Project 2: Rasterization for Accelerating Geometric Queries

ITCS 6120 Computer Graphics
Fall 2009
Professor Zachary Wartell

Revision: 9/17/2012 4:34:32 PM

Copyright August 2006, Zachary Wartell @ University of North Carolina at Charlotte
All Rights Reserved.

Due Date: Part I: Wednesday, September 26th 11:59PM
Part II: Sunday, October 7th 11:59PM

Turnin via SVN

1. Purpose

The project involves some 2D OpenGL programming and using the GLUT library to provide handling of input events and simple GUI components. The key algorithm in this assignment is implementing a rasterization algorithm that converts a coordinate representations of a line segment to a raster representation.

2. Prerequisite Reading

The class website discusses several resources for these toolkits. Most of the OpenGL commands needed for this assignment all come from the first three chapters of the textbook. Additionally, you should use OpenGL Redbook and GLUT documentation linked from the class webpage. The class notes and the text book discuss the Breshenham line drawing algorithm which this assignment builds upon.

3. Problem Statement

Rasterization algorithms convert coordinate representations of geometry to raster representations. These algorithms have multiple uses in 2D and 3D geometric computations. In class we focused on using 2D rasterization algorithms to efficiently scan convert the pixels on a 2D raster grid that are traced by line segments, circles and polygons. We also discussed briefly how in both 2D and 3D, rasterization algorithms can be used for various other geometric computations such as line-of-site queries and collision detection queries.

In this assignment you will derive and implement a modified version of the Breshenham mid-point based line algorithm to accelerate part of a collision detection computation. In this assignment you will *not* have to implement the entire collision detection computation. You will only implement the part of the algorithm that uses rasterization to accelerate the complete algorithm. While the modified rasterization algorithm itself is not too complicated, the motivation requires some additional background.

Consider a 2D environment consisting of discs (**Figure 1**). There are green discs of a constant size that appear at random fixed locations. The user specifies a start and end point (**A** and **B**) and a single red disc starts at location **A** and moves to location **B.** The goal of the algorithm is to detect all the green discs that the red disc passes over while moving between **A** and **B**.

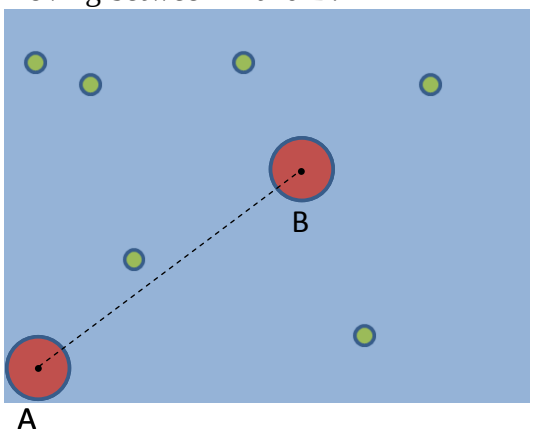

Figure 1: Basic Problem

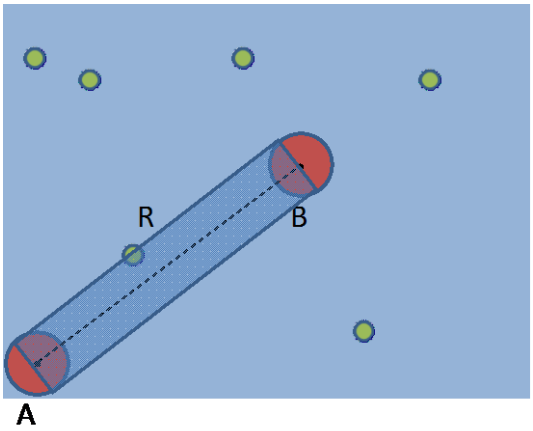

Figure 2: Rectangle-Disc Solution

Approach I: Naïve…

The problem requires computing whether the red circle intersects each and every green circle for all possible positions of the red circle along the line segment path (**A**, **B**). Assume there are *g* green discs. Naïve Approach I is to move the red circle through a fixed number, *s*, of small steps along path (**A**, **B**) and test for intersections at every step with all green discs. This requires $g \cdot s$ calls to a method, `intersects (Circle c0, Circle c1)`, that tests whether two discs intersect. One problem with Approach I is you do not know ahead of time how small each step should be in order to both avoid missing any green disc intersections while at the same time avoiding having the step count, *s*, be too high and causing the algorithm to be slow.

Approach II: Better but still slow

A better approach is to construct a "path rectangle", **R**, that is parallel to the path (**A**, **B**) and whose edges terminate at the ends of segment (**A**, **B**) as shown in Figure 2. In Approach II, we would need a method, `intersects (Disc d, Rectangle r)`, that computes whether **d** intersects rectangle **r.** Approach II avoids the problem with having to guess the right the number of steps, *s*, as required in Approach I. Further Approach II, must perform the circle-rectangle intersection only *g* times whereas Approach I had to

computer circle-circle intersections $g \cdot s$ times. Still Approach II is inefficient when we have a very large number of green discs, g. This leads to the idea for Approach III.

Approach III: Getting faster….

Approach III greatly reduces the number of circle-rectangle intersection tests by preprocessing the green discs into a raster grid of cells and then using an efficient modified line rasterization algorithm to quickly determine which grid cells the "path rectangle", **R**, covers. We then only perform the circle-rectangle intersection tests on those green discs that fall inside the cells covered by **R.** We can develop a very fast, efficient, all integer arithmetic algorithm for determining the covered cells of **R.** Approach III will be much faster than Approach II given the following reasonable assumptions:

1. The number of covered cells of **R** is much less than g the total number of green discs.
2. The computation per covered cell of **R** is much faster than the computation per disc-rectangle intersection test. (This will be even more true if we replace our discs with shapes with more mathematically complicated boundaries).

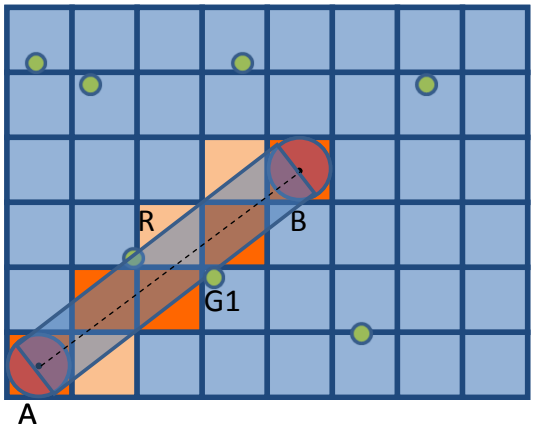

Figure 3: Grid Cells + Rasterization

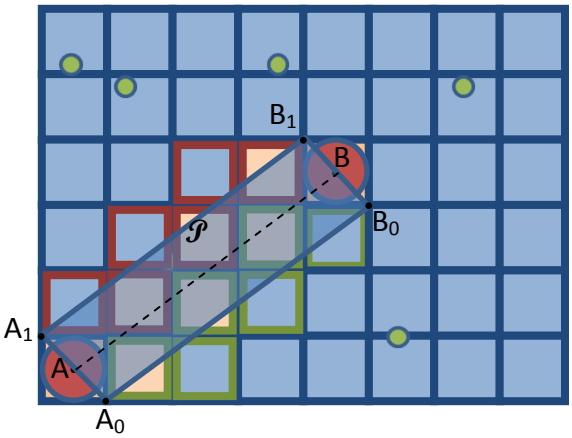

Figure 4: Geometry of final algorithm

To develop Approach III we start with the idea illustrated by **Figure 3**. Our first idea is use the standard line rasterization algorithm to trace cells on line segment (**A**,**B).** Geometrically, however, this is inadequate because the Breshenham algorithm only picks cells/pixels whose centers are closest to the line segment (in dark orange) and we need to visit all cells intersected by the line segment (both the dark and light orange cells). To traverse the intersected cells, the Breshenham mid-point algorithm needs to be modified.

Even with the above modification, however, our work is only partially done. Consider green disc **G1**. Clearly, the red disc will collide with **G1** (**G1** interects **R)** but **G1** is not a member of any the cells intersected by segment (**A**,**B)**, so our current algorithm will

never test the cell that contains **G1**. The problem is that we need to scan convert not just the line segment (**A**,**B**) but also the line segments forming the upper and lower boundary of **R**. The difficulty with the line segment forming these boundaries is that their end-points are at the tangents to the red discs and these points' locations are not at integer coordinates (nor multiples-of-½ coordinates).

A good solution is to replace rectangle **R** with parallelogram $\mathcal{P}$ (Figure 4) where $\mathcal{P}$'s boundary line segments connect to the corners of the cells at **A** and **B** rather than the edges of the red disc. This is illustrated in Figure 4 where we make line segments ($\mathbf{A_0}$, $\mathbf{B_0}$) and ($\mathbf{A_1}$, $\mathbf{B_1}$). Note that these boundary lines of $\mathcal{P}$ are parallel to (**A**,**B**). The cells intersected by ($\mathbf{A_1}$, $\mathbf{B_1}$) are shown in dark red outlines. The cells intersected by ($\mathbf{A_0}$, $\mathbf{B_0}$) are shown in dark green outlines. Note, the pattern of visited cells for segment ($\mathbf{A_0}$, $\mathbf{B_0}$) is exactly the same as for segment ($\mathbf{A_1}$, $\mathbf{B_1}$). The similarity means that if we create a rasterization algorithm for line segment ($\mathbf{A_0}$, $\mathbf{B_0}$), we can modify the resulting code to visit the cells along ($\mathbf{A_1}$, $\mathbf{B_1}$) as the code simultaneously visits the cells of ($\mathbf{A_0}$, $\mathbf{B_0}$). Once we have an algorithm that quickly visits all cells in $\mathcal{P}$**,** we collect the green discs that were pre-sorted into these cells. Additionally, we must remember to collect discs in the two cells at **A** and **B**. With this disc collect, we then perform a disc-rectangle intersection test to determine whether each disc actually overlaps the original path rectangle **R.** (This is needed because the list of green discs return by the rasterization part will occasionally contain a disc not contained in the original **R**).

4. Your Task

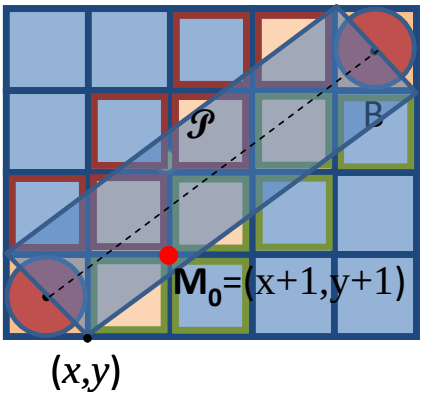

Figure 5: Coordinates and Midpoint

Your mathematical task is to first derive the algebra to make a fast, efficient all integer algorithm for scan converting all cells intersected by segment ($\mathbf{A_0}$, $\mathbf{B_0}$). You will need to perform an algebraic derivation similar to the one used for the mid-point line algorithm. Unlike our mid-point algorithm derivation, I suggest you assume that the integer coordinates are at a cell's lower-left corner shown as (*x*,*y*) in Figure 5. To further help you get started, observe that to visit all intersected cells, we need to change the mid-point to a different offset from our current cell/pixel location (*x*,*y*). Rather than an offset of (1,1/2) for Octant I as done in the line drawing algorithm, you should use an offset of (1,1). (See Figure 5 point $\mathbf{M_0}$). Sketch some examples for yourself and make sure you

understand why this change will traverse all intersected cells, not just the standard Breshenham ones (i.e. both the dark and light orange cells **Figure 3** not just the dark orange cells).

As with the standard Breshenham algorithm, the idea is to plug the coordinate of the mid-point, **M,** into the implicit equation of the line, ***F***, and evaluate the result. However, unlike the standard Breshenham algorithm, in our algorithm we distinguished three possible outcomes instead of just two. The three are: $\boldsymbol{F}(\mathbf{M}) > 0$, $\boldsymbol{F}(\mathbf{M}) < 0$, and $\boldsymbol{F}(\mathbf{M}) = 0$. These 3 correspond to stepping to: the right cell, the upper cell or the diagonal cell.

4.1. Algorithms and Math

Due: Wednesday, September 26th 11:59PM

You must submit via SVN your sketches and derivations and pseudo-code for your algorithms by the above date. You should be started on the code as well by this point.

Rasterization –

Step 1: Make some sketches or illustrations to help you understand the rasterization problem and label the important the geometry, etc.

You must submit your sketches or illustrations electronically. It is fine to scan in your hand drawn illustrations into a .PNG file.

Step 2: Derive the algebra to make an all integer approach for scan converting all cells intersected by a line segment for Octant I.

Step 3: Determine whether your algorithm needs a different formula for lines in Octant II or whether the same formula's from Octant I work for Octant II.

Step 4: Decide on a strategy for the other eight octants.

Disc-Rectangle Intersection

Step 1: Make some sketches or illustrations to help you understand the problem of testing whether a disc intersects a rectangle.

You must submit your sketches or illustrations electronically. It is fine to scan in your hand drawn illustrations into a .PNG file.

Step 2: First develop an algorithm for performing a point-rectangle intersection test (i.e. is the point inside the rectangle). The algebra we reviewed regarding line equations is sufficient for creating this algorithm.

Step 3: Extend the above algorithm to test for a disc-rectangle intersection. As a hint, note the following, if the disc center is inside the rectangle then the disc and rectangle intersect! Additionally if the disc center is with a distance *r* from any edge of the rectangle where *r* equals the disc radius, then the disc also intersects the rectangle. Computing the distance between a point **P** and line segment ***l*** defined by points (**P0**,**P1**) requires that you compute the two distances, *d0* and *d1*, between the **P** and the segments two endpoints**, P0** and **P1**, and the distance *dl* between the point **P** and the line (see textbook pg673). [Draw a sketch!] The distance *d* between **P** and (**P0**,**P1**) is *dl* if and only if the dot product of vector **P1 - P0** with vector **P - P0** is negative, otherwise *d* is the smaller of *d0* and *d1*.

4.2. Code

Due: Sunday, October 7th 11:59PM

SVN update to get the skeleton code for Project 2. The primary functionality provided by this skeleton code is the ability to pan and zoom with the mouse. You should render all your geometry in 2D integer world coordinates inside the axis-aligned rectangle with corners [(0,0), (1,000,000, 1,000,000)]. This rectangular region in world coordinates will be referred as the *play area*. The provided pan/zoom interface lets you pan and zoom over the play area.

Create a data structure called `Grid` that represents a 1000 by 1000 regular grid of cells that cover the play area in world space. For each cell, `Grid` must be able to store a set of pointers to all the green discs that fall into that cell. Generate 1 million green discs at random x,y locations in the play area and insert each disc into all cells in `Grid` that intersect each disc. (Note, a disc could lie in 1 to 4 disc).

(Make the green discs diameter is ½ the width of a cell).

Your program should draw the grid cells and all the green discs using OpenGL commands. In addition to using the pan/zoom interface, the user should be able to select two grid cells by right clicking in two cells with the mouse. After such a selection, your program should immediately draw the following:

- the parallelogram that connects the corners of the selected cells
- the cells intersected by one boundary edge of the parallelogram should be filled in red. (The outline of each grid cell should still be visible)
- the cells intersected by the other boundary edge of the parallelogram should be filled in blue. (The outline of each grid cell should still be visible)

Next, when the user hits the space bar once, the program should visually highlight all the green discs that are contained in any of the highlighted cells.

When the user hits the space bar a second time, your program should delete all the green discs that are actually contained in the path rectangle **R**.

When the user hits the space bar a third time, your program should return any of the remaining highlighted green discs to their original appearance.

At this point, the user should be free to pan and zoom and to select another two grid cells to repeat the above process.

The assignment does not require the actually animation of the red disc moving between its two locations or the animation of each green disc being knocked out of the way as the red disc hits each one. Additional work is needed to do this after computing all the green discs that overlap the path rectangle and this work is beyond the scope of this project.