Textual Concepts and Their Conveyance through Digital Media

Katsuaki TAIRA

Concepts and their vehicles. The scheme sounds very familiar and it should be. The idea of dichotomizing the linguistic components has been in existence for years and the latest fashion raged just a couple of decades ago. But in light of the computer technology I would like to dig out the old argument once again and present various possibilities based on that. To tell the truth, the discrepancy between the inchoate ideas and their vehicle, or means of communication, can be so great that no attempts can possibly bridge the two state of human expression in the successful manner as we ordinarily attribute to the word. We must be humble and admit that indeed extraordinary obstacles lie between the original ideation as it happens in human psyche and a modicum of semiotic coalescence that ultimately derives from the original ideation. But, if I may repeat myself and thus run the risk of becoming tautological, we should try to find a breakthrough and open that redoubtable gate of that conceptual field and come up with the means to render the conceptual flux more immediate and less reliant on the conventional linguistic means. Yes, I have already let you in on my hidden agenda. I am in fact in search of the mechanism in which the two entities or states I have referred to can be seamlessly connected and the generating conceptual image can be directed to the party intended by the very setup that constitutes the two-way communication. Of course, I am not proposing a theoretical setup in which Leibnizian[1] atoms of human thought do all to facilitate the conceptual translation. That would be great. But considering all the difficulties that besiege such a scheme, it may be well nigh impossible. However, what if some virtual space existed where the bi-directional communication envisaged by Leibniz is not only possible but actually practiced without even the conceptual heterogeneity being experienced by the party

[1] Leibniz hypothesized that human beings were born with "the alphabet of human thoughts." All complex thoughts, according to him, were different combinations of such alphabet. Thus, no matter how abstruse the ideas were, they could be explained away as long as those essential constituents were securely grasped and the paths to complex ideas were meticulously studied. See *Opuscules et Fragments Inedits de Leibniz*, edited by Louis Couturat.

engaged in it? That is my objective and that is my topic I would like to pursue in this essay. That is, to come upon that plane and explorer all kinds of possibilities when such bi-directional communication is reified in some form, especially via the computer technology. Needless to say, to perfect such a virtual world would be prohibitively complicated and technically demanding, very likely far beyond the reaches of an individual intellection. Although my intention is ambitious, I am not arrogant enough to exclude the hypothetical ratiocinations to approach the "immediate communication" here envisaged. Now, without much ado let me plunge into the arcane world of conceptual transposition without the traditional linguistic mediation as we have known it.

I would like to start with a simple proposition that substantive concepts are easier rendered through non-linguistic means than abstract and oftentimes multivalent complex nouns that are often the metaphorical extensions of the former.[2] Or, perhaps I may be just simplifying the matter. But for a starter let me devise a scheme in which such nouns in their conceptual essence can be conveyed through multimedia. Needless to say, any kind of multimedia qualifies for such mediational candidate. However, for the arguments' sake, I will limit the type of multimedia in question to the kind delivered through and controlled mainly by scripts.[3] I do not intend to limit my horizon by the proposed means to an unnecessarily smaller reaches and simplify the variegated and multifarious territory that is comprehensively referred to here as extra-linguistic/nonlinear communication. On the contrary, the narrowed focus is chosen to bring the paper within a manageable scope and no ulterior motive is swept under the rug to over-generalize the complex communication scheme heretofore repeatedly touched upon. Now, without further delay let me shift to my main argument and introduce the reader to what I have been hinting at for quite some time now.

---

[2] By the two opposed concepts I am distinguishing the concrete terms such as apple, orange and desk from abstract nouns such as rationalization, concretization and condensation. The boundary between the two opposed terms may be blurred if you classified every single nouns listed in English dictionaries into two camps. But I am here resorting to the traditional device of grammatical taxonomy (and very simplified version at that) for the sake of bringing my point across.

[3] The scripts here refer to the type generated by a proprietary computer language called *Lingo*. Although it is not generic in the sense that everyone can access and generate and edit the core language itself in the manner that suits the programmer, it is nevertheless a language that allows the programmer every manner of flexibility to control the contents in his presentation. Because of the prominence the company that owns the right to Lingo enjoys in the industry and the dominance of the application itself that incorporates Lingo, I am emboldened to proceed to develop

First, let us study the optimum merger of sound and sight, or in this case sound clips and graphics, using Lingo. Any combination between the two elements can be rendered synergistically if they are yanked together appropriately. What about the linguistic concepts conveyed through the said means? How about, for instance, the eye as in Shakespeare's "Is it for fear to wet a widow's eye...?"[4] How can such conveyance be made possible in the environment I have been describing since the beginning of this paper? The explication of which will be my main concern for the time being. The mere presentation of the eye can be accomplished by displaying the graphic that is literally associated with the thing in question. It could be a mere photographic image of the same or painted representation of the eye. It does not matter but as long as the eye in its semantically as well as semantically conveyable form is depicted on the stage,[5] the party interacting with the program should have no difficulty interpreting the image. In fact, interpreting may not be appropriate here. Since interpretation implies mental ratiocination that brings the subject's mind from state A to stage B in a logical process most likely underpinned by linguistic mediation, the less such activity is involved in the act of perception the better the extra-lingual communication is surmised to take place. Needless to say, that may as well be over-generalizing. However, the point of my argument is that when the image is presented before the interacting subject mind, it is better be intuited as representing one thing rather than another. If one insists that such idealized communicative approach is like building a castle in the air, then that is fine. But let us for the moment presume that such "idealized" extra lingual approach is indeed realizable.. (Otherwise, my argument is so much wasted air.) The presentation of the image can be achieved in the following manner.

```
on beginSprite me
  sprite(me.spriteNum).member=member("eyeFilledWithTears")
  sprite(me.spriteNum).ink=36
end
```

The handler beginSprite catches the message the program releases when the

[4] The quote is from Shakespeare's Sonnet 9. It is the first line that is proceeded by "That thou consum'st thyself in single life?/ Ah, if thou issueless shalt hap to die,/ the world will wail thee like a makeless wife; etc." (Excerpted from *William Shakespeare: The Complete Works*, edited by Alfred Harbage.)

[5] The stage is the area where all the visible elements appear. It is an interface through which the parties concerned interact with each other. If the presentation being built involves only one actant, or the party which initiates actions, then the stage can also be conceived as a window through which communication of any kind can be achived.

presentation starts. Because this handler is one of the first ones that are evoked when the program gets under way, it is convenient to assign the target image to the channel that is referred to here as me.spriteNum. The value me.spriteNum obtains is an integer such as 1,2 and 3. Because the dot syntax indicates that the number of channel returned is the channel the very script is attached to, the item on the right side of the equation ultimately is assigned to the channel or sprite me.spriteNum. The second line makes the redundant part of the image transparent if there is such a peripheral area. In other words, ink is a property that determines the visibility of the graphic depending on the elements that particular image contains. If all the rectangular area of the image is absolutely necessary, then either the property is set to 0 or copy. That is one method to translate the image onto the stage. Another way to achieve the same goal is to place the target image in the desired channel before the program is actually run or while the presentation is being built. In that case there is no need to specifically assign the graphic image to the channel of choice. As the image is dragged from the cast interface onto the stage, it obtains its channel as well as stage position automatically unless the channel or sprite is specifically designated by clicking on the intended sprite before the drag and drop move is executed. The ink value needs to be assigned if the peripheral area needs to be made transparent. But that is also accomplished by choosing an appropriate ink type from the popup menu included in the interface.

Although a simple stationary graphic ocular image might do for the occasion, in a presentation here envisioned there is a choice whether to animate the scene or present it as a fixed presence, or a combination of the two. Let us consider the first case. How is it accomplished? There are a number of ways, in fact. But I will introduce one strategy that incorporates two kinds of scripts. First, bring in the eye to the scene by the following handler.

```
on beginSprite me
  sprite(me.spriteNum).member=member("eyeFilledWithTears")
  sprite(me.spriteNum).ink=36
  eyeObj=new(script "eyeAnimPrnt", me.spriteNum, “eyeFilledWithTears”)
  goAppend(eyeObj)
  puppetSound 1, "widowsEyeI"
end
```

The first two lines are the same as above. That is, the script assumes that the image is to be displayed through the script rather than manually by dragging from the cast window. The next line instantiates the named parent script and pass an argument (in this case the channel number in which the animation is to take place) to the same script and finally assigns the whole thing to the variable named eyeObj. Once the script is instantiated it is included in the actorList by the ensuing line. The goAppend is a customized function that invokes a handler in the designated parent script. The object needs to be added to the actorList, which is a global list, in order for the stepFrame message is to be intercepted every time the playback-head moves across the frame. As the stepFrame message becomes accessible from the called upon script, the graphic that is under the control of the instantiated object is animated. The last line in the above script is a little embellishment. As the object is instantiated from the named parent script, the designated sound is played to create the necessary ambience to convey the maximum semiotic significance hopefully through the combination of the graphic and the sound. The sound can also be controlled by the same object whether it is played once or prolonged as long as the situation demands. Or, more likely when to turn off the sound. The same control can be obtained also by dint of the stepFrame handler. As the animation is generated through the tactful use of the moving playback head and the temporal flow created by the punctuation of the stepFrame handler, any command such as muting the sound can be issued as soon as certain condition is satisfied. But all that contraption cannot be realized without the referred to parent script. Now, let us look at it and see how it is built and how it handles all the requests made upon it by the above handler.

```
property pSpriteNum
property pMemNm
property pInitT
property pMmSffx
property pMnMmSffx
property pMxMmSffx
property pBlinkCycle

on new me, spriteNum, memNm
  pSpriteNum=spriteNum
  pMnMmSffx=1
  pMmSffx=pMnMmSffx
  pMemNm=memNm
  sprite(pSpriteNum).member=member(pMemNm&&pMmSffx)
  pInitT=the ticks
  pMxMmSffx=4
```

```
    pBlinkCycle=8
    return me
  end

  on goAppend me
    if getOne(the actorList, me)=0 then
      append(the actorList, me)
    end if
  end

  on stepFrame me
    if the ticks<(3+random(pBlinkCycle))*60+pInitT then
      exit
    else
      pMmSffx=pMmSffx+1
      if pMmSffx>pMxMmSffx then
        pMmSffx=pMnMmSffx
      end if
      sprite(pSpriteNum).member=member(pMemNm&&pMmSffx)
      pInitT=the ticks
    end if
  end
```

The above is a rather simplified receiving script that is called upon by the preceding handler. First, the local properties are declared to be used later further down in the parent script.[6] They are rather numerous. Although all of them need not be used for this particular script they are there to exemplify the convenience of utilizing variables to store values that are later to be retrieved. The first handler in the parent script is

[6] There are three kinds of script in Lingo. A behavior script is a kind that is directly attached to the acting element in the presentation, thus its mechanism most intuitively conceptualized. A movie script is a kind that has relevance to the entire presentation, usually functioning as a central fulcrum to synchronize each and individual behaviors. A parent script is an object that is instantiated from the parent script itself and maintains an independent and encapsulated status once given birth to by the new() function. There is an insightful observation on the subject by Christopher Robbins, Brian Douglas and Justin Clayden in *Director 8.5 Studio*, published by friends of ED. For further details consult their articles titled "Object Orienting Lingo Programming" and "Gaming with OOP" in the same book.

used to instantiate the script itself and return the object to the calling handler. The first parameter that follows "me" in on new me handler represents a value that corresponds to the sprite number the calling script is attached to, i.e. the channel the graphic member is assigned to. The second parameter indicates the member name of the cast that is associated with the designated channel. The first argument is in integer and the second in string. Once the appropriate exchanges are made between the two handlers, each value is stored in descriptively named local properties. The first parameter is evaluated to pSpriteNum and the second one to pMemNm. The rest of the properties are hard coded and generalized to facilitate the work later on. The generalization is the key here. Hard coding is not necessarily a wise strategy but by making the property-assigned values available throughout the script all kinds of operations that need to access those values are likely to be executed without careless flaws and insignificant yet accumulative blunders. Once the needed values are stored in respective variables, the instantiated object is returned to the calling handler by return me in the last line of the present code. That brings us back to the initial handler, which invoked the parent script. What the obtained object allows one to do is call the handler within the parent script and put the object in the global list, the actorList. The process in turn enables the object o detect the stepFrame handler that occurs every time the playback head crosses the one frame into another. The setup, as you can see, is fairly simple but it is nonetheless effective and suffices our need for the moment. Once the object is property stored in the actorList the code in the next handler is immediately run. What it does is as follows. First, the animation cycle is checked by comparing the time elapsed from the previous cycle to the randomized waiting time set by the variable pBlinkCycle and the random function random() multiplied by 60 and added to the integer value 3. (The latter integer is added to secure the minimum waiting time for the animation to cycle through.) When the elapsed time is less than the designated time span the exit command forces the program to rerun the stepFrame handler, as it were, until the designated waiting time is reached. Once that condition is satisfied, the line pMmSffx=pMmSffx+1 forces the cast member to switch to the next one in order according to their position in the cast member allocation. Before actual reassigning of the member takes place, the suffix value of the member is checked in order that the number does not exceed the pre-assigned value, which is 4 in this case as indicated by pMxMmSffx=4. If the suffix is found to be larger than 4, then, it is guaranteed to be returned to its initial value, pMmSffx=pMnMmSffx. Finally, the member that has survived all the conditionals is assigned to the designated channel by sprite(pSpriteNum).member=member(pMemNm&&pMmSffx). Simultaneously, the ticks are equated to the current ticks so that the time difference between now (zero time) and the one temporal cycle later can be effectively measured, allowing another member change to take place. If one wants to subject the whole animation sequence to music or any sound effect in its temporal expansion, one can insert lines to check the cessation of audio effect or simply, if the audio effect is under temporal control, the time lapse. If the former is the case, one can write the code in the following manner.

```
 if soundBusy(1) then
   deleteOne(the actorList, me)
   --any appropriate commands
 end if
```

The first line checks whether the sound is playing. If not the next line removes the object from the global actorList, which effectively ends the animation cycle by making the stepFrame message "inaudible" to the object. The proper place for these lines to be embedded is right after the stepFrame handler declaration. The end of sound unconditionally puts an end to the cycle. For the latter method, at least one more property variable is required—one that stores the initial ticks so that the time lapse can be measured uninterrupted throughout the entire animation. It needs to be set in the object initializing handler somewhat in the following manner.

```
 pInitCntrlTicks=the ticks
```

Of course, the property needs to be declared along with other properties at the top of the script. Once the pInitCntrlTicks is evaluated, the conditional lines are to be inserted in the same place as the sound controller in the following manner.

```
 if the ticks>pInitCntrlTicks+waitTime*60 then
   --waitTime here stands for the time in integer the program waits until executing
 the following lines
   deleteOne(the actorList, me)
   puppetSound 1, 0
 end if
```

Whichever is the case, when the designated time arrives the program terminates the current sequence of animation and goes on to the next phase. But before going on to the next phase, let me introduce another approach to convey the possible non-lingual yet semiotically significant and sententially filled images through multimedia.

The approach I just mentioned is quite new. In fact, it has been just recently made possible by the inclusion of what they called "3D engine" in the application, which is the center of my whole argument. By taking advantage of the engine it is now possible to manipulate images and animations that populate the Lingo enabled virtual space. Although creation of such space is calculation intensive (in may sense) and demanding on the CPU, the reality effect it achieves in the process seems to promises so many possibilities that it would be a shame not to break away from the constricting environment the old 2D has been imposing on us. Now, without much delay let me delineate the reality-impacted world of 3D through Lingo.

First, you need to instantiate a type called #Shockwave3D and assign it a member status.

```
eyeModel=new(#Shockwave3D, member(10))
```

For convenience, let us give the member a name, which is in #string.

```
eyeModel.name="weepingEye"
```

Although we have given birth to a Shockwave3D instance, it is not a visible entity yet. In fact, if you count the number of polygons and models they amount to 0. that is, the entity that has been instantiated by the new() operation is there in the member slot but still invisible. The camera and two lights do not help much because there is there is nothing to capture and shine on in the virtual space. Now, in order to create the target model, we need to create the bare minimum structural constituent that will be used to configure the shape we are trying to give shape to. The primitive is generated in the following manner.

```
res=eyeModel.newModelResource("eyeBall",#sphere)
```

However, still no visible presence in the target space. What is the reason for such aversion to be included in the 3D space by our poor weeping eye? Nothing. That is as it should be. The above function merely creates a bare primitive and it simply give rise to a possibility that it may later be included in the target space. Be that as it may, let us define the size of the globe. (By the way, eyeball is taken rather literally here as a perfect globe—well, almost, depending on the drawing power of the individual computer.) We will set the radius.

```
res.radius=20
```

It is not time to create or declare an eye model that will be worked on to form the final weeping eye.

```
mdl=eyeModel.newModel("eyeBallL")
```

This process is merely declaring a possibility of realizing a visible model "eyeball." The model is not there yet, at least in the visible form. In order for that to happen, the model needs to be associated with the resource, out of which it will be constituted.

```
mdl.resource=res
```

Finally, a lone presence of a checkered globe in the middle of the window. It is gratifying as well as disheartening. Although the globe is certainly there, it hardly looks the eye we have been striving to create. We need more than the simple motley design on the surface. What is it lacking? Texture and shader. First, we need to create a material that is used as a texture on the surface of the eye. It could be either a color chip or an image that effectively represents the weeping eye. The texture is not used as a two-dimensional image, however. It is mapped to the curvature of the eye. Therefore, proper projection of the former has to be calculated in order for the perfect wrapping of the image takes place. We decide to use the image as a surface representation of the eyeball.

```
tex=eyeModel.newTexture("eyeTexture", #fromCastmember, member("eyeTexture"))
```

However, creating the texture is not enough to represent a sphere in the 3D space. We need to give rise to a shader, which in the words of Joel Baumann "describes the quality of the surface of a model the texture is applied to...."[7] Then how do you generate the shader? It is simple. Just as if you generated the texture, you instantiate a shader by the instantiation function by setting the name and the type in the following manner.

```
shd=eyeModel.newShader("eyeShader", #standard)
```

Thus the possibility of applying the shader to the target model has been created. The long awaited manifestation is just around the corner. All you need is to attribute the texture that has been defined above to the shader just instantiated.

```
shd.texture=tex
```

Finally, you attach the shader to the model itching to appear before your eyes. That is just about the bare bone procedure to create a model in the 3D space. If you want to add a little panache, which is actually required if you want to make the animation convincing, then you need to rotate, translate and do other things on the target model by member.object.transform.translate(x,y,z), member.object.transform.rotate(x,y,z, #center), etc. Needless to say, the sound element can be incorporated in the sequence. The method is quite intuitive and not much different from the one used in the 2D environment. If you want the sound to start as the image is started to be drawn, just plant the following code in the beginSprite handler.

```
sound(1).queue([#member:member("weeping eyes"),#loopCount: 1])
sound(1).play()
```

The first line prepares the target sound to be played with the additional condition, how many times to be played. The second line actually executes the sound playback. Once

[7] In other words, a shader determines the perceived quality of the material surface. Without this attributes no model can present itself in the manner that appeals to the ordinary perceptive responses of a viewer. The quoted passage is from Chapter 7 of *Director 8.5 Studio*, p. 295.

the simple code is safely tucked in the designated handler, the animation is played out to the tune of the heart-wrenching or ironic, depending on the intention of the author, melody, which, if successful, draws the audience's hearts into the scene being panning out right in front of their eyes.

Since I have so extensively talked about the visual aspect of the conceptual representation, (although I have without much explanation already incorporated auditory elements in the presentation), it is time to lay out its sensory counterpart as to how it is reified in the consciousness of the person communicating with the hypothetical program and how it will be put forward to aid the kind of semio/semantic exchange I have already characterized as non-linear and nonverbal. This time, let me attempt to establish a rather abstract concept that is yet associated with the auditory sense. Without garbling my argument, let me announce, (needless to say, as rendered in the conventional non-linear epistemology), my next target word: echo. That is, echo as in "Echoes traveling/Off from the center like horses," as Sylvia Plath well described in her suicidal limbo.[8] The choice of the word is completely arbitrary and there is no absolute category to favor that particular word over others. Now, back to the multimedia scheme to represent the concept somehow. What is needed first of all to put the idea on the stage is to "spatialize" the audio element, as Andrew Allenson says in *Director 8.5 Studio*. In order for that to happen we resort to the volume property of Lingo as manifested in the following syntax.

```
sound(channel).volume
```

The idea is to manipulate the volume so that it increases and decreases in synchronization with the moving object, which functions as a visual meeting point between the program and the user. Another element that enters the picture is the pan effect. Pan of course controls the spread of the sound between the numeric value of –100 (far left) and 100 (far right). It is another property that adds dimensionality to the entire presentation.

[8] The line is from Sylvia Plath's "Words." The choice, once again, is random but the title fortuitously seems to bear out my argument in the most tangential sense. Because of the agony the poet was subjected to she needed to find the outlet to express her psychotic state through. But at the same time it tormented her as they pulsated their way through her mind out of control. Now, my attempt is to channel that not-easy to control semiotic energy through the media computer based presentation offers and reconcile that with the image that appears on the stage/monitor and hopefully find the rapprochement between the two as they congeal into an intelligible concept in the minds of the people involved in the multimedia environment. The quote is from *The Norton Anthology of American Literature*, volume 2, edited by Ronald Gottesman and others.

sound(channel).pan

Since all the Doppler experience is to take place in 3D space, let us incorporate all the elements necessary to structure that scheme. First, we assign a 3D member to the target channel and store the member reference in variable p3D. (It has been declared to make itself available throughout the script.) Next, we issue the resetWorld() command to keep the acting models from cluttering the virtual space every time the movie runs.

```
p3D=sprite(me.spriteNum).member
p3D.resetWorld()
```

The preliminary setup phase ends as the camera is positioned at the proper vector coordinates, allowing the viewer to enjoy the whole spectacle as it plays out before his eyes. In this particular case, it is located 600 values on z-axis, that is, away from the viewer.

```
p3D.camera.transform.position=vector(0,0,600)
```

When I said the preliminary phase I was not quite accurate. There area number of more lines to be inserted in the initial handler, which the above codes are a part of. Creating a model resource is a de rigueur. The way it is executed is as follows.

```
eyeResource= p3D.newModelResource("eyeResource", #eye)
```

The syntax can be explained thus. First, the model resource that is coming into being needs to be associated with the member, which it would be a template constituent of. In other words, it has to be defined as residing in the overall environment of the 3D space here named as p3D. Once the association is made, the function newModelResource() generates the designated resource in the type appended in symbol, which is preceded by #. The line finally endows the variable with the proper reference to the eyeResource. Creating a model template is just a beginning of the process. Next, let us create a texture template. The logic behind it is the same as generating the model template. The syntax to execute the task is very much the same as the previous one.

```
eyeTexture=p3D.newTexture("texture", #fromCastmember, member("texture"))
```

The first term on the right side of the equation is the reference to the 3D parent environment. The dot effectively associates the product of the function newTexture() to the overall 3D space. The first parameter in the parentheses is the name given to the generated texture. The second one indicates the source type, the third the constituent by which the member is referred to. The result of the operation implemented on the right side is stored in variable eyeTexture. There are several more steps before model is wrapped with the generated texture. The order of some of the steps involved is somehow arbitrary. Let us create a model next to be populated in the virtual space. The new model also has to be associated with the overall space in which it will be thrown into. All the initiation, therefore, assumes a similar syntax.

```
eyeModel=p3D.newModel("eyeModel", eyeResource)
```

For the first time, however, the code inside the current handler refers to the variable evaluated during this session. The eyeResource, defined as the second parameter inside the newModel function, is the reference to the model template instantiated in the above line. Therefore, what this line means is that the given name "eyeModel" is a referent of the model that is built upon the model resource "eyeResource." At this point, there is not only the possibility of building a model but also inhabiting the p3D space with actual models. The rest of the code focuses on how the models (here we will be using only one model, however) will be represented and made visible to the interactive participant. If the target objects need to be placed at a particular special location in the 3D world, then the following code is necessary to accomplish that.

```
eyeModel.transform.position=vector(x,y,z)
```

Needless to say, the x, y, z in the vector object indelicate the appropriate float that determines the vector value of each 3D coordinate axes. Since the default position of objects when they are created through Lingo commands is the origin of the world (that is, the world of 3D space that develops before the user's eyes as he faces the interface), manipulation of the object location may be necessary in order to place it at the right place, no matter whether the target elements are part of a larger object or an independent entity that is self contained to itself. But since we are dealing with the eyes that are very likely attached to other models, thus constituting the parent child model relation, the positioning involved here is most likely to be controlled through the vector object that deals with the hierarchical group. Now, without too much delay, let us focus on the representational process of the model generation. As has been already fully adumbrated, models need to be associated with textures and shaders. As for the texture the current model already has a potentially usable texture, as is evident from the above texture generation code. That leaves a shader to be created. The way to accomplish it is analogous to the texture generation except the command utilized is appropriately different.

```
eyeShader=p3d.newShader("eyeShader",#standard)
```

Here, the function newShader() generates a shader named "eyeShader" and attributes it to the 3D environment represented by p3d. As usual with other codes, the resultant object is stored in the variable eyeShader so that it will be conveniently used as a pointer to the relevant shader later on. The second parameter in newShader()

indicates the type of shader that in turn allows further elaboration on the shader property itself. For this occasion, we will assign to the newly instantiated shader the texture that has been given birth to in the above command line. The syntax necessary to link the texture and the shader is as follows.

```
eyeShader.texture=eyeTexture
```

Once the shader-texture association I made, the final step is to link the actually instantiated model with the shader. That is the end of the rather lengthy process to present a model into the 3D world. Admittedly, the process involved in setting up the 3-demensinal models in the target environment is quite lengthy compared with that needed to introduce 2D images to the interactive players. But considering the potential impact the well-delineated models have on the users, the labor that goes into creating the 3D environment may be well worth it. Needless to say, the use of each dimensional approach is dictated by the effects each one is supposed to have on the users. Of course, even those effects may be dictated by the context in which the presentation, or fro that matter the users, are placed as they try to absorb the full significatory implications of the intended extra-lingual epistemological and experiential emanations. Before I leave the general structuation of the 3-dimensional eye model approach, let me append the sound-trigger mechanism that is analogous to the sound cue method I introduced above. Like the previous example, the synergistic effects both the visual components and the auditory elements of the presentation bring to the whole setup are expected to be considerable. There is no reason to skip an opportunity to create the psycho-spiritual encounter between the program user and the program itself through a crafty manipulation of the multimedia elements that are a part of the whole presentation. The codes needed fro the proposed scheme is not complicated. At least just to trigger the sound at the moment the target eye initiates a certain action, or more likely a group of actions. Suppose the model(s) generated through the above process will immediately step into an animation cycle, the occasion for that cycle has to happen only once in order for later string of eventualities to take place. If that is the case (and indeed that has to be the case unless the animation scheme breaks down and the program willy-nilly grinds to a halt in no time), then the optimum place to insert the sound trigger cue would be where the action is initially called. Therefore, the favorite handler to hold the initial codes would be "on beginSprite" and one likely location to insert the sound trigger mechanism would be at the end of the model generation procedure. That is because models have to exist first in order for them to dance to the tune of well-calculated sound effect. At least logically speaking. (But in the world of computers, logic often has to bend to pragmatic exigencies and quirky responses certain codes educe from the computers. Therefore, the moral of the story here is that you should take nothing for granted.) The lines to trigger the sound effect are as follows.

```
sound(x).queue([#member: member("3DeyeBallSfx"),#loopCount: n])
sound(x).play()
```

Here, x and n are to be substituted by appropriate integers. The x , of course, stands for the channel in which the sound plays and the n the number of times the sound element repeats after it has been triggered by the play() function.

Other than triggering the sound when the initial action is started, you can make the triggering of sound or other series of animation contingent upon a certain packet of conditions. That is not very difficult. The ideal handler that meets such demand would be the exitFrame, (or for that matter, anything that deals with the frame action).[9] Since the frame message is incessantly released as long as the playback head loops in the perimeter of the designated frame, the appropriate structure needed here is the if... else syntax. For instance, if the desired effect one is seeking is the rotation animation, in which the mysterious eyeball rotates around a certain object (what object is something one has to decides before hand). Then, the movement of the eye has to be controlled by the vector manipulation. Needless to say, the position and rotation of the eyeball, or the object, parent or not, the eyeball is most likely belongs to, have to be defined in the initial handler before the current animation is started. Therefore, setup in the previous handler needs to be elaborated upon further a little bit. (Here, the previous does not necessarily betoken previous in spatial order, as in previous to the current handler, but more to do with temporality. Anything that occurs previous to the current action is comprehensively referred to as previous.) Suppose the positionality or transform of the eyeball is to be defined in the beginSprite handler, then the method to do it is expressed in the following manner.

```
eyeModel.transform.position=vector(x,y,z)
eyeModel.transform.rotation=vector(xDegree,yDegree,zDegree)
```

The first line is the code that has already been introduced. The second one is a further elaboration on the first line, which in turn gives the programmer more control over the object in question. As the last property indicates it manipulates the rotation or angle

[9] Or, more concretely those handlers that have the frame component as a part of their handler name such as enterFrame, prepareFrame, stepFrame and exitFrame, which happens to be the handler I have adopted for this demonstration. Needless to say, there are many other mechanism that can intercept the message that are sent when the playback head crosses the denominated frame demarcation. But for this occasion simplicity should be the dominant ethos to make my point to convey a maximum amount of intangible, non-linear, non-verbal concepts through the media I have been dealing with.

the given object is facing toward the target view. (For simplicity, let us assume, the eyeball is an independent object that moves on its own, apart from anything else it is, as I have been repeating, very likely attached to.) Once the initial position and rotation have been defined, the object can be controlled by the handler that intercepts messages that cyclically arise as the program runs. In the current presentation, the visible indicator that signals the cyclical emission of messages is the playback head. If you keep the playback head looping on the same frame, the script that is attached to the frame can be run as often as the immediate needs demand. Suppose, as I have been suggesting, the eyeball needs to be animated, then simple codes that respond to the ever-moving playback head suffice for the occasion. If the eye is to be moved and rotated by certain units and degrees, the following lines would represent the general adumbration of the syntactical scheme that is likely to be employed in the interceptor script.

```
  eyeModel.translate(x,y,z)
  eyeModel.rotate(xDegree,yDegree,zDegree)
```

Since the unchecked translation and rotation sends the object away from the view, some conditional statement would be needed to rewind the movement back to where it has started, or at least curtail the unwanted positioning of the object within an allowable degree. That should not be difficult, of course. All you have to do is to calculate the values in the opposite direction so that the object's position and rotation are returned to their original transform as the frame number is counted down backwards, as it were, by the looping playback head. But one caveat here would be not to reverse the order haphazardly and carelessly reverse the translation and rotation process. Because of the way objects are situated in 3D space, the procedural stack of commands needs to be accurate in order to bring the identical object back to its original positionality.

Since I ventured into the arcane world of 3D, it may be appropriate to demonstrate the procedural coding to create the 3D elements from scratch. The technique of populating the 3D world needs to be considered later. All the complex union of 3D shapes and movements and then the harmonious presentation of the whole in combination with music can be further delved into once the necessary steps are taken. Simplifying the twisted and knurled 3D world may in fact bring the sense of the virtual world even closer home in the long run. With that in mind, let us start with the simplest, or rather the most elementary constituent of the 3D world. In order to streamline the process, let us come up with the routine handler that responds to the demand to generate the type of primitive that will reside in the virtual world. First, allow the 3D space to become a global variable by declaring it as such at the top of the projected handler. That operation is very simple. You merely announce it by appending the variable to the term global. The simple process prepares the way fro the complicated operations that are later to be attributed to the 3D world, which is exactly the domain demarcated by the global variable thus declared. Supposed our purpose is to create a plane. The handler reflects the teleological intent by assuming the name within its constitutive self. The handler, appropriately enough thus, is on createPlane and the arguments it takes is the name that is used to reference the end product, length and width of the plane and rgb color of the plane at the end. Put it more schematically, the handler reads

on createPlane(planeName, L, W, planeColor).

Once again the first argument takes a string, the next two arguments integer, the last, by implication, rgb(R,G,B). Everything should be clear so far. Before the actual command runs, a conditional line is inserted to safeguard the program from populating the constitutive elements in an empty, or rather non-existent, 3D world. Because populating 3D elements in non-existent 3D environment is a non sequitur it cannot be allowed to happen. That is exactly what the conditional if statement is coded to achieve. The syntax for the conditional is if check3Dready(scene) then. Of course the argument the function check3Dready takes is the global variable that has already been declared above. Because the variable has been evaluated, the handler thus called returns the argument in the Boolean without any demurral. Now we need to create the fundamental resource on which the ultimate product—in this case a mere plane object—is based. The process is consistent with those involved with creating other primitives.

Res=scene.newmodelresource(planeName & "res", #plane)

The first term used on the left side of the equation is the world in which all the objects that are to be created will in habit. The generic term used in the present program is 3D world. The dot following it indicates that the model resource created in the given manner is a property of the given 3D world. The proceeding operation newmodelresource(planeName & "res", #plane) signifies that the newmodelresource() is a function that gives rise to the type of resource referred to as planeName & "res." Needless to say, &, the concatenator, indicates that the two string aggregates are to form one name without any space in the chunk. The method used here is intended to reduce the complexities arising from naming each and every resource in entirely unrelated string identifiers. Of course every term referred to each discreet object has to be differentiated somehow, but by controlling the naming to a certain extent, in this case the first x number of characters represented by the variable laneName, reference to the target resource becomes more manageable. As the newly created resource is successfully linked to the scene, the equation hands over the reference object to the variable that is stated on the left. Since the intended final product is a plane, the size of the perimeter has to be defined. The parameters that represent the needed dimensional values are L and W in the handler cited above. The value passed on in the L is used to set the length of the plane.

Res.length = L

This time the variable Res is used as a point of reference. Of course, Res is the reference object defined above. What the dot syntax accomplishes in this case is that the property of Res, i.e. length, is to be evaluated to L, or whatever that term signifies. Because the length is an attribute of Res the overall result is that the Res length is defined by the value the parameter represents. Needless to say, the length property is not enough to define a plane. Since planes exist in at least two-dimensional world, the object in the process of creation needs to be given a width value. That is when the W parameter comes handy. To attribute the width value to the plane resource is analogous to the operation just preceded. Using the dot syntax it is written as follows.

Res.width = W

That should be clear enough. Now that the foundation to create the plane model has been laid down, all you need to do is to create the model to build and base the final product upon. As the flow of the plane mode creation indicates, the model, which will be utilized to build the final plane model, itself will be constituted by the resources that have been generated by the preceding operations.

A model is generated by a function similar to the one used to generate a model resource. Before dwelling much upon the methodology to create a model, let me state the very coding used for that particular purpose.

Obj = scene.newmodel(planeName, res)

Here the left side of the equation is the actual operation to produce the target model. The syntax, as I already mentioned, is very similar to the one used to generate the model resource. First, the function instantiates the named plane, here indicated by the variable planeName, that is based upon the model resource referenced by res. Once the newmodel() function generates the desired model, the model is incorporated, or linked, to the 3D world represented by scene. When the linkage is complete and the model has successfully become an attribute of the scene, then the object, which now is endowed with a number of attributes, is made to be referenced by the pointer Obj. The process is smooth and logical. At this stage, however, the object thus generated has a mundane shader[10] and does appear simply as checkered entity, which is the default of the program we are dealing with here. The model has to be linked with more attributes to make itself more appealing to the viewer and more appropriate under the given context. In order to attach the new shader to the model, we need to give rise to the shader "potential" in the same manner as we did to create the source of the model we are engaged with just now. The procedure needed for that is as follows.

Shd = scene.newshader(planeName & "shd", #standard)

Here newshader() is the function needed to produce the shader potential as I mentioned above. The two parameters within the parentheses are the arguments needed to produce the shader with needed attributes in the end. The dot effectively connects the habitat for the 3D world with the new ingredient that will be pretty soon an integral part of the scene. The equation evaluates the left term to the pointer that references the corpus of the 3D world build up so far. If I may further insert an interpolative explanation, the concatenator & performs the function of combining the string held in planeName with the explicit 3-chrater string shd. The resulting string will be used to refer to the shader object later on in the handler. The symbol preceded by # indicates the type of shader, which happens to be convenient for the operation now being executed.

As I already hinted, creating a shader object itself is not enough to produce the surface quality that is associated with the convincing wrappings models are presented in. The shader needs to be further modified. The method used for that purpose is to put the value that is held in the parameter on the very first line of the current handler in property diffuse of the shader object that has been just created.

Shd.diffuse=planeColor

Here the argument planeColor has a rgb value represented in such a manner as rgb(145, 200, 78). Needless to say, the numbers within the parentheses take any value between 0 and 255. Once an actual color tint has been passed on to the shader object, it is ready to attach itself to the target models to endow them in the convincing color to the viewers. But before that step is taken it is better to remove whatever texture is already assigned to the models. The coding for that is Shd.texture = void. With the voiding of the default texture, we are really ready for attaching the shader object to the target models

[10] A shader is a surface quality 3D models assume. Let me quote from the definition given by Paul Catanese in his *Fundamentals of 3D Programming in Director 8.5*: "A shader is an element of 3D environments used to define the surface qualities of a Model, such as its color and how shiny it is." Thus, without a shader no model is able t5o assume the appearance that is convincing to the viewer. The quote is from page 869 of the book identified.

with all the properties that have been generated through the preceding operations. Then without undue delay let me state the much-awaited line.

Obj.shaderlist = shd

Here the right term represents the object that is crated by the preceding operation. The equation puts the value contained in that pointer in the attribute shaderlist, which is simultaneously linked to the model obj already generated. The property shaderlist is shorthand used to set the value of each side of the plane to the value represented by shd in case there are more faces than one in the object under consideration. Now that the shader is attached to the object, it should appear as the creator intended. Before, however, the coding ends another line is inserted to return the created object to the calling function. If you do not remember the whole set up of the current handler, it is worth reminding you once gain of the purpose of the generalization of the script we re working with now. The on creatplane handler is intended to facilitate generation of planes of various types so that every time an author is tempted to create one he can call the handler and skip all the time consuming coding work and concentrate on other important issues. In other words, the current handler is conceived as a utility to help create a construct in which one can attain an ideational interface that is freed of all conventional verbal trappings. Whether that is easily concretized in virtual space may be moot, let us proceed with that ideal as the premise of our argument.

When on createplane handler is called it is designed to return a plane model to the calling function. That procedure is achieved by the following line.

Return scene.model(planename)

Here return literally means returning the object produced by the rest of the line. As already evident by now model() is the model with all the attributes attached to the initial model a number of lines above in the script coding and the intervening dot between the scene and the function hitches the model with the 3D world, effectively incorporating the plane not the virtual space. Therefore, what is returned is the entity with all the appropriate visible presence that actually presents itself before the viewers' eyes. The lines between the handler and return Return scene.model(planename) are coded to properly respond to the demand made on he currently handler. But the demand is only met if the 3D world object here signified as scene is indeed ready for editing. If that is not the case, the alternate coding takes control and rejects the demand.

Else

Return false

Here the first else indicates the beginning of the alternate course. The next line executes the operation which corresponds to the one used to return the object in the other alternative. This time the answer sent back to the calling function is in Boolean and 0 or false, meaning the requested object has not been created due to the non-existence status of the 3D world. (If you remember, the 3D habitat or world is requisite for all the models to exist. It is the only environment they can thrive, or more strictly speaking it is the sine qua non of 3D models in the program we are dealing with.) The return operation essentially ends the current coding to build three-dimensional models. But if I may pursue the work I have initiated, there are still a few more lines to close the coding in the syntactically correct manner. Since the preceding lines are supposed to be within the conditional clause they have to be proceeded by the end-if statement. The end-if statement is followed by the concluding line in the following manner.

```
end if
end
```

Those two lines successfully conclude the plane production utility handler. Although the explanation was rather long-winded, the time the current handler will help you save creating various plane objects more than justifies it.

Now that the utility handler is fully explained, it is time to put it to the test. What is it so good for? That is to be soon made apparent. Let us consider a case in which we want to produce plane objects in quantities. Ordinarily we pay minute attention to each and every single process that is involved in generating such objects. But 3D models by definition assume a number of parameters that need to be filled every time a model is generated. Otherwise, the program complains not without justification. Because wrong or invalid values will eventually boggle down the overall attempt to create the world sketched out in the first place. The utility handler could indeed save programmers countless hours which otherwise they would most likely waste on futile efforts to fix kinks and quarks in coding here and there. But justification for such handlers has already been made. Let us demonstrate the power and usefulness of the current coding in the following manner. For instance, suppose one wants to produce a number of objects that are consistent in type but different in minutiae, the loop operation exemplified next will easily fulfill most essential needs such an exigency demands.

```
repeat with x = 1 to 20
createplane("plane" & x, 10, 20, rgb( 150, 204, 57))
end repeat
```

Of course one way to differentiate the dimensions of the models would be to randomize the hard coded values so that each time the present loop runs diverse values are generated. When the target function is called it responds effortlessly to give rise to the number of objects requested by the calling routine. In this case the number of models created will be 20 because the number of times the order is executed corresponds to the number of loops run under the dictate of value x.

The argument I have been deploying may sound rather irrelevant in light of the seamless communications I proposed in the beginning. The criticism, if there is any and I expect plenty, may well be justified. Indeed, all the methods I have explained so far that are used to generate the conditions of extra-lingual communication are merely the possible setups where the ideal interchange of thoughts and ideals could take place. If they, meaning those methods and means, simply suggest the mere tools, in other words possibilities, to initiate communications that transcend verbal exchange of conceptual significations, then they certainly need to be followed through by a corpus of propositions that will likely play auxiliary roles to achieve the ideal bi-directional flow of semantic units. That is a very reasonable argument and I completely agree with that. Therefore, the rest of this essay (unfortunately very limited in space) will be devoted to finding ways to utilize the techniques that are deemed to be capable of potentializing a communicative sphere in which unhindered smooth ideational transactions take place. But even before we go into that profound topic we are immediately forced to face a very complex task of bridging the verbal—the kind that is literally manifested through the aggregates of textual compositions—with the kind I have been primarily concerned with when I presented the codings and procedures involved in creating the presentational setup—the world that is underpinned by the sensory dictates and oftentimes situated in different conceptual coordinates. How can they really be yanked together and made to contribute to one synergistic semantic and semiotic confluence? In other words, how do they form a better cognitive, conceptual and sensory plane where the traditional intellection is amplified and accomplished to an exponentially greater extent? When we consider the implications of the extra dimensional communications the digital and programming technology has made possible, we must answer the question and must meet the challenge head on to arrive at the ideational plane that seems tantalizingly hanging out there. That purported plane may as well be virtual, as the trite phrase goes, but even if the posited dimension proves to be infinitely illusive, we must have courage and pursue the course we are daringly going to chart for ourselves. The task is not for the weak-hearted. It is only for the brave who knowingly sail into the darkness of the vast conceptual nebula by the only map he has at this relatively inchoate stage of digital technology.

Rather than spending too much time on the abstruse argument as to how to find the paths to reach human conceptual and perceptual nirvana, let us follow and continue the mundane steps I have been pursuing in the foregoing perambulation. First question I would like to pose is how does the programmatic setup accommodate for the receptive psychological state so that the presentations delivered through the digital means can find the audience truly attuned to take advantage of the nuances and ambience that may not be available in the solely textual environment?[11] Consider the following excerpt. It is from Franz Kafka's "Description of a Struggle," a rather strange tale with two complete strangers interacting with each other. The protagonist of the tale observes a policeman in the distance and imagines and describes what appears to be happening inside as well as outside himself.

> In front of a distant coffeehouse with black windowpanes a policeman let himself glide over the pavement like a skater. His sword hampering him, he took it in his hand, and now he glided along for quite a while, finally ending up by almost describing a circle. At last he yodeled weakly and, melodies in his head, began once more to skate.
>
> It wasn't until the arrival of this policeman—who, two hundred feet from an imminent murder, saw and heard only himself—that I began to feel a certain fear. I realized that whether I allowed myself to be stabbed or ran away, my end had come. Would it not be better, then, to run away and thus expose myself to a difficult and therefore more painful death? I could not immediately put my finger on the reasons in favor of this form of death, but I couldn't afford to spend my last remaining seconds looking for reasons. There would be time for that later provided I had the determination, and the determination I had.[12]

It is difficult to convey the eerie and almost dream-like atmosphere Kafka creates through careful juxtaposition of the mundane and the extraordinary and the inner- and outer-scape of the characters. Especially so if the lines quoted represent a mere fragment of the entire backdrop against which the story is based. Nevertheless and

[11] Needles to say, the nuances and ambiences I refer to here are purely subjective concepts. They may as well be biased and (mis)leading to the extent that traditionally text-bound audiences may find them objectionable. Especially so, if they have already cultivated the delicious habit of laying over subtle emotional and intellectual shades and hues over the original textural sources. But on the other hand, some of them actually enjoy the interpretational suggestions filtering through the new media. In either case, the approach suggested in this essay opens up a new dimension to the communication between the sender of a message and its receiver.

[12] The excerpt is quoted from *The Collected Short Stories of Franz Kafka*, p. 17, by Penguin Books, published in 1988.

because of that it is worthwhile to attempt to supplement and generate the ambience what the author considers is appropriate for the story. By what means? Of course through the language we have been dealing with so far. Is it really possible to augment and generate the right kind of ambience everyone agrees is right? Both yes and no. Strictly speaking, any kind of secondary interpretations, of which audio and visual supplementations are an important part, is subjective. Without the mediation of the editor no translation from the verbal text is possible. However, if a certain amount of sensory consensus is established, such secondary interpretations could become or at least be perceived as something inherent in the original text. Let us look at the concrete scripting procedures involved in generating such apt ambience that is deemed to contribute to the semantic swelling of the original text. Look at the first couple of lines of the excerpt. The policeman glides along on an icy surface and even yodels as complacently as one can be. The mood is both comedic and strange. How do we indeed render that complex combination of moods that at the same time reflect the subjective state of mind of the perceiver, the narrator/protagonist? It may be a tall order but at least we can begin.

Since we have been looking at the various possibilities 3D animations bring to the textual world, let us start by declaring a property that refers to the very world in which all kinds of models eventually inhabit.

```
Property p3D
```

That is simple enough. We follow it through with the following coding. Although we attempted to create a template that could deal with various types of 3D models, the lines enumerated below demonstrate the procedure to generate the target ambience from scratch. It is not meant to stretch out the explanation more than necessary but to elucidate the creation of that subtle and complicated interface between the audiovisual elements with the textual content.

```
On init3d(me)
P3D = sprite(me.spriteNum).member
P3D.member.resetWorld( )
P3D.camera[1].transform.position = vector(0,0,600)
BoxResource = p3d.newModelResource("boxResource", #box)
BoxTexture = p3D.newTexture("texture", #fromCastmember, member("texture"))
BoxModel = p3D.newModel("boxModel", boxResource)
BoxModel.transform.position = vector(100, 0, 0)

End
```

The first line hands over the reference of the 3D object to the pointer p3D. It is not absolutely necessary, of course, to create a pointer at this point but the procedure will save time as the object, which is indicated by the member of the referred to channel, is frequently cited throughout the current coding. The second line is an absolute necessity. Without it the world referred to as p3D will be cluttered with unwanted models and other objects that will inhabit the 3D world every time the script runs. (In reality such operation is rejected for the obvious reason that cluttering of objects will destroy the order of the virtual space.) Once the world is set in order the camera, a window on the p3D world, is positioned appropriately so that we can view the world from an appropriate distance (line 3). In this case the camera is located at 600 units in the positive units on the z-axis. The index in the square brackets refers to the default active camera that already exists when the 3D world is initialized. The next line is used to initialize a resource to build the models upon to ultimately constitute the world we have been trying to link with the text cited above. We happened to choose the box type here, indicated by the symbol #box. Next we generate a texture from a prepared texture file here conveniently named "texture." In reality a texture does not have to be prepared beforehand. It can be created while the script runs, or at runtime. But the texture that contributes to reality and verisimilitude has to be carefully prepared and elaborated upon. That is why a member cast is used in this script. Next, we create a box model from the resource that has been initialized above (line 6). The reference to the model is made by the second parameter of the newModel( ) function. The next line positions the model to the location deemed appropriate for the purpose we have been working. In this case 100 units in the positive direction on x-axis.

The current handler, however, is not complete as it is. It needs to incorporate more lines to make the world a habitable sphere for all the things to be generated and exude the right mood to match and supplement and augment the ambience the textual world is perceived to contain. The lines for the immediate purpose of creating the fundamental elements to let the viewers at east perceive the objects in front of their eyes would be written as follows.

```
BoxShader = p3D. newshader("boxShader", #standard)
Boxshader.texture = boxTexture
BoxModel.shaderList = boxShader
```

The above coding is nothing other than generating a shader from the texture created above and applying it to the model in question, here indicated by the referent boxModel. Now that all the preliminary procedures are complete we can safely add the product of our labor to the 3D world.

```
BoxModel.addToWorld()
```

However, adding the model to the world is only a step in a further complicated process of building a general setup in which our purported ambience will be readily perceived by the viewer of that 3D world. In order for that general scheme to function there needs to be more elaborations on the general structure, of which the preceding coding is only a part. Let us assume that it is best to trigger the generating of the 3D world manifested in the preceding example when a certain laps of time is detected after the general presentation starts, be it a fraction of a second or a few seconds whichever is deemed convenient. In this is the cue is triggered immediately after the presentation starts.

```
On beginSprite(me)
Me.init3D( )
End
```

The handler here demonstrated acts upon a cue that is issued when the current object embedded with this very same handler is detected to be called upon (or for accuracy's sake it is better put that when the object is detected to be used by the presentation). Next comes a rather esthetic maneuver. After all we are embarking on this perilous journey to add a new dimension to the text that may be adequately compensatory to the overall mood of the textual world as a whole. To give the mysterious as well as somewhat eerie atmosphere the protagonist of Kafka's story perceives in the distance, the newly generated model is given a spin. First the coding.

```
On prepareFrame(me)
Me.update3d( )
End
```

This handler calls update3d( ) to implement the rotation. The purpose of this handler then is to intercept the message that is issued every time the playback head moves across frames and transmit that t the next level of coding. Needless to say, the above handler is to be attached to the as-yet-not-born world object, or the channel that object will occupy. Now, the handler that is waiting for the message sent from the prepareFrame interceptor is described as follows.

```
On update3d(me)
V = vector(0, 2, 0)
P3d.model[1].rotate(v, #world)
End
```

The first line defines the vector that is used to rotate the model around the world axis in the second line. When you play the animation the model turns around 2 units on the y every time the playback head pass through the boundary that demarcates that particular frame. Of course the speed of rotation can be changed according to the needs. All those cosmetic flourishes are to be adjusted to fit the ambience that was initially envisaged. One model we created here s not going to be the only resident that will inhabit our 3D world. But for the moment let us focus on the sound aspect of our overall setup.

To make it simple, let us plunge straight into the sound playing mechanism by declaring the coding in the following manner.

```
On initSound(me)
Sound(1).queue( [#member: member("Ksound"), #loopCount: 0])
Sound(1).play( )
End
```

The number in parentheses represents the channel the sound will be played in. The function added to after the sound channel declaration prepares the sound object to store the named sound member in memory. In the next parameter the function is supplied with the number of loops the sound object to pay the sound when it comes to time to actually play the sound. The next line completes the cycle by sending command to trigger the prepared sound. In order to throw the given handler into action, another handler is required that intercepts the message when the playback head passes through a certain position in the entire animation. In this case when the 3D world is initiated, that is when the animation primarily based on the 3D constituents starts.

```
On beginSprite(me)
Me.init3d( )
Me.initSound( )
End
```

However, allowing the sound to continuously playing may not be a good idea if the ultimate purpose of the whole coding is to create the maximum rendering of the textual atmosphere through multimedia. With that noble objective in mind let us modulate the sound as it plays during the animation. How to is the question we have to wrack our heads for a moment. Suppose we decided to settle on panning and volume gain modulation. The following argument is an elaboration based on that assumption. The panning effect can be achieved by mapping the model position on the x-axis and projecting its relative value to the current pan in the range between –100 and 100. The actual panning value can be set as follows.

```
Sound(1).pan=p3D.model[1].worldPosition.x
```

This is not so complicated. The model's worldPosition[13] value is directly translated (actually the units on the given axis) to the left-right pan the interactive player perceives as he experience the multimedia aided textual space. To adjust gain we map the z position of the model relative to its world position to the value of the volume. However, because volume shifts in the range between 0 and 255, the value obtained in the previous method has to be readjusted to reflect that contingency. The coding to accomplish it can be described as follows.

```
Sound(1).volume = (p3D.model[1].worldPosition.z + 100) * (255.0/200)
```

The final parenthesis is added to readjust the volume in the range specified above. All you need to do now is to insert the two lines that pertain to the pan and volume properties. As for the interactive mechanism to reflect the mouse move on the part of the user, the following calling function needs to be inserted in the prepareFrame handler.

```
Me.updateSound( )
```

That is just about the end of the general scheme. However, we can give a little twist to the whole 3D interactive animation. One strategy would be to rotate the cube according to the relative position of the mouse. The ever-shifting vector can be created from the mouseH and the mouseV that the system allows the editor to obtain automatically. First we arrive at the width of the stage, i.e. where the action is taking place.

```
The stageRight – the stageLeft
```

What we are aiming at is to map the mouse coordinates so that they coincide with zero right at the center of the stage. The coding for that can be described as follows.

```
The mouseH – ((the stageRight – the stageLeft)/ 2)
```

At this point we remap the vertical mouse value to the x rotation of the object and the horizontal mouse value to the y rotation of the object. In order to execute the series of operations I have presented, you need to substitute the original v = vector(0,2,0) in the update3D handler with the following.

```
X = the mouseV – ((the stageBottom – the stageTop)/2)
Y = the mouseH – ((the stageRight – the stageLeft)/2)
```

[13] The worldPosition s a vector position an object is presumed to hold relative to the center of the 3D world space. The concept is opposed to the local or object position which is hypothetically defined as the center of the object itself. Although they are convenient concepts in describing the relative positionality of each model as it resides in the virtual 3D world they are by no means absolute defining terms to fix models in a certain quadrant in the virtual universe. What is complicating about the 3D space is that nothing is absolute and everything is what it seems like depending on how you look at them. In other words, one view of an object is as true as another of the same.

Z = 0
V = v/20.0
Here the final line is intended to scale down the speed at which the object is to rotate and at the same time the rate of modulation at which the sound is to be played.

Needless to say, the scheme I have proposed here is a very basic one that is hardly sufficient in real interactive situations. But the setup in its simplicity should be enough to evoke the kind of text-multimedia synergy envisioned in the beginning of this essay. The simplest way to experience the effectiveness of this approach would be to play the animation according to the plan adumbrated in each coding. Unfortunately, that is only to be accomplished in the multimedia environment, which this essay is just an extension rather than the other way round. If that is a grave disappointment for many who have pursued this project on the pages up to this point, do not let that discourage you from envisaging that marvelous space in which the textual nuances are finely interleaved and greatly enhanced by the interactive digital elements made alive by various scripts and sounds and graphics where the meanings that may have been hidden between the interstices of various shades of phrasing suddenly jump at the interactive participants. Just imagine! If this traditional approach is not the very best, it may perhaps be the second best. Perhaps one of these days you will be able to tune in to the ineffable world of Kafka and the iridescent and "negatively capable" sphere of Shakespeare. Then, if not without a modicum of bafflements in your knitted brows but at least with the eyes shining with excitement, you will have finally glimpsed into the arcane private world which until then has never revealed its jealously guarded pleasures.

References

Allenson, Andrew, et al. *Director 8.5 Studio.* Birmingham, United Kingdom: Friends of ED, 2001.

Catanese, Paul. *Director's Third Dimension.* Indianapolis: Que, 2002.

Couturat, Louis. *Opuscules et Fragments Inedits de Leibniz.* Hildesheim, Germany: Olms, Georg Publishers, 1988.

Glatzer, Nahum N., ed. *The Collected Short Stories of Franz Kafka.* New York: Penguin Books, 1988.

Gottesman, Ronald, et al. *The Norton Anthology of American Literature.* 2 vols. New York: W.W. Norton, 1979.

Harbage, Alfred, ed. *William Shakespeare: The Complete Works.* New York: The Viking

Press, 1975.
Rosenzweig, Gary. *Advanced Lingo for Games.* Indianapolis: Hayden Books, 2000.
Wierzbicka, Anna. *Semantics, Culture, and Cognition.* Oxford: Oxford University Press, 1992.

文字とディジタルメディアとの概念的シナジーについて

伝統的な印刷物によるテキストの総合体で表された概念的まとまり、ないしは文学的記述はそれ自体で独立した意味的空間を構成してきた。そのようなテクスチュアルエンティティーにディジタルメディアがどのように入りこんでいけるか、というのがこの論文の出発点になっている。そしてもし両者の融合が aesthetic ならびに semantic なレベルで可能ならばそれはどのような具体的な手段で実現できるのか、というな hypothetical　問題を proprietary な言語である Lingo を使用して議論してみた。勿論、ディジタルメディアならびにそのインターアクティブなテクスチュアルコンテントへの合体は厳密にいうとディジタルな環境でしか具現することはできないが、テクスチュアルな世界に限定されたこの論文の中で最大限に両者のシナジスティックな融合の可能性を読者に把握してもらえるよう可能な限り詳細なスクリプトを交えながら議論を展開してみた。