DGTL

Dave G's TimeLine 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 a TimeLine within POV-Ray animations without external programs or multiple source files.

Table of Contents

Introduction and Overview

Welcome to Dave G's TimeLine System (DGTL). DGTL is an include file for POV-Ray which allows for an animation TimeLine within a single POV render. Since it is not a separate program, or a stand alone executable, it should be portable across all POV-Ray platforms.

While I attempt to explain DGTL here, it is much easier to work through the tutorial (or at least follow along) so I can introduce the features of the system one at a time.

What Does It Do?

In a nutshell, DGTL breaks an animation into smaller chunks. Each chunk, or segment, is assigned a duration. The segments are proportional to each other. A segment with twice the duration will get twice as many frames in the final render. And the TimeLine scales as you increase the number of frames you render with POV-Ray. If you render the file with twice as many frames each segment takes twice as long as it did previously.

A TimeLine is made of up to 99 segments. As POV-Ray renders individual frames of the final render it determines which segment each frame is in. It also determines how far through the current segment the current frame is. Within each segment you can animate as you normally would within an entire POV-Ray animation. This allows you to break a longer animation into manageable chunks automatically.

In a simple POV-Ray animation you utilize the clock variable to determine how far through the animation you are. You can then do almost anything to the scene as the value of clock changes from 0 to 1. With DGTL you are told which segment the animation is currently in, as well as being told how far through the current segment you are. The DGTLCurrent_Segment variable returns the number of the current segment (from 1 to 99). You end up using a DGTLSegment_Clock variable, which varies from 0 to 1 within each segment, to animate within a particular segment.

Who's It For?

Like all of the files in the Dave G's series it is meant for anybody creating pov files. I myself use it to 'hand roll' scenes in my favourite text editor. I manually type the DGTL commands into the pov file and POV-Ray and DGTL do the work of segmenting my animation.

However it is also meant for programs that automatically generate pov files. Usually these are scene builders and modellers. The modeller can let the user define the animation in the scene building program and then translate that into commands in a single pov file with the DGTL commands embedded. POV-Ray will render the scene properly, independent of the number of frames rendered. That way modellers can create complex animations without creating a single pov file for every resulting frame.

This could make it easy for any modeller to add animation capabilities. And this is the capability most often on the wish list of the people creating scene builders. The Dave G's series also allows for paths, curves, and particles to be included within a single pov file (see the other documentation).

This also explains why there are up to 99 segments in a TimeLine. For most of us working with 'hand rolled' pov files dealing with 99 segments may seem slightly absurd. But for a scene builder doing it automatically it can be very simple to use them all.

Tutorial

Scene 1 - A Simple Animation

The easiest way to see DGTL in action is to enter a scene. So type the following into a new pov file (Or load TL01.POV).

```
#include "colors.inc"

camera { location <0, 3, -6> look_at <0, 0, 0> }

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

plane { y, 0 pigment { checker color White color Black } }

#declare DGTLSegment01 = 1
#declare DGTLSegment02 = 1
#declare DGTLSegment03 = 1
#declare DGTLSegment04 = 1

#include "DGTL.INC"

#switch (DGTLCurrent_Segment)
      #case (1)
            sphere { <-2,0.5,-1>, 0.5 pigment { color Green } }
      #break
      #case (2)
            sphere { <-1,0.5,-1>, 0.5 pigment { color Blue } }
      #break
      #case (3)
            sphere { <1,0.5,-1>, 0.5 pigment { color Red } }
      #break
      #case (4)
            sphere { <2,0.5,-1>, 0.5 pigment { color Brown } }
      #break
      #else
            sphere { <0,1,-1>, 1 pigment { color White } }
#end
```

The file starts by including “colors.inc” and setting up a camera, a light, and a checkerboard plane to stage the action. Then there are a few declares. DGTLSegment01 is assigned a value of 1. This is the duration the animation will spend in segment 1. DGTLSegment02 is assigned the length of time the animation will spend in segment 2. And so on....

How many frames will segment 1 last in the final animation? It depends on two things.

First, it depends on the total duration of time spent in all segments. Since there are four segments (01, 02, 03, 04) and they are each given a value of 1, the total duration is 4. Therefore one quarter of the total frames will be spent in each segment.

The number of frames spent in segment 1 depends on the number of frames you render. If you render this file with 16 frames then segment 1 will last 4 frames. If you render 1000 frames then segment 1 will last 250 frames.

In this file four segments are declared. You can declare the time for up to 99 segments. Any segment you do not declare will automatically set themselves to a length of 0.

After you declare all segments you want to be in the animation you include the DGTL.INC file. Then DGTL takes over and calculates which segment the current frame is being rendered is within, and how far through that segment the animation has progressed.

Just after the DGTL.INC file is included this information is used to animate the scene. A switch-case statement is used to alter the scene depending on which segment is current.

If you render the file with POV-Ray and generate a number of frames using the animation feature and then watch the result. You should see four spheres appear, one at a time, from left to right. Each having a different colour. From Green to Blue to Red to Brown.

If you look at the resulting frames you will notice that the animation progresses from segment 1 to segment 2 and then 3 and finally 4. Each of the four spheres appear in order. Just as you might suspect. Two things to notice. In the switch statement the else is used to show a white sphere during all other segments. Since it never appears we know the timeline progressed only through the first 4 segments.

If you alter the scene so the four declarations read:

```
#declare DGTLSegment04 = 0.5
#declare DGTLSegment02 = 0.5
#declare DGTLSegment03 = 0.5
#declare DGTLSegment01 = 0.5
```

And re-render the file the four spheres will change appear in the correct order. You can declare the duration of the segments in any order, they will still occur in numerical order. First segment 1, then segment 2, and so on. And, even though the durations are set to 0.5 instead of 1, the file will render identically. Each segment still takes up a quarter of the total number of frames.

If you alter the scene so the declarations read:

```
#declare DGTLSegment99 = 1
#declare DGTLSegment01 = 1
#declare DGTLSegment02 = 1
#declare DGTLSegment03 = 1
```

And look at the results (forgive me for not typing ‘and then re-render the animation’) you will see the first three spheres appear on queue and then the white sphere will appear as segment99 occurs right after segment 3.

While there are 99 segments available for your TimeLine, all segments default to a duration of 0 unless you specifically declare them otherwise. Therefore segments 4 through 98 have no duration and do not take up any frames during the render.

So far all is working as you might expect. So now, change the four lines again to:

```
#declare DGTLSegment01 = 1
#declare DGTLSegment02 = 2
#declare DGTLSegment03 = 3
#declare DGTLSegment04 = 4
```

In this case the total length of time for all the segments is 10 units (I sometimes refer to it as seconds, but there is no direct connection to playback speed. There is only a connection to the proportion of rendered frames). Segment 1 will last for one tenth of the entire animation. Segment 2 will last the next two tenths, segment 3 for 4 tenths, and segment 4 for four tenths (two fifths).

If you render this for only a few frames (ten or so) it can be difficult to notice the true proportions, but as the number of frames increase the proportions become more and more obvious. Either way DGTL will space out the frames as best as it can.

For the next change, imagine you are working on a large multi-segment animation. Imagine it all works, except for the third segment. You want to concentrate on only the third segment. So change the lines to read:

```
#declare DGTLSegment01 = 0
#declare DGTLSegment02 = 0
#declare DGTLSegment03 = 3
#declare DGTLSegment04 = 0
```

Now the scene renders with only the third segment active. Every frame you render happens during the third segment. This is the perfect way to work on only part of your animation.

Scene 2 - Accessing The Master Clock

Enter the following scene, or load TL02.POV.

```
#include "colors.inc"

camera { location <0, 3, -6> look_at <0, 0, 0> }

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

plane { y, 0 pigment { checker color White color Black } }

#declare DGTLSegment01 = 1
#declare DGTLSegment02 = 2
#declare DGTLSegment03 = 2
#declare DGTLSegment04 = 2

#include "DGTL.INC"

#declare Count = int(DGTLMaster_Clock)

#declare Looper = 0
#while (Looper <= Count)
      sphere { <(Looper - 3),0.5,1>, 0.5 pigment { color Red } }
      #declare Looper = Looper + 1
#end
```

If you run this scene you will see the spheres appear from left to right as the animation continues. Each sphere appears as the DGTLMaster_Clock variable grows.

In a pov file the clock variable ranges from 0 to 1. This is provided by POV-Ray. If you set up a TimeLine the DGTLMaster_Clock variable ranges from 0 to the total length of all the segments in the file. In this case the four segments add up to 7. Therefore the DGTLMaster_Clock variable ranges from 0 to 7. Notice how the loop works though. The loop will display one sphere even if the Count variable is 0. That is why there is always a sphere in the image. Also, if you render this using the Cyclical Animation Option you will end up with 7 spheres. If you don’t use this option you will end up with 8.

This has many possible uses. Add the following to the bottom of the file and render it again. (Or load TL02B.POV)

```
cone {
      <-1, 0, 0>, 0.3
      < 1, 0, 0>, 0.3
      pigment { color Green }
      rotate <0,(180 * DGTLMaster_Clock),0>
      translate <0,2,1>
}
```

The new cone will rotate every 2 DGTLMaster_Clock values. This cone will rotate during all segments of the TimeLine. In this case the cone rotates three and a half times during the animation. If you modify the declarations to:

```
#declare DGTLSegment01 = 0.5
#declare DGTLSegment02 = 2
#declare DGTLSegment03 = 2
#declare DGTLSegment04 = 3.5
```

Then the cone will rotate four times during the animation. So it is dependent on the total duration of all segments.

However if you just change the segment durations to:

```
#declare DGTLSegment01 = 0
#declare DGTLSegment02 = 0
#declare DGTLSegment03 = 2
#declare DGTLSegment04 = 0
```

Then the cone will rotate the correct number of times during segment 03. It will rotate once. Just as it would during segment 03 during the entire animation. (However it may not start in the proper point along its rotation. In this case it would normally be one quarter the way around its second trip to start the third segment. But without the previous durations it will start segment 3 in its original orientation.

DGTLMaster_Clock is given to allow continuous events to occur throughout an animation.

If you need to know more about the new duration of the animation DGTL also provides several other variables.

DGTLTotal_Time is the duration of all segments
DGTLTime_To_Go is the duration still to be covered in the animation

NOTE: Yes, I know that given the clock and the DGTLTotal_Time variables you can calculate the other yourself. However its easy for me to have DGTL calculate them for you.

Scene 3 - I Didn't Do It, I Was Framed

There is one other feature that DGTL provides. To see it in action enter the following into a new file (TL03.POV).

```
#include "colors.inc"

camera { location <0, 3, -6> look_at <0, 2, 0> }

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

plane { y, 0 pigment { color Gray50 } }

#declare DGTLSegment01  = 1
#declare DGTLSegment02  = 1
#declare DGTLSegment03  = 1
#declare DGTLSegment04  = 1
#declare DGTLFrames_Per = 10

#include "DGTL.INC"

text {
      ttf "TIMROM.TTF", "   Total Frames", 0.2, 0
      translate <-4,1,1>
      pigment { color Green }
}
text {
      ttf "TIMROM.TTF", str(DGTLFrames,0,1), 0.2, 0
      translate <2.5,1,1>
      pigment { color Red }
}


text {
      ttf "TIMROM.TTF", "Current Frame", 0.2, 0
      translate <-4,2.5,1>
      pigment { color Green }
}

text {
      ttf "TIMROM.TTF", str(DGTLCurrent_Frame,0,1), 0.2, 0
      translate <2.5,2.5,1>
      pigment { color Red }
}
```

I hope the results easy to understand. DGTLFrames_Per is simply multiplied by the total duration of the animation and DGTLFrames returns the total number of 'frames' in the animation

and DGTLCurrent_Frame returns the current ‘frame’ value. Notice that the DGTLCurrent_Frame value ranges from 0 to the DGTLFrames value.

HOWEVER - This value does not tell you which frame POV-Ray is rendering. In this example the DGTLFrames value will always be 40. It doesn’t matter if POV-Ray renders 10 or 10,000 frames.

I put this in for several reasons. First it gives a simple counting device throughout the animation. This can be used for whatever purpose you can dream up. Unlike the DGTLMaster_Clock variable it only has integer values, no decimal fractions.

Second if you know that eventually you are going to render the animation for broadcast of some sort then you can treat the duration of each segment as being seconds and then use the DGTLFrames_Per to give you an accurate internal frame counter. Again, if you render the animation without a matching value for frames being given to POV-Ray then the DGTLCurrent_Frame will not match the actual frame number.

Lastly, I left this in so that frames could be grabbed from another source and used as animated image maps. For example, if you want another animation loaded into POV-Ray on a rendered television set, then you can use DGTLCurrent_Frame to help you load the correct television frame as an image map. So far I haven’t had the time to attempt this trick. But as far as I can tell POV-Ray has all the necessary functions.

In Conclusion?

I hope this short tutorial has not left you completely confused.

DGTL is meant to take a single POV rendered animation and simply break it into numerous chunks. Each of which can be animated and dealt with separately.

You can use the DGTLSegment_Clock, DGTLMaster_Clock, DGTLCurrent_Segment, and the pov clock variable together to render very complicated animations.

Anything you can animate within a full POV-Ray animation (textures, lights, objects, scaling, rotations, translations,...) can be animated with a DGTL segment.

From now on its up to you.

Good Luck rendering!!

Reference

A little note about the variables used by DGTL. All start with DGTL 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.

User Assigned Variables

DGTLSegmentxx - float

There are 99 DGTLSegmentxx variables. Ranging from DGTLSegment01 through DGTLSegment99. If you declare one of them, and assign a float value, before you include "DGTL.INC" then the appropriate animation segment is assigned a duration.

To create a two segment animation with a duration of 4 for the first segment and a duration of 50 for the second you simply declare:

```
#declare DGTLSegment01 = 4
#declare DGTLSegment02 = 50
```

Segments do not have to be declared in order. DGTL will run the animation in numerical order, by segment number, regardless of the order they are declared. You may enter negative values, but DGTL will convert them into positive numbers automatically (sorry, no negative times here).

Segments do not have to be declared. If segments are not declared (segments 3 through 99 in the above example) the segments are automatically assigned a duration of 0.

If you do not declare any segments DGTL will assign a duration of 1 to segment 1, resulting in an animation with a total duration of 1.

DGTLFrames_Per - float

This variable is used by DGTL to calculate the total number of ‘frames’ in the animation. The resulting ‘frames’ values have no direct connection to the number of frames rendered by POV-Ray. Instead the returned values can be used as a counter throughout the entire animation.

Returned Variables

After you include DGTL it returns the results of its calculations in these varaibles.

DGTLTotal_Time - float

This is the total of all durations declared for all the segments in the animation. It is simply a total of all the DGSegmentXX variables.

DGTLMaster_Clock - float

This is the portion of the total duration that has passed in the animation before the currently rendered frame. If DGTLTotal_Time is 10 and POV-Ray is one quarter through the animation then DGTLMaster_Clock will be 2.5. (Yes this does depend on the Cyclical Animation option of POV-Ray).

Usually this is used to animate items that repeat throughout the entire animation.

A suggestion? You could use it to advance a clock during an animation. Digital or Analog, whatever you can construct.

DGTLCurrent_Segment - float

This gives the value of the segment of the currently rendering frame. It can be used in any numeric construct (#switch or #if, for example) to break apart the action with a single pov file.

DGTLSegment_Clock - float

POV-Ray provides a clock variable that ranges from 0 to 1 across an entire animation. DGTLSegment_Clock ranges from 0 to 1 across all the frames that appear in the current segment. If segment 02 appears for 25 frames then DGTLSegment_Clock will increase towards 1 through those 25 frames.

Note: There is no guarantee that it will start at 0 or end at 1 exactly. The first frame of a segment may appear a small fraction into the segment, and the value of DGTLSegment_Clock would be greater than 0.

DGTLSegment_Duration - float

This is the duration of the current segment. If POV-Ray is currently rendering a frame that occurs within segment 4 then this will have the same value as DGTLSegment04.

DGTLSegment_Start - float

This gives the duration of the animation before the current segment occurred. Given the following:

```
#declare DGTLSegment01 = 1
#declare DGTLSegment02 = 2
#declare DGTLSegment03 = 3
```

If we are currently rendering a frame within segment 3 then DGTLSegment_Start will equal 3.

DGTLSegment_End - float

This gives the duration through the animation when the current segment is finished. It is simply DGTLSegment_Start added to DGTLSegment_Duration. It is provided for convenience.

DGTLTime_Before_xx - float

This returns the total duration in the animation before segment xx.

If segment 05 is the first segment in the animation and it has a duration of 19 then,

```
DGTLTime_Before_01 = 0
DGTLTime_Before_02 = 0
DGTLTime_Before_03 = 0
DGTLTime_Before_04 = 0
DGTLTime_Before_05 = 0
DGTLTime_Before_06 = 19
```

would be the resulting values.

DGTLTime_To_Go - float

This is simply the duration left to go in the entire animation. It is the same as:

```
DGTLTotal_Time - DGTLMaster_Clock
```

and is calculated for convenience

DGTLFrames - float

This returns the total number of ‘frames’ in the current animation. See the tutorial section for a discussion of ‘DGTLFrames’ versus the number of frames actually being rendered by POV-Ray.

DGTLCurrent_Frame - float

This returns the current ‘frame’ number. This is based on the total time of the animation and the number of ‘frames_per’ unit duration. This is not based on the number of frames rendered by POV-Ray.

Hints and Tricks

Spread Out The Animation

While I assume that I'll never make mistakes, it's been known to happen. Specifically I've been known to create an animation and then want to add a segment to the middle. Either to space out, or add to the action.

If I've hard coded the animation with sequential segments (01, 02, 03, 04, 05) I'm not able to add a segment between any two.

So I've borrowed a technique from the days when I was entering basic programs into an old 6502 machine (has anybody out there ever heard of an Acorn Atom?). I simply skip a few segments between each part of the action when I first create it (05, 10, 15, 20, 25). Then if I have to enter a new segment I have a few slots left over. I can easily create a new segment at number 7.

DGTL will simply treat all the intervening segments as having zero length. You don't have to set them to zero length. In fact all the segments you don't specify are set to zero length. (In the above example 26 through 99 are set to zero).

Animate Across Segments

Occasionally you may want an action to cover multiple segments. You may want an action to start at the beginning of segment 3 and run through until the end of segment 5. This is possible, but you have to do some math yourself. To accomplish this you could do the following:

```
#if ((DGTLCurrent_Segment >= 3) & (DGTLCurrent_Segment <= 5))
  // Animate across these segments
  #declare NewTotalTime = DGTLSegment03 + DGTLSegment04 + DGTLSegment05
  #switch (DGTLCurrent_Segment)
    #case (3)
      #NewClock = (DGTLSegment_Clock * DGTLSegment03)
      #NewClock = NewClock / NewTotalTime
    #break
    #case (4)
      #NewClock = DGTLSegment03 + (DGTLSegment04 * DGTLSegment_Clock)
      #NewClock = NewClock / NewTotalTime
    #break
    #else
      #NewClock=DGTLSegment04+DGTLSegment03+(DGTLSegment04* DGTLSegment_Clock)
      #NewClock = NewClock / NewTotalTime
  #end
  //animate using NewClock as the clock value between Segment03 and Segment05
#end
```

So, while you have to do a bit of work to create a multiple segment clock value you can then easily animate using the new clock value.

You can also write more complicated statements. If you wanted to figure out the math it would be possible to animate from 2 seconds into segment 3 until 4 seconds before the end of segment 5. It would just be a matter of building the correct math to figure out the new clock value.

Declare Your Segments!

This trick is mainly for very complicated scene files. If you intend to construct a file that is very complicated, or includes many inc files with additional objects and animation then declaring names for segment numbers can make a great deal of sense. For example:

In MYFILE.POV

```
#declare MonsterStand   = 5 // Segment with a Standing Monster
#declare MonsterFalling = 6 // Segment with a Falling Monster
#declare MonsterDown    = 7 // Segment with a Grounded Monster
```

In MONSTER.INC

```
#switch (DGTLCurrent_Segment)
   #case (MonsterStand)
      // show the monster standing

   #break
   #case (MonsterFalling)
      // make the monster fall during this segment

   #break
   #case (MonsterDown)
      // show the monster on the ground

   #break
   #else
   // ignore all other values
#end
```

This way you can change the segments in which the monster appears, falls, and rests on the ground by simply altering the declares in your master pov file.

Another variation allows for more control within the master file:

```
#declare MnstrStanding    = 3   // When the monster starts standing
#declare MnstrFalling     = 6   // When the monster starts falling
#declare MnstrGrounded    = 10  // When the monster ends up on the ground
#declare MnstrDpears      = 12  // When the monster dissappears

#declare MnstrStand = false
#declare MnstrFall  = false
#declare MnstrDown  = false

#if ((DGTLCurrent_Segment>=MnstrStanding)&(DGTLCurrent_Segment<MnstrFalling))
   #declare MnstrStand = true
#end
#if ((DGTLCurrent_Segment>=MnstrFalling)&(DGTLCurrent_Segment<MnstrGrounded))
   #declare MnstrFall = true
#end
#if ((DGTLCurrent_Segment>=MnstrGrounded)&(DGTLCurrent_Segment<MnstrDpears))
   #declare MnstrDown = true
#end
```

The internals of the MONSTER.INC file are left for you to figure out.

Declaring segment names is the easiest way to incorporate action from included files. This way the included files can also be re-used in other scenes. And when they are re-used it is easy to specify new segments for the animation to take place.

More Hints?

A simple thought to help you create more complicated animations. You can treat every segment as an individual POV-Ray animation. Any, or all, tricks to make an animation using a single clock can work within a single segment. If you look at the Hints and Tricks section of Dave G's Bezier Curve System you will find a number of ways of using curves to generate animation that accelerates and decelerates. All these techniques can be used within a single segment of the TimeLine (or as seen above - across multiple segments). So you can use other tools and techniques to enhance your animation.

And I'm always interested in hearing about more hints and uses of DGTL. If you come up with any interesting techniques or ideas please let me know!