Puzzles in 2D and 3D Visualized with DSGI and Perspective Mapping

Ted Clay, Clay Software and Statistics, Ashland, Oregon

ABSTRACT

Do you enjoy puzzles? SAS® can be used to solve not only abstract problems but also the physical kind of puzzles you can hold in your hand. An algorithm is presented which finds all possible solutions to puzzles which require fitting a collection of pieces in various shapes into a limited space. This is applied to (1) Pentaminos, a flat puzzle with 12 pieces to be fit into a 6x10 rectangle, and (2) a 3x3x3 cubic analog to Pentaminos. Solutions are presented using SAS/Graph® and its DSGI component.

INTRODUCTION

This paper presents a SAS® program which solves certain kinds of puzzles. The goal of this presentation is both share my own fascination with puzzles and to pass along some SAS® techniques that were required along the way. In particular, the techniques for displaying 3-D objects can be applied to many other problems.

There are two basic requirements in any searching algorithm: (1) to know where you are in the search, and (2) to test out your alternatives in an organized sequence. If you have tried to solve puzzles yourself, you may know that part of the difficulty is getting lost and finding that you end up going in circles, trying the same dead end path more than once. The computer, at least, is good at keeping track of a search, and making up with speed what it lacks in insight.

PENTAMINOS

In the Pentaminos puzzle, there are 12 pieces, each of them a different configuration of 5 squares. That makes 60 squares total. (As an aside, these 12 pieces are the only ways 5 squares can be combined.) They are to be packed perfectly into a board that is a 6 by 10 rectangle. The pieces can be flipped over and rotated in any direction to fit into the board.

Each piece has its own symmetry properties. The most symmetrical piece, the cross, has only one orientation, while the most asymmetrical pieces have 8 different ways in which the piece can be placed on the board. In all, there are 63 different orientations of the 12 pieces. I use the word "shape" to mean a piece in a particular orientation. Figure 1 shows each of the pieces and their 63 different shapes.

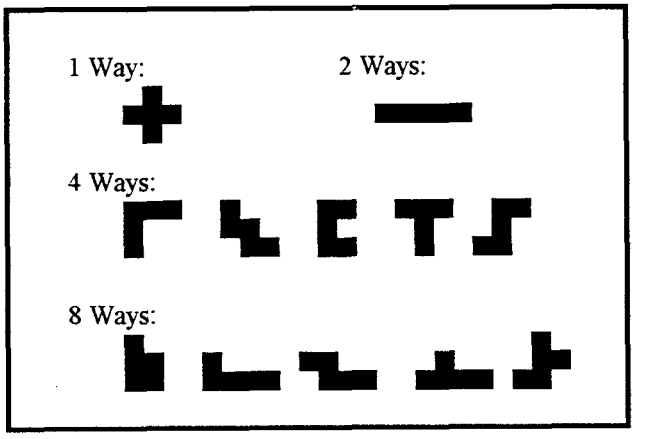

Figure 1 -- The 12 Pentaminos pieces showing how many ways each can be oriented as "shapes"

So we have the concepts of a ***board***, and ***shapes***, which belong to ***pieces***. In its simplest form, the solution algorithm we use is as follows: Repeatedly pick an empty space on the board, which we will call the ***target space***, and try to fill it using one of the shapes. If it fits, you put the shape on the board and pick another target space to try to fill. If no shape can be found to fill a space, you remove the most recently placed shape and try the next shape after the one you just removed. Obviously the fewer things you have to try, the quicker the solutions will be found. We will come back to this algorithm in more detail, but first we must deal with the issue of how these concepts are represented in the computer.

Data Structures and Coordinates

The board is represented in a SAS data step as an array, in this case two-dimensional.

```
ARRAY BOARD_ {0:5,0:9} board1-board60;
```

One can think of the (0,0) square as being in the top left corner of the board, and coordinates representing points as (col, row) or (x,y).

The shapes are also represented by a series of (x,y) points relative to some origin. Obviously the origin could be anywhere, but it made the most sense to have the origin or (0,0) square be one of the five squares making up the shape. There are 5 squares in each piece, so there are 5 different ways to define the coordinates of a shape. Since one of the five points is always the origin, with coordinates (0,0), we only really need to store the coordinates of the other four points. We let these be stored in SAS variables X1-X4 and Y1-Y4. We now have all the machinery to "place" a shape on the board. We adopt the convention that the origin square on the shape is the one that fills the target square on the board. The following SAS code shows how a shape could be placed on the board at the column BDX and row BDY of the board (assuming that it fits).

```
ARRAY X_ {*} X1-X4;
ARRAY Y_ {*} Y1-Y4;
BOARD_(BDX,BDY) = shapenum;
DO I=1 TO 4;
  BOARD_(BDX+X_(I),BDY+Y_(I) )= shapenum;
END;
```

Notice that we mark a square on the board as "taken" by storing a shape number in the array element for that square. So 5 board array elements would be assigned the same shape number. Solutions are stored by outputting an observation with the board array variables.

So we see that each alternative to be tested can be represented as a set of coordinates. Now let's look at the following question: How many different sets of coordinates is it necessary to search? Suppose you pick an arbitrary square on the board and want to fill it. There are five different ways that a given shape could be placed onto the empty square. (Not all of them would fit, of course, but we can't know that until we test each one.) Each way corresponds to picking a different square as the origin square for that shape. If there are 63 different shapes, there must be 5 times 63 or 315 different sets of coordinates to be tested.

We could construct an array of four X,Y coordinates for a total of 315 different combinations to try, but a convenient trick allowed us to cut the possibilities down to 63.

A Helpful Trick

The above discussion is true if you are considering how to fill **any** empty square on the board. But if you pick a **particular** empty space, you only have to consider one choice of origin per shape, which reduces the number of alternatives to try down from 315 to 63.

The convention is as follows: Always pick the upper-left-most empty square. Then you only need to consider cases where the origin for the shape is the upper-left-most out of the 5 squares on the piece. (To be exact, by "upper-left-most" is defined as following: Locate the minimum row of the squares having the minimum column.) Figure 2 gives an illustration of a partially filled board, and a shape which has its origin in the upper-left corner.

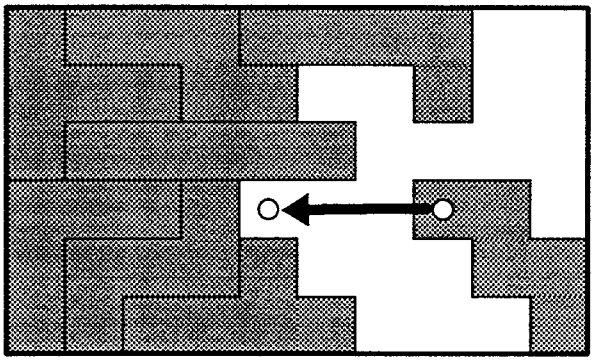

Figure 2: The "upper-left" empty space and the "upper-left" square of the shape being placed.

So the data structure required to store the list of shapes to be tested consists of a matrix which is 63 shapes by 4 points by 2 dimensions (row and column). For clarity these are stored as two arrays instead of one:

```
ARRAY X_ {63,4) _temporary_;
ARRAY Y_ {63,4) _temporary_;
```

The Solution Algorithm

There is only one more data structure that has not been mentioned. This is a stack of "Placed Pieces". Whenever a piece is placed on the board, a row is added to this table, recording the (x,y) coordinates of the target square that was filled, and the number of the shape that filled it. When a piece is removed from the board, the (x,y) coordinates of the target square that was filled by that piece become the new target square to be filled, and we continue the search starting after the number of the shape that was just removed. For Pentaminos, this table would need room to hold all 12 pieces. It works as a classic stack, with a

pointer variable holding the index of the top of the stack.

A shape is marked as "unavailable" if some other shape from the same physical piece is already placed on the board.

The objective was to locate all possible solutions to the puzzle, not just one. When a solution was found, the program outputs a SAS observation to a dataset, allowing further manipulation of the solutions later on. Add some attention to initialization and termination, and you have a finished algorithm. Figure 3 illustrates the overall algorithm.

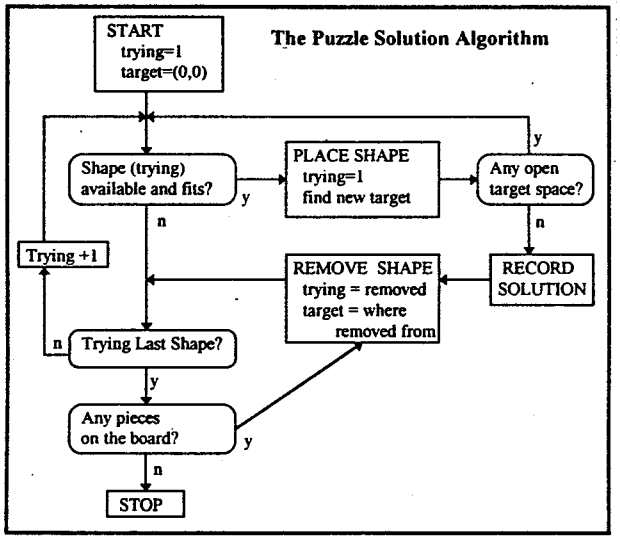

Figure 3: The general algorithm for puzzle-solving.

The variable "Trying" points to the shape number we are trying to fit into the current target square on the board. The most frequently-executed operation the one which tests whether the current shape fits into the current target square. The "Remove" process pops the stack of placed pieces, and sets to missing the occupied squares on the board. The key point is that trying a piece and having it not fit is equivalent to placing a piece and eventually reaching a dead end. In either case, you go on to the next shape in the sequence. Also, in the search for all possible solutions, finding a solution is logically equivalent to any other dead-end in the search, and requires us to back up, after outputting the solution, of course. Any reader able to "structure" the above algorithm can contact the author for congratulations.

The Solutions to Pentaminos

There are 9356 solutions to Pentaminos. When you flip and rotate the solutions, you find that there are a mere 2339 truly unique solutions. It was possible to use the statistical tools of SAS® to find unusual solutions among the entire set. Figure 4 shows the only unique solution which has the long straight piece in the location shown.

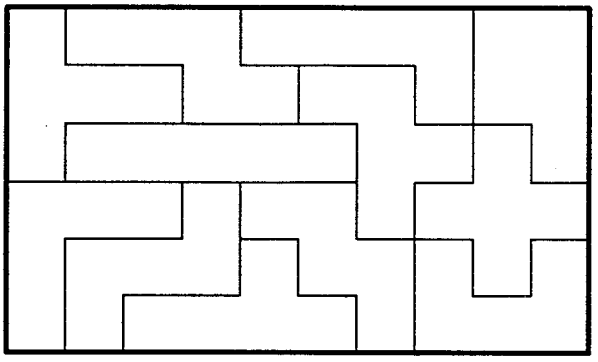

Figure 4: The most unique out of 9356 solutions.

It was very simple to change the dimensions of the Pentaminos board from 6 by 10 to other shapes. Here are some of the other shapes and the number of unique solutions to them:

6 by 10 has 2339 unique solutions.
5 by 12 has 1010 unique solutions.
4 by 15 has 368 unique solutions.
3 by 20 has 2 unique solutions.
8 by 8 minus 2 by 2, a square with a hole in the middle, has 130 unique solutions. Interestingly enough, this is much more than the similar 3 by 20 shape.

DSGI Techniques

A way was needed to illustrate the complete or partial solutions to Pentaminos, using the Data Step Graphics Interface component of SAS/GRAPH®. The following DSGI calls were placed inside a DATA step to create the graphic segments just showing the pieces placed on the Pentaminos board. They illustrate the minimum set of DSGI statements needed for doing a graphical display.

```
rc=gset('CATALOG','sas','pentacat');
rc=ginit();
rc=graph('CLEAR');

* set up color numbers assigned to each
of 12 pieces;
```

```
rc=gset('COLREP',1,'RED');
rc=gset('COLREP',2,'BLUE');
                ...
rc=gset('COLREP',12,'YELLOW');

rc=gset('FILTYPE','SOLID');

do row=0 to 5;
do col=0 to 9;
   * Look up the piece number (1-12);
   piece = piece_(board_(row,col));
   * color number = piece number;
   rc=gset('FILCOLOR',piece);
   * 4 corners of  square at row,col;
   rc=gdraw('FILL',4,<coords>);
end;
end;

rc=graph('UPDATE');
rc=gterm();
```

The above code was used to display solutions using color to distinguish one piece from the next. For this paper a modified version was used which drew only the edges in black.

A 3-D PUZZLE

A 3-dimensional analog to Pentaminos turned out to be a very simple modification to the Pentaminos program. What proved to be the real challenge with this puzzle was how to visualize the puzzle and its solutions in two dimensions. This lead to some research into the basic techniques of how to map 3D into 2D with perspective.

The 7 puzzle pieces consist of 4 unit cubes glued together in various shapes. (A unit cube is a cube with an edge of length 1.) They are to be fit together into a larger cube with 3 units along each edge, for a total of 27 unit cubes. Since this is clearly impossible, one of the 7 pieces is made up of only 3 unit cubes.

This puzzle has 11,520 solutions, which are made up of the 24 possible rotations of 480 unique solutions.

One of the puzzle pieces is shown in Figure 5. Notice the forshortening and apparent distortion created by the perspective transformation of the pieces. They should appear as if they were viewed through a camera placed a finite distance away from the objects in the real world. This image was created using the DSGI. The basic outline of the DSGI program is the same as above with Pentaminos, with the exception that the polygons are no longer square, and hidden surfaces were not rendered.

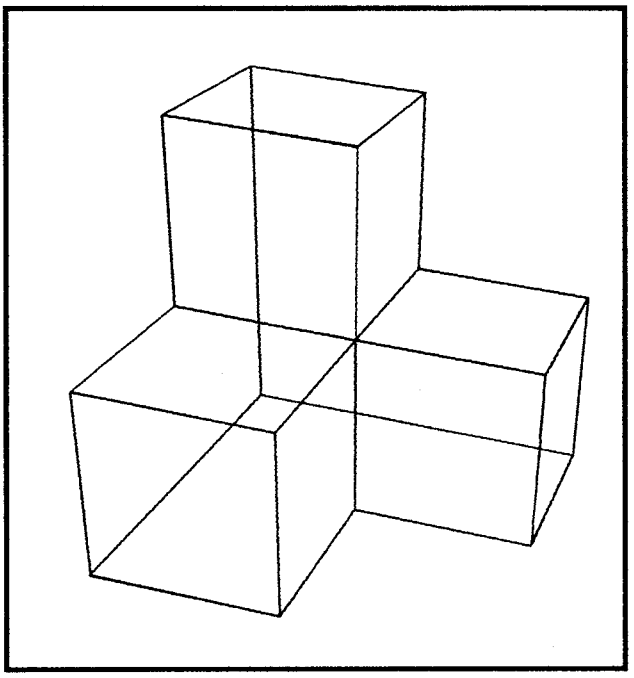

Figure 5: A 3-D rendering of a puzzle piece using DSGI within a data step.

The Simplest of 3-D Theory

The 5 steps in displaying a 3-D image in a 2-D medium are as follows:

Step 1. Build a set of 3-D points using a set of natural coordinates known as the "world coordinates." Do this by picking an origin point, a set of X,Y and Z axes. Then describe where every other point of interest lies relative to it.

Step 2. Decide on your "view parameters", which describe the location of your camera and film. We assume that the camera is always pointed toward the origin of the world coordinates, and that the camera cannot be tilted. The view parameters are:

X-Angle (XA): the angle between the view and the x-axis. For example, an x-angle of 0 degrees would result in a 2-D representation of the x-axis as a line pointing straight down from the origin. A positive X-angle would show it as a vector pointing down and to the left.

Z-angle (ZA): the angle between the line-of-sight and the Z-axis.

Distance(CD): the distance from the camera to the origin.

Screen-distance (SD). The distance from camera to the 2-D screen onto which the 3-D world is projected. It is assumed that the "screen" is a plane perpendicular to the line of sight, located just in front of the camera. A value of 1 usually works fine.

Step 3: Transform each of the 3-D points (x,y,z) in real-world coordinates into 2-D points (screen_x, screen_y), using the following program statements:

```
newZ =-cos(XA)*sin(ZA)*x -
       sin(XA)*sin(ZA)*y-cos(ZA)*z + CD;
       * NewZ is a temporary variable
Screen_x = (-sin(XA)*x +cos(XA)*y) /
       (NewZ/SD);
Screen_y = (-cos(XA)*cos(ZA)*x -
       sin(XA)*cos(ZA)*y+sin(ZA)*z )/
       (NewZ/SD);
```

For increased efficiency, the coefficients can be calculated ahead of time. For a thorough and more analytical presentation of this material, please refer to the book "Fundamentals of Three-Dimensional Computer Graphics", by Alan Watt.

Step 4: Display the 2-D points on a 2-D medium using a scaling factor. The 2-D coordinates coming from the previous step have the same scale as the original 3-D world coordinates. So their scaling needs to be adjusted to fit the scale (inches, pixels or whatever) of the display medium. SAS® does this automatically when you use PROC GPLOT. In addition, there may be points outside the range that you want to display. These must be eliminated. But suppose what you plan to do with a point is use it as one end of a line segment, or as a vertex of a polygon. Generally speaking, you would have to carefully "clip" these objects to fit into your display space, turning a long line, for instance, into a shorter one ending at the edge of your window. Once again SAS® makes this easy, automatically clipping line segments and filled areas to the edges of your window.

Step 5: Adjust parameters until the scene looks right. Whenever you modify the camera distance, the image will grow larger or smaller, which you can compensate for by modifying the scaling factor used in Step 4. When you increase the camera distance there will be less distortion due to the perspective projection, and the result will show parallel lines closer to parallel. On the other hand, a too-distant perspective results in a loss of the 3-D effect. So we experiment to find the happy medium.

These and other more complex 3-D tricks, plus animation of the resulting scenes, are available using the SAS® NVISION software.

OTHER APPLICATIONS

The puzzle-solving algorithm has been successfully adapted to the solution of other puzzles as well. One is a flat triangular puzzle in which the patterns on the edges of neighboring pieces have to match up, much like a jigsaw puzzle. Another is a 3-D puzzle with 25 pieces each made up of four spheres stuck together in all possible ways, fitting into a hollow pyramid shape. Currently the author is working a puzzle which may win him a free para-gliding trip to New Zealand. With or without the laptop, that is the question.

CONCLUSION

The Data Step Graphics Interface opens up a world of possibilities for graphical presentation of information.

The application of SAS® to the solution of puzzles provides hours of entertainment, as well as the satisfaction of seeing solutions emerge which otherwise would defy the efforts of the human puzzle-solver.

Ted Clay
Clay Software and Statistics
168 Meade St.
Ashland, OR 97520
e-mail: clay@mind.net