman, *Appl. Phys. Lett..* 96, 071902, 2010

[7] M. Bernardi, N. Ferralis, J.H. Wan, R. Villalon, J.C. Grossman, *Energy and Environmental Science* 5, 6880, 2012.