**Bad Naming Example**

Description

The purpose of the function below is to make a "caricature" of the polyline. It takes as input the indices of the endpoints of the polyline that is to be considered and the maximum deviation that, if exceeded by a given vertex off the straight line between the endpoints of the polyline, will result in marking the point as one to keep for the caricature. This function is recursive.

## What is a [Polyline "Caricature](http://www.codeproject.com/Articles/114797/Polyline-Simplification)"?

## The Algorithm (Douglas - Peucker)

1. Mark the endpoints of the polyline to keep
2. Find the vertex with the greatest perpendicular distance from the line formed by the endpoints
3. if the distance is greater than the prescribed threshold then recursively process the two polylines formed by using the new vertex as endpoint

## Note

The comments before the block of code do NOT indicate the intent of the function. In fact, these comments should be placed elsewhere in the code.

/\*--------------------------------------------------------------------------\*/

/\* angle1 is angle formed between endpoints \*/

/\* angle2 is angle formed between p0 and pi \*/

/\* position is index of point with largest deviation from line connecting endpoints \*/

/\*--------------------------------------------------------------------------\*/

void thinData(coords,stat\_ptr,anchor,float\_vrt,tolerance)

int far\* coords;

char far\* stat\_ptr;

int anchor;

int float\_vrt;

int tolerance;

{

float angle1,angle2;

int dx,dz;

int x1,z1;

int x2,z2;

int i,position;

float dist,deviation;

char t;

int\* t\_ptr;

float max\_dist;

 max\_dist = 0;

// printf("thinData(%i,%i,%i)\n",anchor,float\_vrt,tolerance);

/\* mark endpoints as points to keep \*/

 t = \*(stat\_ptr + anchor);

 \*(stat\_ptr + anchor) = t|0x80;

 t = \*(stat\_ptr + float\_vrt);

 \*(stat\_ptr + float\_vrt) = t|0x80;

 if ((float\_vrt - anchor) < 2)

 return;

 t\_ptr = coords + 2\*anchor;

 x1 = \*t\_ptr;

 t\_ptr++;

 z1 = \*t\_ptr;

 t\_ptr = coords + 2\*float\_vrt;

 x2 = \*t\_ptr;

 t\_ptr++;

 z2 = \*t\_ptr;

//printf("%i,%i,%i,%i\n",x1,z1,x2,z2);

 dx = x2-x1;

 dz = z2-z1;

 if (dx != 0)

 angle1 = atan2(dz,dx);

 if (errno == EDOM)

 {

 printf("%i,%i\n",anchor,float\_vrt);

 printf("%i,%i,%i,%i\n",x1,z1,x2,z2);

 errno = 0;

 }

 t\_ptr = coords + 2\*anchor;

 for (i = anchor+1; i < float\_vrt; i++)

 {

 x2 = \*t\_ptr;

 t\_ptr++;

 z2 = \*t\_ptr;

 t\_ptr++;

 dx = x2-x1;

 dz = z2-z1;

 dist = sqrt(pow(dx,2) + pow(dz,2));

 if (dist != 0)

 {

 angle2 = atan2(dz,dx);

 deviation = dist\*sin(angle2 - angle1);

 }

 else

 deviation = 0; // submitted point coincides with anchor point

 if (deviation < 0)

 deviation = -deviation;

 if (deviation > max\_dist)

 {

 position = i;

 max\_dist = deviation;

 }

 }

 if (max\_dist > tolerance)

 {

 thinData(coords,stat\_ptr,anchor,position,tolerance);

 thinData(coords,stat\_ptr,position,float\_vrt,tolerance);

 }

}

# Naming

## Function Name

"thinData" is not very descriptive at all. Using the method of naming functions of "strong verb, object" we can call this "simplifyPolyline" and understand this much better.

## Input Parameters

In addition to the poor commenting, you should note the choice of variable names.

Let's focus our attention first on the input parameters.

* **"coords"** - this seems tolerable.
* **"stat\_ptr"** - while it IS a pointer, there is no particular meaning conveyed by "stat". "stat\_ptr" points to a list of status bytes, each one corresponding to the status of each coordinate pair. They are marked by setting the highest bit to indicate that the coordinates to which they correspond are to be retained to form the caricature polyline.
* **"anchor"** - this is the index of the first vertex in the polyline. "startIndex" would be better.
* **"float\_vrt"** - the name implies that this is either a floating point value (which it clearly is not) or somehow floats (which it does not). It is really the index of the end of the polyline. "endIndex" might be a better choice.
* **"tolerance"** - this is the perpendicular distance from the line connecting "anchor" and "float\_vrt". "max\_deviation" might be a better name but we could tolerate "tolerance". (see what I did there?)

## Local Variables

* **"position"** is the index of the current set of coordinates being considered. "splitIndex" would be a better choice.
* **"t"** - is the status code that is being modified. Since it goes along with "stat\_ptr" and we have decided it is a status indicator, "status" would be better.
* **"t\_ptr"** - implies or hints that it is related to "t" which it is not. Instead, it is a temporary pointer to a coordinate. It is used to extract individual values in the list. Beyond the criticism that we could have used a pointer to a struct to extract these values, the choice of names is awful because not only does the name not indicate its use, it implies something completely different.

## Other Criticisms

### **Bad Formatting**

Well, we don't have to say much about this. There is not enough whitespace being used.

### **Unnecessary Operations?**

1. Immediately upon entering this function, the programmer retrieves a byte located in the list of status bytes and uses a logical "OR" operation to modify the highest bit. This would be fine if we cared about the lower part of the byte. We might. However, nothing indicates that this is the case. We simply do not know.
2. "if ((dx != 0) || (dx != 0))" - this is redundant. The idea here is to trap a condition where "atan" is undefined. This is not explained and relies on the reader knowing what the intention is.
3. The code would be cleaner if macros or helper functions had been used:
	1. calculating the angle between the x-axis and the line defined by two points
	2. calculating the perpendicular distance of a given point relative to the line defined by endpoints
4. the perpendicular distance calculation could be skipped until the largest angle relative to the endpoints was found

### **Performance**

The calculations here make two noteworthy bad choices for performance. These both occur in the distance calculation.

1. use of "pow(dx,2)" instead of just using (dx\*dx) which would execute faster. Modern compilers MIGHT automatically fix this.
2. use of "sqrt" is also relatively expensive and definitely completely unnecessary. If we compare the distance to the square of the tolerance then the comparison would still be valid. The thing to watch out for on something like this is the risk of overflowing the dynamic range of your variable.
3. the unnecessary calculation of the perpendicular distance mentioned above

### **Picky C-language Things**

|  |
| --- |
| // a struct for each of the vertices in the string of vertices would look like thisstruct vertex{ int x; int y;}; |

Here is an improved rewrite to this code. I've added some comments, improved naming and simplified.

//====================================================================================================

// The purpose of the function below is to make a "caricature" of the polyline.

// It takes as input the indices of the endpoints of the polyline that is to be considered

// and the maximum deviation that,

// if exceeded by a given vertex off the straight line between the endpoints of the polyline,

// will result in marking the point as one to keep for the caricature.

// This function is recursive.

//====================================================================================================

void simplifyPolyline ( vertex\* vrtPtr,

 char\* status\_ptr,

 int startIndex,

 int endIndex,

 int maxDeviation)

{

int dx,dz;

int x1,z1;

int x2,z2;

int i;

int splitIndex; // splitIndex is the index of the vertex that is currently best candidate for inclusion

float angle1,angle2; // angle1 is angle formed between endpoints, angle2 is angle formed between p(0) and p(i)

float distance;

float deviation;

float maxDistance;

 maxDistance = 0;

 // mark endpoints as points to keep

 \*(statusPtr+startIndex) |= 0x80;

 \*(statusPtr+endIndex) |= 0x80;

 // if the two points are neighbors then we're done

 if ((endIndex - startIndex) < 2)

 return;

 x1 = (vrtPtr + startIndex)->x;

 z1 = (vrtPtr + startIndex)->z;

 x2 = (vrtPtr + endIndex)->x;

 z2 = (vrtPtr + endIndex)->z;

 dx = x2-x1;

 dz = z2-z1;

 if (dx != 0)

 angle1 = atan2(dz,dx);

 else

 angle1 = PI/2;

 if (errno == EDOM)

 {

 printf("%i,%i\n",anchor,float\_vrt);

 printf("%i,%i,%i,%i\n",x1,z1,x2,z2);

 errno = 0;

 }

 // find the greatest deviation distance from the straight line

 for (i = startIndex+1; i < endIndex; i++)

 {

 x2 = (vrtPtr + i)->x;

 z2 = (vrtPtr + i)->z;

 dx = x2-x1;

 dz = z2-z1;

 distance = dx\*dx + dz\*dz;;

 if (distance != 0)

 {

 angle2 = atan2(dz,dx);

 deviation = distance\*sin(angle2 - angle1);

 }

 else

 deviation = 0; // submitted point coincides with anchor point

 if (deviation < 0)

 deviation = -deviation;

 if (deviation > maxDistance)

 {

 splitIndex = i;

 maxDistance = deviation;

 }

 }

 // send the polylines bounded by startIndex, splitIndex and splitIndex, endIndex through the same process.

 if (maxDistance > maxDeviation)

 {

 simplifyPolyline(vrtPtr, statusPtr, startIndex, splitIndex, maxTolerance);

 simplifyPolyline(vrtPtr, statusPtr, splitIndex, endIndex, maxTolerance);

 }

 return;

}