1. PlayStation Ordering Table Tutorial

The PlayStation GPU is designed to walk a linked list of graphics primitives and draw each one in sequence until it reaches the end of the list. There are many ways such a list can be constructed. The PlayStation GPU libraries use a method known as an "Ordering Table" because it provides a means of manipulating such a display list in a flexible yet easy to use fashion. This method is ideal for 3D graphics, but is also equally applicable to traditional 2D graphics situations where the drawing order is important.

This document will give you a basic understanding of how PlayStation ordering tables work. A basic understanding of 2D and 3D graphics and programming principles such as linked lists are required.

Note that there are three separate libraries for the PlayStation that deal with ordering tables. LIBGPU provides most of the basic functions for manipulating ordering tables. LIBGTE also has a few functions which place objects into a specified ordering table. LIBGS also deals with ordering tables and attempts to automate some of the tasks that would otherwise be done through a combination of other calls.

We will be discussing only LIBGPU functions in this document. LIBGTE only uses ordering tables when performing a subdivision of a large polygon object into multiple smaller ones, which are then automatically placed into the specified ordering table. We'll cover LIBGS in a future installment.

1.1. What is an Ordering Table?

An ordering table (OT) is a linked list data structure designed to group together graphics primitives for a 3D display (i.e. polygons, line segments, sprites, etc.) according to their Z-depth. When the list is used to draw the primitives, the ones farthest away from the display viewpoint will be drawn first, the closest ones will be drawn last, and all the intermediate distance ones will be drawn when appropriate. The result is a correctly rendered 3D display.

The OT linked list uses a fixed array (either dynamically or statically allocated within your program) as an anchor point for each possible Z-depth value, which allows primitives to be easily inserted into the list at specific Z-depth offsets without having to walk through the list link by link. The size of the array controls how many different Z-depths can be used. To keep things simple throughout our discussion, let's assume that we have a small OT with entries for 10 different levels of Z-depth. This is defined as:

```
u_long      theOT[10];
```

The size of the array depends mainly on how much memory you wish to devote to it. If you use too large an array, you end up wasting memory, but if your OT is too small, the result that primitives may be drawn in the wrong order for a correct 3D display.

An alternative to having a huge main array is to use multiple OTs. This is done in order to provide extra Z-depth resolution at those levels where it is needed. This is a useful technique, but for now let's assume we just have a single basic OT. We'll come back to this idea later.

After performing 3D calculations on each primitive, the resulting Z-depth value is scaled to the range of available entries in the OT (i.e. scaled from 0-32767 to 0-9 for this example) and the result is used as an index into the OT array when the primitive is added to the list.

Finally, once all the required primitives have been added to the list, they can be drawn with a single library call. If you have built the list correctly, everything will be drawn in the correct sequence to create a glitch-free 3D display.

1.2. Initializing An Ordering Table

Before primitives can be added to an OT, the table must first be initialized. This is done using either the ***ClearOTag()*** or ***ClearOTagR()*** functions. They will manipulate the list as follows.

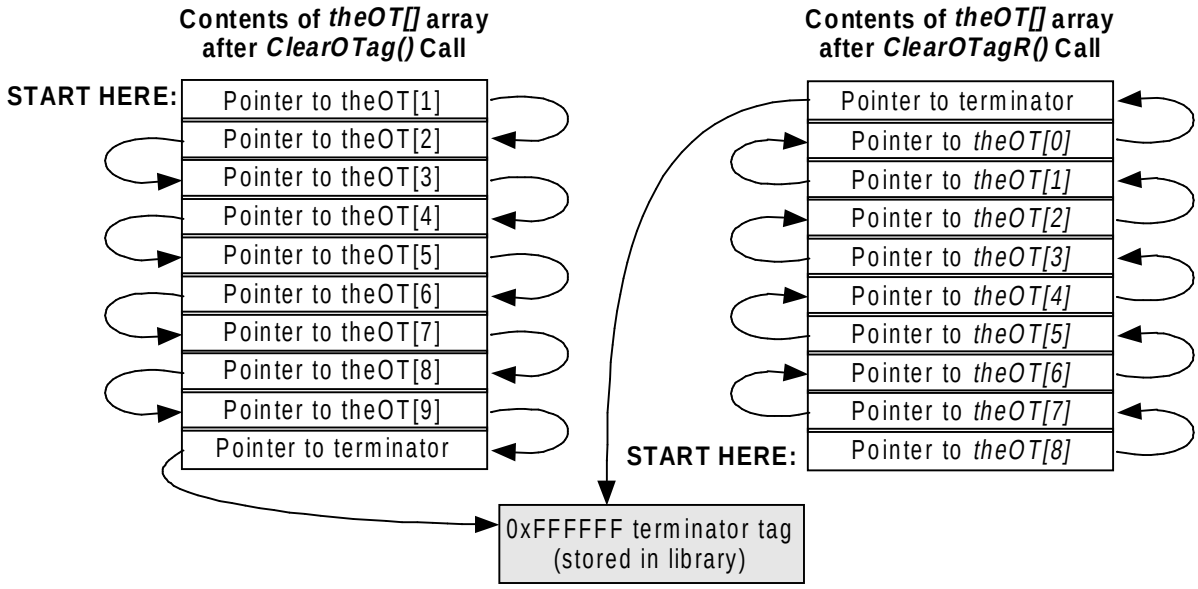

Figure 1

As you can see, these functions turn the empty OT array into a linked list. Note that the last entry of the array for ***ClearOTag()*** and the first entry for ***ClearOTagR()*** is a pointer to a special terminator primitive in the library. The PlayStation libraries and hardware need some way of knowing when they've reached the end of the list, so a special value of 0xFFFFFF has been designated to indicate this. When this value is found in the lower 24-bits of the 'tag' field of a primitive, that indicates that the end of the list has been reached.

Using the ***ClearOTagR()*** function results in an OT that is linked from the highest element going down, rather than from the lowest element going up. This results in an OT designed to be walked in reverse order. We'll discuss this again later in section 1.8.

1.3. Adding Graphics Primitives To An Ordering Table

Now we can add primitives to the OT using the ***AddPrim()*** function. This function takes as arguments a pointer to an OT array element and a pointer to a graphics primitive:

```
void AddPrim( unsigned long *ordering_table, unsigned void *lpGrPrim );
```

If you have only 2D graphics where everything will be drawn at the same depth, then the *ordering_table* parameter can simply be the address of a pointer rather than an array. For 3D graphics (or psuedo-3D like a 2D display with parallax scrolling), the parameter should be a pointer to the element of the OT array that represents the Z-depth of the primitive. For example, let's assume that our 3D calculations have determined the Z-depth of a polygon being drawn to be 16762 (of 0-32767). Our OT doesn't have 32767 elements, so we must scale that down to the range of 0-9. This gives us a new Z-depth value of 5. To add this polygon to the OT, use:

```
AddPrim( &theOT[9-zdepth], lpGrPrim1 );
```

where *zdepth* is the Z-depth value of the polygon (5) and *lpGrPrim1* is a pointer to the polygon structure.

The reason we use *9-zdepth* is so that the farthest objects (those with the highest Z-depth values) will be placed in the beginning of the OT. This way, when we draw primitives starting at the beginning of the table, they will be drawn first. (Alternately, a reversed OT can be used, which we'll discuss in section 1.8.)

If the table was initialized with ***ClearOTag()*** (unless otherwise specified, this is assumed for all further examples) it would look like this:

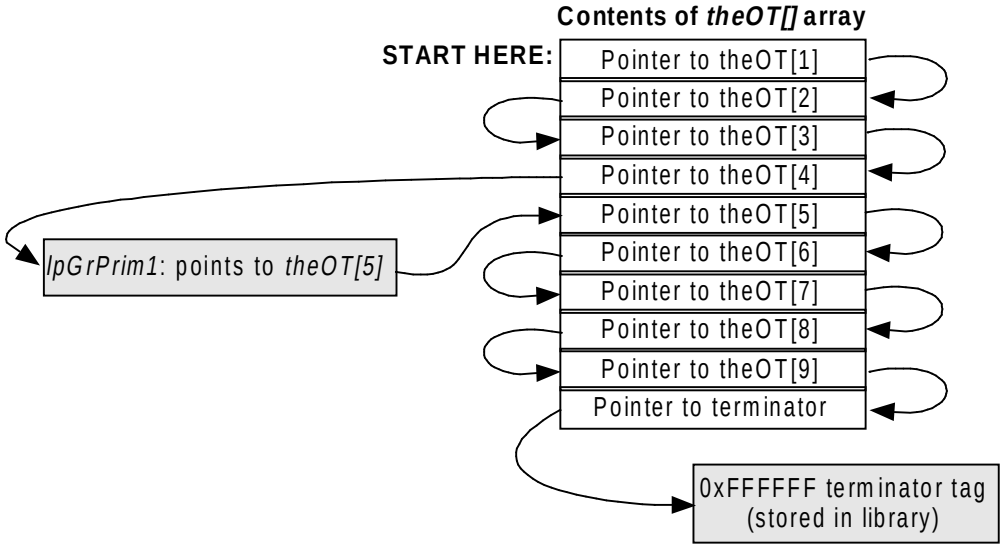

Figure 2

Note that we only show up to three primitives at each depth just to keep our example easy to follow. A real program can have as many primitives linked together at same depth as will fit into available memory.

Note that the same primitive cannot be added to an OT more than once. Doing so will accidentally delete other primitives from the linked list. If you need to have multiple copies of the same primitive, make copies of the primitive structure and add each one separately.

Adding a second primitive at the same depth as the previous one would result in this:

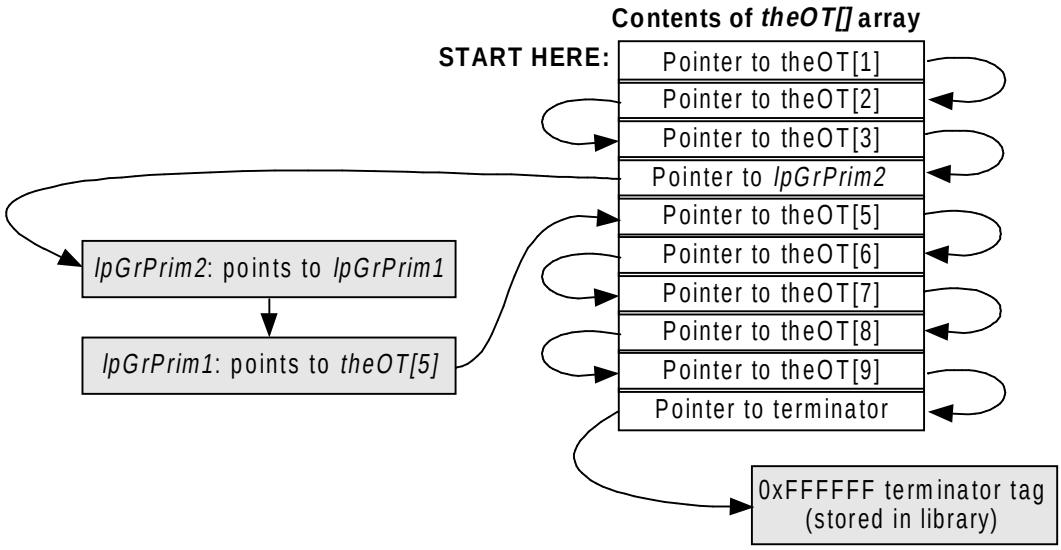

Figure 3

At this point, let's add several more primitives. We'll do two at depth 7 (OT level 2), one at depth 1 (OT level 8), two at depth 2 (OT level 7), and one more at depth 5 (OT level 4). This would result in this:

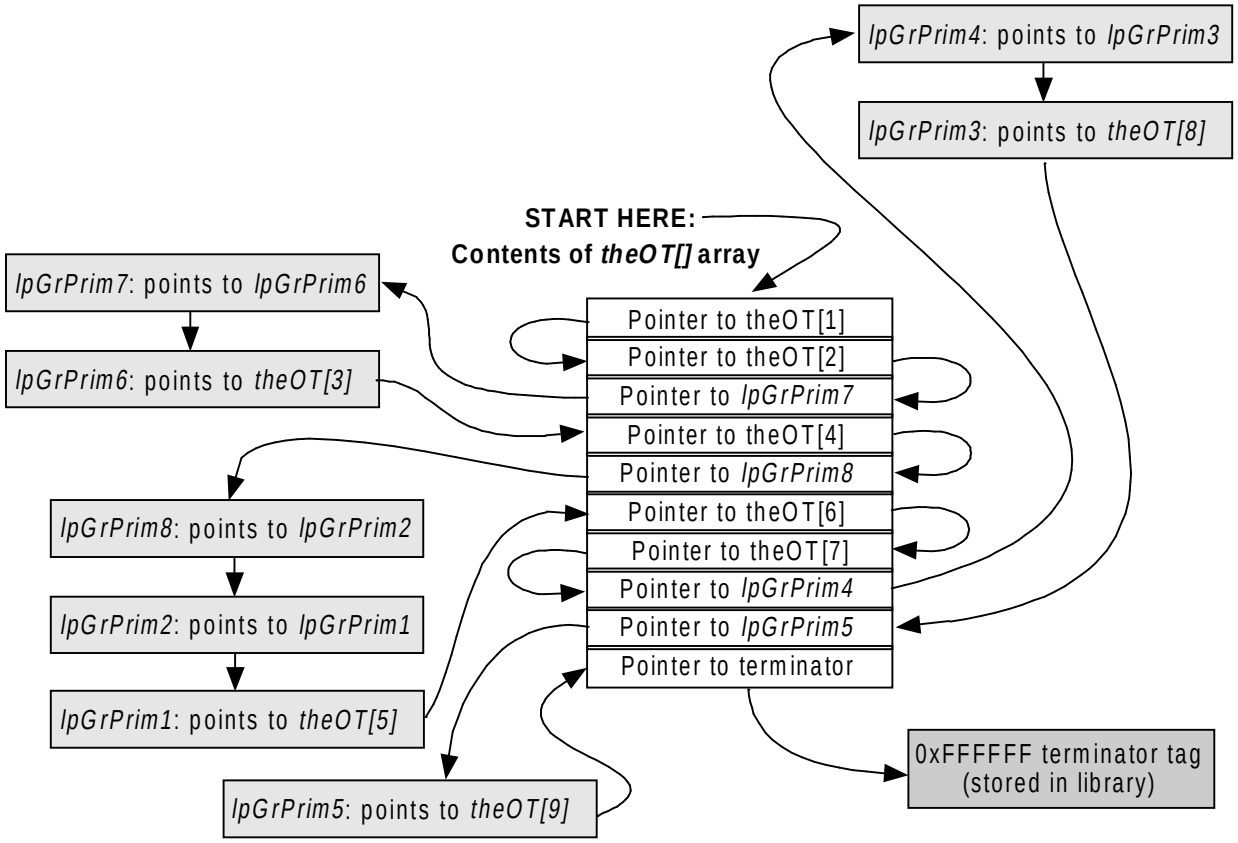

Figure 4

Note that if you start at the beginning of the OT and follow each pointer to the next item, you'll pass through each element of the OT array and through each primitive which we have added. This is how the PlayStation uses the OT to draw a list of graphics primitives.

1.4. Drawing An Ordering Table

To draw the list of primitives in an OT, use:

```
DrawOTag( u_long *ordering_table );
```

where *ordering_table* is a pointer to the starting point within your OT. Usually this is the beginning of your OT array, but for a reversed OT, it will be a pointer to the end of the array.

The PlayStation will start at the specified position and walk the list of graphics primitives, drawing each one in sequence. After each is drawn, it will link to the next item in the list and continue until the special terminator tag is found, indicating the end of the list.

The ***DrawOTag()*** function is non-blocking, meaning that once the PlayStation GPU hardware has been aimed at your primitive list and drawing begins, control will be returned to your program while drawing continues.

To determine if drawing is finished, you can use the ***DrawSync()*** call. Otherwise, an interrupt handler can be installed using the ***DrawSyncCallback()*** function. The callback routine will be executed once the drawing operation is completed.

1.5. Nested Ordering Tables

When the resolution of an OT is too low, elements of the display may be drawn in the wrong order. Using our 10-level sample OT, if one item should be at depth 4.5 and another at 4.8, they will both end up at depth 4 and be placed into the same level of the OT. Since they're at the same level, they'll be drawn in the order they are added, rather than according to their relative Z-depth values.

Fortunately, in most cases when doing 3D graphics, you'll have an OT with more than 10 entries. However, simply increasing the size of your OT isn't always the best way to improve the situation. If your main OT has 1000 levels, most of them may go unused the majority of the time, while some levels may have several hundred primitives each.

The trick is to use nested OTs. This allows you to use higher resolution Z-depth values where they will do the most good without wasting memory space. Suppose our earlier example had 100 different polygons at depth 3. Because they are all grouped together in no special order, we see some glitches on screen when we draw them. This is a good example of when using a nested OT is a good idea.

Our main OT (which we'll call OT1) takes the original Z-depth value from the 3D calculations and scales it from the range of 0-32767 down to the range of 0-9. But this doesn't give us enough resolution in the area where most of our objects are located, so we'll create a second OT (which we'll call OT2) with room for 200 entries. Now whenever we get a Z-depth value between 19660-22935, we'll place that primitive into OT2 instead of OT1.

We have a range of 3276 Z-depth values, but only 200 levels in OT2. Why not use 3276 levels instead? Keep in mind that we may not actually have that many different primitives to worry about. Depending on available memory and how many primitives will be going into OT2, then we might do more or less entries.

The first thing we do when we get a primitive for OT2 is subtract 19660 to translate the Z-depth value to the range of 0-3275. Then we'll scale this down to the range of 0-199 and add it to the appropriate level of OT2, using the same methods we discussed earlier for OT1.

Once we are done adding all of our primitives to either OT1 or OT2, then we need to link the two tables together.

1.6. Linking Multiple Ordering Tables

Linking in an additional OT is done via the ***AddPrims()*** function. Let's assume that OT2 has been defined as:

```
u_long      anotherOT[200];
```

Then it would be linked into the appropriate part of the original OT as follows:

```
AddPrims( &theOT[9-3], &anotherOT[0], &anotherOT[199] );
```

Remember that OT2 contains all of the primitives that would go into Z-depth level 3 of the main OT, so we must add it at position *9-3* to position it correctly. This would give you:

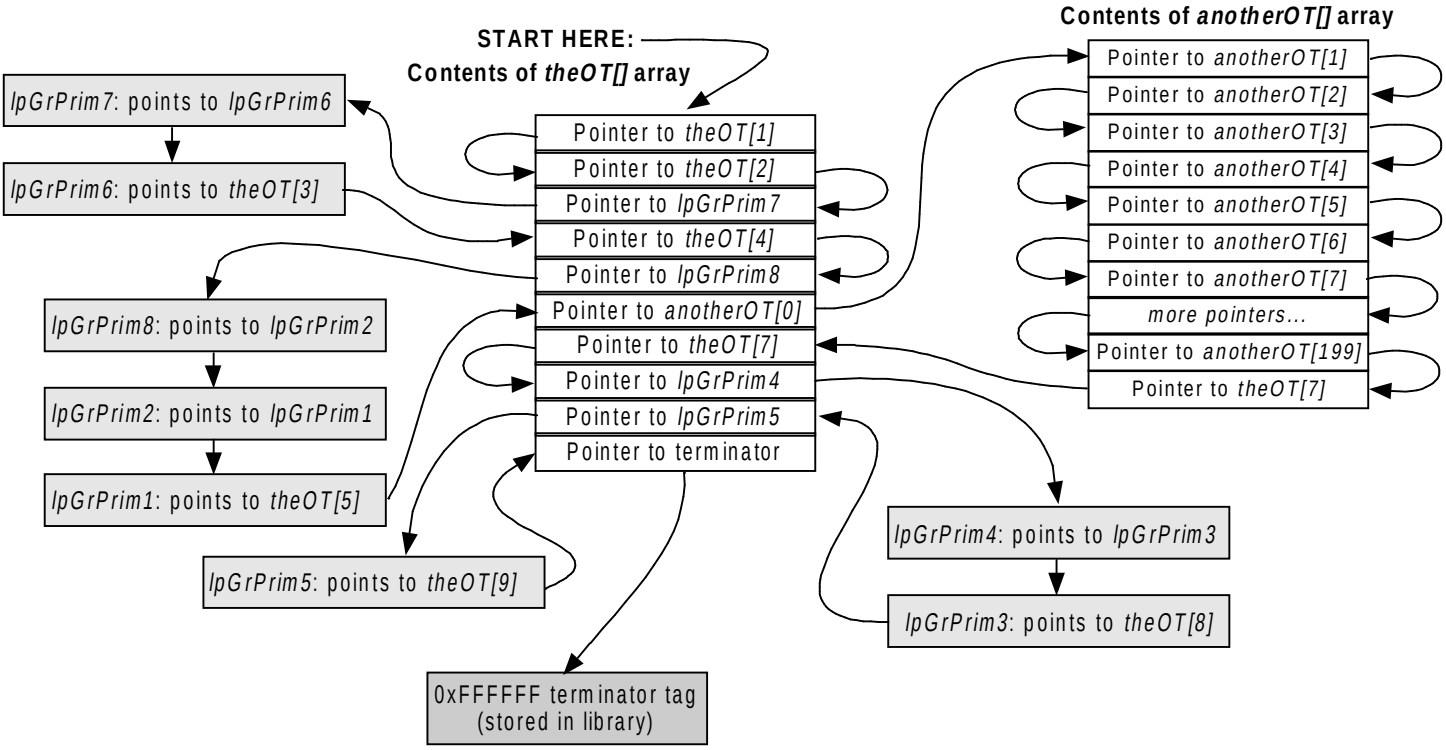

Figure 5

1.7. Notes About Additional Ordering Tables

Memory permitting, you can have as many OTs as you want. Anywhere there is extra resolution required for your Z-depth values is a good candidate. For example, separate OTs may be used for the polygons which make up individual 3D objects. Or you may have a 3D control panel overlaying the rest of your screen which is not changing position or orientation, so you could create a separate OT for it that doesn't need to be rebuilt from scratch every frame.

Always remember that all of your OTs must be initialized via ***ClearOTag()*** or ***ClearOTagR()***.

If you will be linking in an additional OT at the end of your main table, you may wish to allocate an extra element to be reserved for the pointer to the list terminator. That is, for 10 levels of depth, allocate 11 elements in the main array. Then when you call ***AddPrims()*** to add in additional tables at the end of your main table, the last parameter would be a pointer to this element. This will ensure that the additional list is added after all of the primitives in the main list, even those at the maximum depth.

1.8. Why Use Reversed Ordering Tables?

Earlier we mentioned using the ***ClearOTagR()*** function and saw the different way that it initializes an OT. This brings up the question of why would you want to do this? The main reason is that in a reverse OT, higher-level Z-depth values correspond to higher-level entries in the OT. This is a bit more straightforward and easier to follow. For example, if the OT in the example in section 1.3 (*"Adding Graphics Primitives To An Ordering* T") had been initialized using ***ClearOTagR()***, then to add a primitive we would have used:

```
AddPrim( &theOT[zdepth], lpGrPrim1 );
```

Note that we specified *zdepth* this time instead of *9-zdepth*. Once again, *zdepth* is the Z-depth value of the polygon (5) and *lpGrPrim1* is a pointer to the polygon structure. However, this time we used *zdepth* directly

instead of *9-zdepth* because we are placing the farthest objects (those with the highest Z-depth values) at the end of the OT rather than the beginning. Therefore, another small advantage to using reverse OTs is that you avoid a subtraction each time you use ***AddPrim()***. That may be not be a huge optimization, but it doesn't hurt.

If you initialized your table with ***ClearOTagR()***, and added primitives using the Z-depth value as shown above, then the example in Figure 4 would look like this instead:

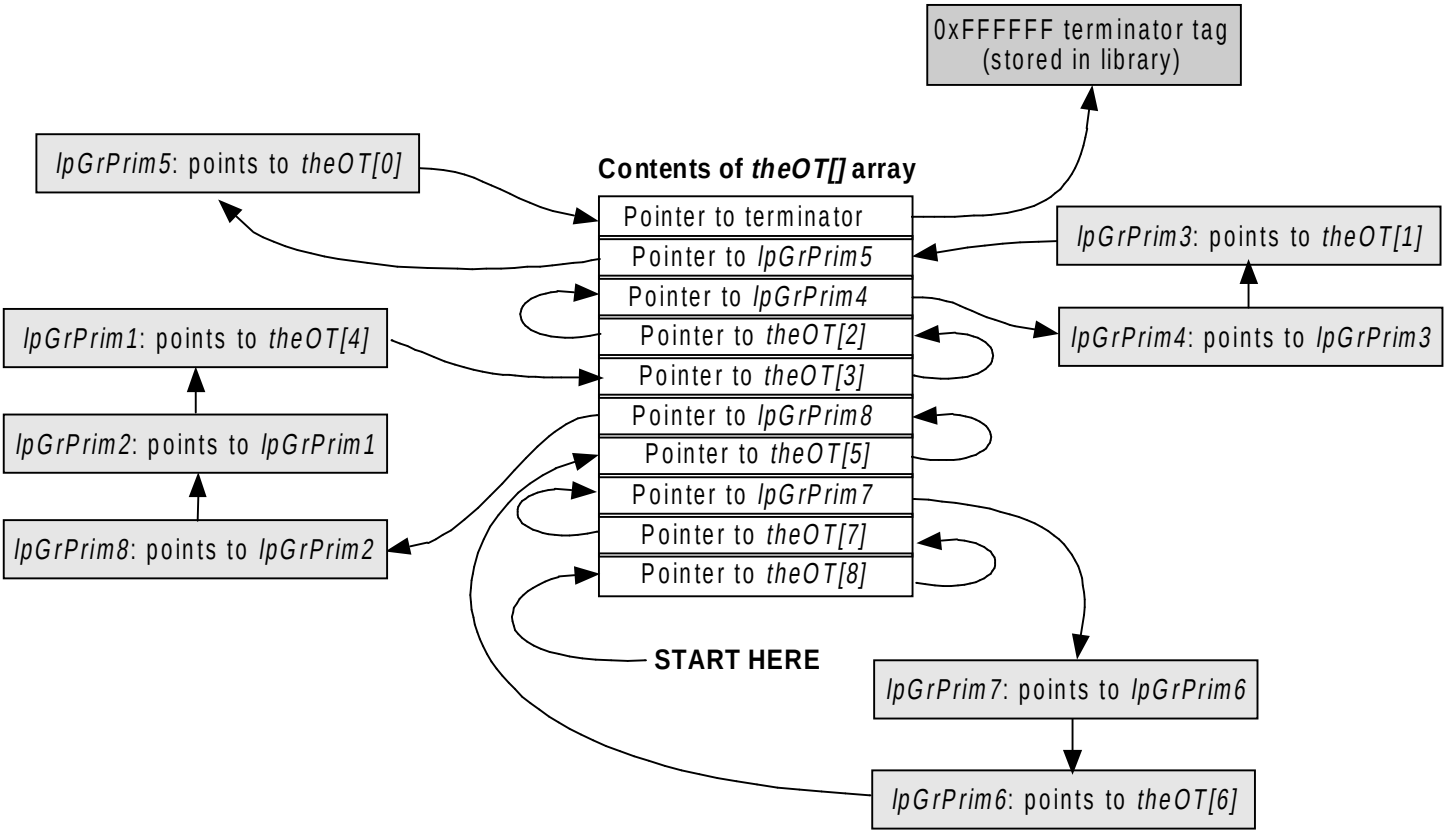

Figure 6

Later when we draw the primitives in the OT, we'll start at the end and work our way down:

```
DrawOTag( &theOT[9] );
```

Note that if you start at the position specified in the ***DrawOTag()*** function and follow the list primitives and pointers shown in Figure 6, and then do the same for Figure 4, then you will step through each of the primitives in the same order both times. The only things that are different are the links to the *theOT[]* array.

In section 1.6, we discussed using a secondary OT and linking it into our main OT. Linking a reversed OT into another table is done using the same method, except that the last two parameters to the ***AddPrims()*** function are reversed:

```
AddPrims( &theOT[9-3], &anotherOT[199], &anotherOT[0] );
```

1.9. Library Functions for Manipulating Ordering Tables & Primitive Lists

We've already discussed the functions that will be used most often, there are other library functions available for manipulating ordering tables and primitive lists (note that they aren't quite the same thing… a primitive list doesn't have to be based on an ordering table). We'll give a brief description of the main ones. For more comprehensive information, please see your ***Library Overview*** and ***Library Reference*** manuals

AddPrim(u_long *OT, void *p1) — This function links the primitive specified by *p1* into an OT at the element pointed to by *OT*. The existing link at *OT* is copied to *p1* and then a pointer to *p1* is moved into *OT*.

AddPrims(u_long *mainOT, u_long OTstart, u_long OTend) — This function links in a secondary OT into a primary OT. The *OTstart* parameter specifies the beginning of the array containing the secondary OT. The *OTend* parameter specifies the end of the array containing the secondary OT. Note that if a reverse OT is specified, then these two parameters should be reversed.

CatPrim(void *p1, void *p2) — This function links the primitive specified by *p2* to the primitive specified by *p1*. Any existing link from *p1* to something else is lost. The link from *p2* to anything else is not changed.

ClearOTag(u_long *OT, u_long len) — This function initializes an ordering table array so that each element points to the following element, except the last, which points to a terminator.

ClearOTagR(u_long *OT, u_long len) — This function initializes an ordering table array so that each element points to the previous element, except the first, which points to a terminator.

DrawOTag(u_long *OT) — This function executes each entry in the list of primitives specified by the *OT* parameter. The *OT* parameter is normally a pointer to the first entry of your primary ordering table array (or the last entry for a reversed ordering table).

DrawSync() — This function is used after ***DrawOTag()*** to detect or wait for drawing to be completed. See the Library Reference also for the ***DrawSyncCallback()*** function.

MargePrim(void *p1, void *p2) — This function is used to merge two separate primitives into a single primitive. The two primitives are expected to be contiguous in memory. The *length* field of *p1* is adjusted to include the length for *p2*. This is usually used to merge a primitive like DR_LOAD with a sprite or polygon primitive.

NextPrim(void *p1) — This function returns the address link of the specified primitive. That is, if primitive 1 links to primitive 2, then this function returns a pointer to primitive 2.

TermPrim(void *p1) — This function sets the link field of the primitive *p1* to point at a primitive list terminator value. Any existing links are lost.

This is a work in progress. Please address comments, questions, or complaints to:
Mike Fulton @ Sony Computer Entertainment America
Phone: (415) 655-5999 • Fax: (415) 655-5511
EMAIL: mfulton@interactive.sony.com