DGBZ

Dave G's Bezier Curve System 0.95 for POV-Ray 3.0

David Govoni
david.govoni@sympatico.ca
www.geocities.com/SiliconValley/Vista/3378

A system for the easy creation of bezier curves and paths within POV-Ray scenes without external programs.

Table of Contents

Introduction and Overview

Welcome to Dave G's Bezier Curve System for POV-Ray (DGBZ). The system is made up of two parts, a include file for single bezier curves (DGBZ.INC) and an include file to handle multiple bezier curves linked together (DGMBZ.INC). There are no stand alone programs. All the calculation is done by POV-Ray's parser, making this a cross platform solution. It should work on every compile of POV-Ray.

What is it?

The bezier curve system is designed to add a great deal of functionality to POV-Ray. It adds the following features:

-Simple and quick calculation of points along a Bezier Curve. Using the Bernstein basis functions.

-Slow, but more accurate, calculation of a point a particular distance along the curve. This is used to step along a bezier curve at a fixed rate.

-Creation of complex paths of multiple bezier curves and can calculate the current position along the entire path. Paths can be made of up to 99 bezier curves. Therefore you can move objects or the camera along complicated winding paths. The camera can be made to move at a fixed velocity, no matter how big or small the pieces of the path are.

-You can also use bezier curves to produce smoothly accelerating motion along paths and other curves (see the Hints and Tricks section).

How Does It Work?

Simply put you set DGBZ variables to the settings for you curve and then include the DGBZ.INC file (or DGMBZ.INC for a multiple curve path). If you need another curve, you just re-declare the variables and include the file again. You can have as many bezier curves in your file as you wish.

All the math is actually done by the POV-Ray parser as it reads the pov file. This way you can use bezier curves within an animation with no difficulty. For each frame POV-Ray recalculates all the curves in the file. Therefor you can easily use the curves for animation.

Who Is It Meant For?

DGBZ is meant for two radically different audiences.

First it is meant for the individuals who 'hand roll' pov files. The syntax is meant to be easy enough to be entered as you type. You command DGBZ using text strings and declaring variables as numbers and vectors as appropriate. All the tutorial files and demonstrations were constructed this way.

However, DGBZ was also meant to be used by programs that generate pov files. It is meant to be used by the modellers and scene builders that generate pov output. It allows a simple way for curves and paths to be added to pov scenes. It allows for the simple creation of motion throughout an animation without creating a pov file for each frame. In conjunction with the other Dave G's files it is designed for computers to use, as well as people.

Tutorial

Scene 1 - Traditional Bezier Curves

The easiest way to see DGBZ in action is to make it do some work. Enter the following as a pov file, (or use BZ01.POV):

```
#include "colors.inc"

camera { location <0, 0, -10> look_at <0, 0, 0> }

light_source { <20, 20, -20> color White }

#declare Curve1C0 = <-6, -6, 4>
#declare Curve1C1 = <-6,  6, 4>
#declare Curve1C2 = <-1, -6, 4>
#declare Curve1C3 = <-1,  6, 4>

sphere { Curve1C0, .3 pigment { color Green } }
sphere { Curve1C1, .2 pigment { color Blue  } }
sphere { Curve1C2, .2 pigment { color Blue  } }
sphere { Curve1C3, .3 pigment { color Green } }
cylinder { Curve1C0, Curve1C1, 0.05 pigment { color Yellow } }
cylinder { Curve1C2, Curve1C3, 0.05 pigment { color Yellow } }

#declare Looper = 20

#declare DGBZCommand = "RESET"
#include "DGBZ.INC"

#declare DGBZC0        = Curve1C0
#declare DGBZC1        = Curve1C1
#declare DGBZC2        = Curve1C2
#declare DGBZC3        = Curve1C3
#declare DGBZCommand   = "POSITION"

#declare Count = 0
#while (Count <= Looper)
      #declare DGBZAmount = Count / Looper
      #include "DGBZ.INC"
      sphere { DGBZPosition, 0.2 pigment { color Red } }
      #declare Count = Count + 1
#end
```

Reading from the top, the file starts by adding a simple camera and light. It then declares four points in space. These are the traditional points of a bezier curve. A bezier curve starts at point 0

and ends at point 3. In between it is controlled by the other two points. The final curve will start at point 0 moving towards point 1, and when it ends at point 3 it will be moving away from point 2. Then these four points are shown on the screen as spheres. The two end points are green and the control points are blue. Yellow lines connect each end sphere with the nearest control point.

Further down the file a variable, DGBZCommand, is set to "RESET" and the DGBZ.INC file is included. This command is interpreted by the DGBZ system and all the variables used by the system are set to their default values.

Then a group of DGBZ variables are set. DGBZC0 through DGBZC3 are the four points of the curve. The four previously declared points are used to set DGBZ's points. (The DGBZCx variables could have been set directly as in DGBZC0 = <-6,-6,4>).

DGBZCommand is assigned a value of "POSITION". This is string will command DGBZ to find a position along the curve

Next a simple loop is set up to loop 20 times. During each loop the value of DGBZAmount is set to a value. DGBZAmount ranges from 0 to 1. This is the how far the position along the curve you wish to calculate is located.

Then the DGBZ.INC file is included. DGBZ interprets the command and calculates the position in space along the curve. The position is returned in DGBZPosition. A small red sphere is then placed at this point in the scene.

After 20 loops there are 20 spheres placed along the curve.

If you render this file you will see the control points and the points along the curve filling the left side of the image.

This is a standard looking bezier curve.

Scene 2 - A More Perfect Curve

If you look at the curve drawn in the previous tutorial you might notice something strange. Most of the red spheres are concentrated along the middle of the curve.

The typical method of calculating positions along a bezier curve, the Bernstein Basis Functions, does not give equally spaced points. As DGBZAmount varies from 0 to 1 the functions calculate how much influence each of the four control points has on the final position. In most cases this does not result in equal spacing.

For some uses of curves this is just fine. However if you plan to move an object (or the camera) along a bezier curve it will not move at a constant speed. Where the points are far apart the object will move quickly. Where the points are close together the object will slow down.

DGBZ does give you a way to avoid this problem.

To the top section of the pov file from scene 1 add the following definitions: (or use BZ02.POV

```
#declare Curve2C0 = < 6, -6, 4>
#declare Curve2C1 = < 6,  6, 4>
#declare Curve2C2 = < 1, -6, 4>
#declare Curve2C3 = < 1,  6, 4>

sphere { Curve2C0, .3 pigment { color Green } }
sphere { Curve2C1, .2 pigment { color Blue  } }
sphere { Curve2C2, .2 pigment { color Blue  } }
sphere { Curve2C3, .3 pigment { color Green } }
cylinder { Curve2C0, Curve2C1, 0.05 pigment { color Yellow } }
cylinder { Curve2C2, Curve2C3, 0.05 pigment { color Yellow } }
```

And below the loop add the following code:

```
#declare DGBZC0        = Curve2C0
#declare DGBZC1        = Curve2C1
#declare DGBZC2        = Curve2C2
#declare DGBZC3        = Curve2C3
#declare DGBZCommand   = "POSITION"
#declare DGBZCalc_Type = "EXACT"

#declare Count = 0
#while (Count <= Looper)
      #declare DGBZAmount = Count / Looper
      #include "DGBZ.INC"
      sphere { DGBZPosition, 0.2 pigment { color Red } }
      #declare Count = Count + 1
#end
```

The new lines at the top add another set of points. This time for a mirror image curve on the right hand side of the image. Then spheres and lines are placed to show the curves points.

The code at the bottom assigns DGBZ variables with the second curve’s points and then loops through and places red spheres along the new curve.

The only addition is the assignment of “EXACT” to DGBZCalc_Type.

Now render the scene (This may take a while to parse, please be patient!!) If you view the results you will see two curves. On the left is the curve from scene 1, and on the right is a mirror twin curve.

The right hand curve has red spheres spaced out evenly along the length of the entire curve.

DGBZ performs this trick by making POV-Ray do a great deal of math. Since all the work is being done by the POV-Ray parser (which is an interpreter) the results are not very fast. On my Pentium 100 using POV-Ray for windows parsing the scene takes almost 90 seconds.

But, the results are worth it. The curve on the right can be used to move objects at a constant velocity.

DGBZ performs this trick by brute force. It simply calculates an approximate length for the curve by incrementing through the curve using quick method. By default DGBZ steps through the curve in 1,000 increments. (You can set this value higher or lower, as needed) Then, knowing the length of the curve, and the amount you want to travel, DGBZ re-increments through the curve until it finds the final position.

In this image there are 20 points along the curve. For each point DGBZ finds the curves length and then re-increments through the curve. Over all the 20 spheres DGBZ has to re-increment, on average, halfway through the curve. This gives a total of 1500 quick bezier calculations per sphere, and a total of 30,000 calculations for all the points.

That POV-Rays' interpreter takes only 90 seconds on a Pentium 100 is impressive. I don't think the POV-Ray team spent too much time optimizing the parser for speed. I would have spent the time optimizing the renderer myself!

Also, keep in mind that unless you are using the curve to produce multiple pointsto create a more complex object (as in this example) you normally will only calculate one position for each curve. If you where animating a pov scene then for each object moving a single calculation would be performed every frame. Then the overhead required is less of a problem.

Scene 3 - Two Curves Strung Together Make a Path

Now that DGBZ can calculate the length of a curve, DGBZ can calculate the length of a path made up of curves.

And if DGBZ can find the total length of a path, then DGBZ can calculate a point a certain distance along the path.

In other words, since DGBZ can step through a curve in equal increments its only a small step to connect two or more curves and make a path.

So, enter the following pov file (BZ03.POV):

```
#include "colors.inc"
camera { location <0, 3, -10> look_at <0, 3, 0> }
light_source { <20, 20, -20> color White }

#declare DGMBZCurves = 3

#declare DGMBZ01C0 = < 0, 6, 1>
#declare DGMBZ01C1 = < 2, 6, 1>
#declare DGMBZ01C2 = < 5, 2, 1>
#declare DGMBZ01C3 = < 4, 0, 1>

#declare DGMBZ02C0 = < 4, 0, 1>
#declare DGMBZ02C1 = < 3,-2, 1>
#declare DGMBZ02C2 = <-3,-2, 1>
#declare DGMBZ02C3 = <-4, 0, 1>

#declare DGMBZ03C0 = <-4, 0,1>
#declare DGMBZ03C1 = <-5, 2, 1>
#declare DGMBZ03C2 = <-2, 6, 1>
#declare DGMBZ03C3 = < 0, 6, 1>

sphere { DGMBZ01C0, 0.2 pigment { color Green } }
sphere { DGMBZ01C1, 0.2 pigment { color Blue } }
sphere { DGMBZ01C2, 0.2 pigment { color Blue } }
sphere { DGMBZ01C3, 0.2 pigment { color Green } }
sphere { DGMBZ02C0, 0.2 pigment { color Green } }
sphere { DGMBZ02C1, 0.2 pigment { color Blue } }
sphere { DGMBZ02C2, 0.2 pigment { color Blue } }
sphere { DGMBZ02C3, 0.2 pigment { color Green } }
sphere { DGMBZ03C0, 0.2 pigment { color Green } }
sphere { DGMBZ03C1, 0.2 pigment { color Blue } }
sphere { DGMBZ03C2, 0.2 pigment { color Blue } }
sphere { DGMBZ03C3, 0.2 pigment { color Green } }

#declare Looper = 20

#declare Count = 0
#while (Count <= Looper)
      #declare DGMBZAmount = Count / Looper
      #include "DGMBZ.INC"
      sphere { DGMBZPosition, 0.1 pigment { color Red } }
      #declare Count = Count + 1
#end
```

If you look at the resulting image you will see a path nicely traced out by little red spheres (and there are no missing spheres, they just are hidden inside the green ones).

Looking at the code you will notice that all the variable start with DGMBZ. To calculate paths you use the DGMBZ.INC file and use the DGMBZ variables.

The code starts with the DGMBZCurves variable. This tells DGMBZ how many curves are in the path. In this case it is 3, but it can be as high as 99.

Then the curves are defined. DGMBZ01C0 is the first point in the first curve. DGMBZ03C3 is the last point in the third curve. After the definitions, spheres are placed on the control points, just as in the earlier scene. This isn't necessary for the calculations, it is only meant to show where the path travels.

There is a simple loop that places points along the path by assigning a value to the DGMBZAmount variable and, after including DGMBZ.INC, using the DGMBZPosition variable to place a red sphere.

Again this takes a long time to parse. A single point may only take a small number of seconds, but a series of points such as these can take a long time.

And that's the short tutorial for DGBZ and DGMBZ. Using it should be simple. I didn't try and make it difficult to use. I did try and make it very useful. A few uses are explained in the Hints and Tricks section. But the best tip I can give is you don't have to use a curve to move on object. There are any number of other uses.

You can move lights, the camera, or even textures through an object. You don't have to use the curves to produce a single point. If you want to you can use the curves to generate points for the creation of some amazing objects.

And, if you come up with any techniques that produce results I'd love to hear about it. I'm learning how to use this system myself.

Good luck rendering.

Reference

A little note about the variables used by DGBZ and DGMBZ. All start with DGBZ or DGMBZ and then have properly capitalized names. This may seem like extra typing but it was done to avoid conflict with other POV-Ray files and objects.

All variable names in the Dave G's series start with "DG" and then more letters to indicate which part of the Dave G's System it is part of. All internal variables in these files follow this format as well.

Unless you happened to name your variables with these letters (which I hope is highly unlikely) there should be no conflict between the Dave G's series and any other POV-Ray files.

DGBZ Variables

These are variables you can set before you include DGBZ.INC in your file. DGBZ does not alter these variables as it calculates. You can re-include DGBZ.INC and all the variables will be the same.

In the tutorial files all of the variables that needed to be set are set once. Then only the variable that changes (in that case DGBZAmount) is set again before each call to DGBZ.

DGBZCommand - string

This is literally the variable that tells DGBZ what to do. There are 4 main options and many sub-options.

1.) "RESET"

When DGBZ receives this command it simply resets all variables to their default values. While there are not too many variables now, this is also reserved for the future when I may make DGBZ more capable.

2) "POSITION"

Given the four points of a curve (see DGBZCx) and an amount (DGBZAmount) this command instructs DGBZ to calculate a position in space. DGBZ will either calculate a quick position along the curve, or a more 'correct' position a certain distance along the curve's length, depending on the DGBZCalc_Type variable. The position is returned in DGBZPosition.

3) “LENGTH”

Given this command DGBZ will calculate the length of the current curve. It returns the length in DGBZLength. This can be used to calculate multiple exact points along a single curve without the need for DGBZ to recalculate the length of the curve for each point.

4) “DISTANCE”

For this command DGBZ will step through a curve and find a position DGBZDistance through the curve. If you let DGBZ calculate the length of a curve (see LENGTH above) you can then pass distances directly to DGBZ. Then DGBZ can calculate a point without calculating the length of the curve first.

5) “TRANS_xxx”

For now, better description of the TRANS commands is included in the Hints and Tricks section.

In a nutshell TRANS_001 through TRANS_034 give different transition curves that can be used to smoothly accelerate and decelerate particles along different paths.

Given an amount (DGBZAmount) through the transition and a transition number (TRANS_001 through TRANS_034 in the DGBZCommand variable) DGBZ will return a value in DGBZTransition.

DGBZTransition usually ranges from 0 to 1, indicating the amount of transition that has occured. However it can range below 0 and over 1 depending on the transition chosen. This can simulate recoil before a motion or the overshooting of a final resting point.

DGBZCalc_Type - string

You can set this to two values - “EXACT” and “APPROXIMATE”. The default value is “APPROXIMATE”. This determines which method DGBZ uses to locate a point on a curve. See the tutorial section for an example.

If you use “APPROXIMATE” DGBZ will calculate a point on the curve based on the Bernstein Basis Functions. Given “EXACT” it will calculate a point a correct amount along the length of the curve.

DGBZC0, DGBZC1, DGBZC2, DGBZC3 - vector

These are the four points of the bezier curve DGBZ will calculate. The curve will start at DGBZC0 and end at DGBZC3. The two middle points control the bend of the curve.

Simply put the curve will start from DGBZC0 moving towards DGBZC1 and will end at DGBZC3 coming from DGBZC2.

DGBZAmount - float

DGBZAmount is used with the DGBZCommand “POSITION”

This is the proportion through the curve of the point to be calculated. DGBZAmount ranges from 0 to 1. If the value falls outside 0 to 1 DGBZ will ‘correct’ the value and continue. It is meaningless, in a bezier curve, to calculate a position before or after the curve.

DGBZDistance - float

DGBZDistance is used with the DGBZCommand “DISTANCE”.

DGBZDistance stores the length of the curve you want to be traversed before the point you are looking for. If the curve is shorter than the distance you specify DGBZ will return the final point on the curve.

DGBZSteps - float

DGBZSteps is used by DGBZ when calculating an exact length of, or distance along a curve.

It represents the number of increments used by DGBZ to find the length of the curve, as well as the number of increments used to find the position of a point.

DGBZSteps defaults at a value of 1,000. You can set this to any positive number.

The more steps used by DGBZ the ‘more correct’ the length of the curve can be calculated. As well with more steps there is a finer control of the calculated point. If, in the course of animating a scene, you are going to calculate a certain number of different points along a curve, DGBZSteps should be a much higher number to give accurate positions.

If you leave DGBZSteps at 1,000 and render an animation with a curve that will be recalculated with 2,000 different values then the motion along the curve will be jumpy and imprecise. If DGBZSteps is set to 20,000 then the movement along the curve will be much smoother. However the calculation time will also increase.

DGBZ Returned Variables

After you include DGBZ.INC it returns the results of its calculations in these variables.

DGBZPosition - vector

This is the point in space that results from either a “POSITION” or a “DISTANCE” command. It is a point along the bezier curve just calculated.

DGBZLength - float

When DGBZ finds the length of a curve for the “LENGTH” command it returns the value in DGBZLength.

DGBZTransition - float

When DGBZ calculates an accelerating or decelerating transition using the “TRANS_xxx” commands it returns the amount of the transition in DGBZTransition.

DGBZTransition usually ranges from 0 to 1, but it can fall below zero and rise above 1. All transitions start at 0 and end at 1. If you need to start at 1 and end at 0 you can simple subtract DGBZTransition from 1.

DGMBZ Variables

The DGMBZ multiple curve system has been deliberately simplified. There are no complex options. It simply takes in a path and returns a value. To enter a path you use the following variables.

DGMBZCurves - float

This is the number of curves in the path. It can be up to 99. For most applications this limit will not be reached.

DGMBZxxC0, DGMBZxxC1, DGMBZxxC2, DGMBZxxC3 - vector

For each curve that DGMBZ will process as part of the path it needs to know the location of the four points of the curve. These points are stored in these variables. For example:

DGBZ01C0 - the first point of the first curve
DGBZ99C3 - the final point of the 99th curve.

If you do not assign a value to one of these variables DGMBZ will assign a default value. This is so you do not have to enter 99 curves if you are only using the first 3.

DGMBZAmount - float

The amount, from 0 to 1, through the path that the resulting point will be located. When DGMBZAmount is 0 the first point of the first path is returned. When it reaches 1 the final point of the final path is returned (which path is the final one is determined by DGMBZCurves).

DGBZSteps - float

Whether for multiple curves in a path or a single curve, DGBZSteps is used to pass along the number of increments used to calculate the length of a curve. If you need finer steps or detail you can set DGBZSteps to a higher value. DGBZSteps defaults to 1,000.

DGMBZ Returned Variables

DGMBZPosition - vector

This is the position of the point DGMBZ calculated along the path.

Hints and Tricks

Smoothly Connecting Curves In A Path

Unless you know a simple trick you may find that connecting multiple curves smoothly is difficult. If you don't connect curves smoothly then the motion along the curve jumps in another direction as the transition from one curve to another is made.

To smoothly connect curves you must do two things.

1) The ending point of the first curve (C3) must be the starting point of the next one (C0).

2) The common point must fall on a line drawn between the last control point of the first curve (C2) and the first control point of the second curve (C1). In other words, the joining point must be on a line drawn between the two control points. When the points form a line the path will smoothly transition between the two curves.

Don't Connect Curves In A Path

I don't force the curves in a path to be connected. I don't make the final point of one curve the first point of the next curve. So you can take several different curves and make them part of one continuous path. DGMBZ will gladly compute which curve the path is on, and at which point along the curve the path is on.

You can use this to have actions occur directly after one another all over the scene. (For example as soon as an object disappears into another, it can re-appear across the scene.)

Unlike the paths in some animation programs you are not required to create a continuous one.

Using A Path As Motion Control

This may be the most interesting use of bezier curves. And one of the simplest.

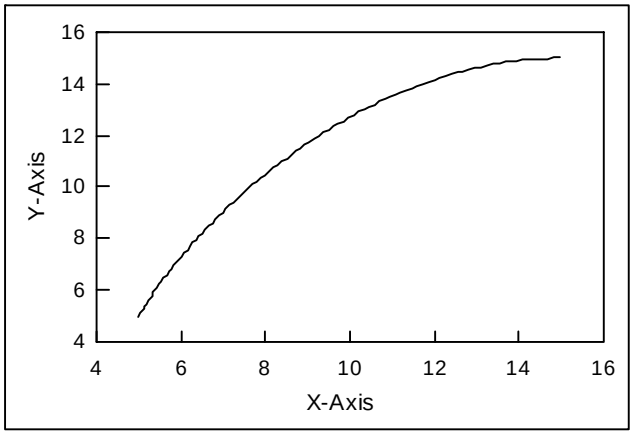
When I created the Dave G's series (Particle Animation, TimeLine, and Bezier Curve Systems) I wanted to be able to accomplish most, if not all, of the tasks that could be accomplished by a separate stand alone program. However I wanted to do it all within POV-Ray (or more precisely POV-Ray's parser,since I didn't want to have to patch the source code).

Most of the tasks are accomplished directly by the include files. But one large exception remained.

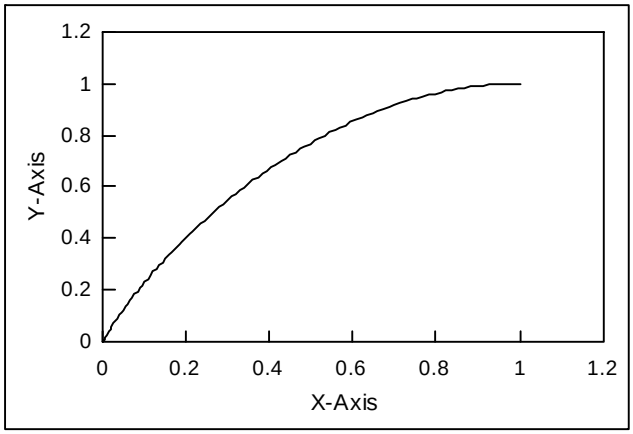
I wanted to be able to smoothly accelerate and decelerate an object along a path. I wanted to be able to modify the amount of acceleration. I wanted to be able to have the object anticipate its motion by backtracking momentarily or overshoot its ending point and settle back down. Having convincing motion in an animation was my goal.

I tried to come up with many complicated ways of doing it. And then it struck me.

I was staring at bezier curves while debugging the DGBZ.INC file and it struck me. Too see my inspiration look at the following 2 dimensional bezier curve.

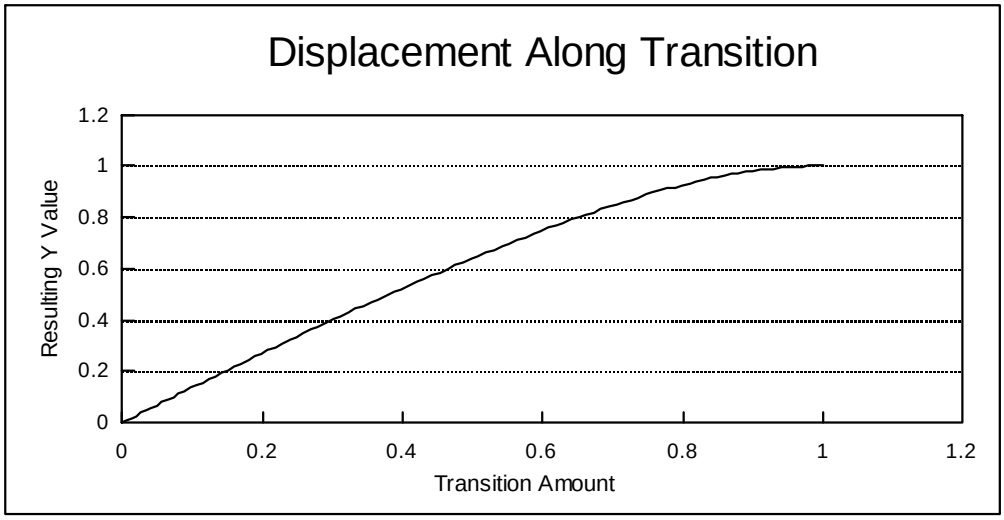


Ignore the arc of the curve for a moment. Instead look at the starting and ending points. Picture the curve positioned in space so the starting point has a Y value (vertical displacement) of 0. Scale the curve so the end point has a Y value (vertical displacement) of 1.



Now imagine yourself being a point in space travelling along the curve. As you travel your co-ordinates change. In particular, your Y value will change from 0 at the beginning to 1 at the end. However the transition will not be linear. At first you will move quickly upwards away from zero. Then the rate of change will slow and you will gradually approach Y=1. In other words travelling along the curve gives a transition from 0 to 1 that smoothly decelerates along the way.

If I step along the curve, using the EXACT calculation method and plot the resulting Y values against the percentage of the distance through the curve, I end up with:



In other words I can create a transition that is not linear, and yet I can easily calculate it using the DGBZ system I already have.

If I want to know what my Y value is one quarter of the way through this transition I simply take the original bezier curve (properly translated and scaled) and I ask for the EXACT position at a value of 0.25. Then I simply take the resulting DGBZPosition vector and pull the Y value. Its as simple as:

```
#declare NewValue = DGBZPosition.y
```

If I want to have a different transition from 0 to 1 I simply modify the two control points of the curve. Each new position for the control points gives another bezier curve, which results in another transition between 0 and 1.

To make my life, and yours, easier I have collected a number of interesting transitions. I have taken these and made them an integral part of the DGBZ.INC file. Simply, each transition can be called directly with a simple DGBZCommand. Then the resulting Y value can be found in the DGBZTransition variable. You don't have to declare the curve, just the position along the final transition.

So far there are a 34 curves (TRANS_001 through TRANS_034). Description aren't much use, so I'm assembling a gallery of them. But until that’s ready all I can suggest is to experiment. To see a transition curve being used, look at the following pov file fragment:

```
#declare DGBZAmount = 0.5 // halfway through the transition
#declare DGBZCommand = “TRANS_022" // the twenty second pre-defined curve
#include “DGBZ.INC”
```

Then a value is returned in DGBZTransition.

Until I make better documentation of the pre-built curves have fun experimenting!