Game Engine Programming

SDK Version 1.0

Contents

Code Examples

Figures

Tables

1 Using runtime libraries

In this document, we focus on game engine programming for the Character Pipeline. This first chapter provides simple examples to foster a basic understanding of the Character Pipeline as quickly as possible. The examples are split up on a per-library basis, and they attempt to illustrate the basic functionality of each library as simply as possible. These examples should be studied by anyone wishing to use the Character Pipeline runtime libraries.

1.1 Libraries and prefixes

Each library in the Character Pipeline has a function and data structure prefix associated with it.

Library	Prefix
geoPalette	**GEO, DO**
texPalette	**TEX**
actor	**ACT**
animBank	**ANIM**
control	**CTRL**
Light	**LIT**
shader	**SHDR**

Table 1 Library prefixes

1.2 Source code

Source code for each of the runtime libraries can be found in each library's respective subdirectories under

`/cp/build/libraries`.

The header file for the runtime libraries is

`/cp/include/charPipeline.h`,

which includes individual library header files from

`/cp/include/charPipeline/*`.

1.3 File cache

The Character Pipeline contains a very simple file cache that, when initialized, will prevent multiple copies of files from existing simultaneously in memory. This file cache simply checks to see if a requested file already exists in memory, and if so, returns a pointer to the file in memory rather than loading it off of the disk. This file cache is initialized by calling `CSHInitDisplayCache`. Once the cache is initialized, all Character Pipeline file-loading calls will behave in a cached manner. We strongly recommend that this cache be initialized for optimal Character Pipeline use.

1.4 Geometry

The Character Pipeline Geometry Palette format (GPL) is designed to encapsulate basic geometry creation and display functionality. A geometry palette is defined as a collection of one or more display objects. A display object is comprised of the following components which tell the NINTENDO GAMECUBE (GCN) hardware how to display the geometry:

- Geometry data (positions, normals, colors, etc.).
- Connectivity data (i.e., information on how the geometry data is related to form some geometric object).
- State.

A display object does not support any notion of animation or hierarchy. These features are covered later in sections 2.4 and 2.3, respectively.

The following section presents simple examples of how to load, display, manipulate, and free display objects and geometry palettes.

1.4.1 Loading, displaying, and freeing a .gpl file

```
void Main ( void )
{
    DODisplayObjPtr dispObj = 0;  // Declare a Display Object pointer
    GEOPalettePtr pal = 0;        // Declare a Geometry Palette pointer

    // Initialize GCN here

    // Load the Geometry Palette off of disk
    GEOGetPalette(&pal, "test.gpl");

    // Create an instance of the Display Object "test object"
    DOGet(&dispObj, pal, 0, "test object");

    // Do game loop here
    {
        .
        // Render the Display Object – don't use any lights
        DORender(dispObj, cameraMatrix, 0);
        .
    }

    // Release the instance of the Display Object
    DORelease(&dispObj);

    // Release the Geometry Palette
    GEOReleasePalette(&pal);
}
```

Code 1 Loading, displaying, and freeing a .gpl file

Code 1 loads the geometry palette file "`test.gpl`" from the disk, then unpacks it into the geometry palette pointer *pal* with a call to the `DOGet` function. An instance of the display object, called "`test object`", is then created with a call to the `DOGet` function.

The display object is rendered by calling `DORender` and passing it a pointer to the display object and a camera matrix. The last parameter specifies the number of per-vertex lights affecting the given display object. Since we are not using any per-vertex lights, in this example the parameter is set to 0. Per-vertex lights are discussed in section 1.8.1.

Finally, once the game loop has exited, the display object is released with a call to `DORelease`, and the geometry palette is released by `GEOReleasePalette`.

1.4.2 Controlling a display object with a matrix

```
Mtx m;
u32 angle = 0;

//game loop
{
    // Build a matrix to rotate the Display Object around the Y axis
    MTXRotDeg(m, 'y', (f32)angle);

    // Set the Display Object's world matrix
    DOSetWorldMatrix(dispObj, m);

    // Render the Display Object
    DORender(dispObj, cameraMatrix, 0);

    angle++;
}
```

Code 2 Controlling a display object with a matrix

Code 2 uses the GCN Matrix-Vector library (MTX) to create a rotation matrix that rotates a display object incrementally around its *y*-axis. Once the matrix for a given frame is created, `DOSetWorldMatrix` sets it as the display object's world matrix and passes the display object pointer and the matrix as arguments.

1.4.3 Instancing a display object

```
void Main ( void )
{
    Mtx m;
    u32 angle = 0;
    DODisplayObjPtr dispObj1 = 0;
    DODisplayObjPtr dispObj2 = 0;
    GEOPalettePtr pal = 0;

    // Initialize GCN here

    // Load Geometry Palette "test.gpl"
    GEOGetPalette(&pal, "test.gpl");

    // Create an instance of "test object"
    DOGet(&dispObj1, pal, 0, "test object");

    // Create a second instance of "test object"
    DOGet(&dispObj2, pal, 0, "test object");

    // Do game loop here
    {
        // Rotate dispObj1 and dispObj2 differently to show they are independent
        // instances of the same Display Object

        MTXRotDeg(m, 'y', (f32)angle);
        DOSetWorldMatrix(dispObj1, m);
        DORender(dispObj1, cameraMatrix, 0);

        MTXRotDeg(m, 'x', (f32)angle);
```



```
        DOSetWorldMatrix(dispObj2, m);
        DORender(dispObj2, cameraMatrix, 0);

        angle++;
    }

    // Release both instances of the Display Object
    DORelease(&dispObj1);
    DORelease(&dispObj2);

    // Release the Geometry Palette
    GEOReleasePalette(&pal);
}
```

Code 3 Instancing a display object

Code 3 creates two instances of the same display object using the `DOGet` function. These two display objects are then animated differently to show that they are two independent instances of the same display object. When objects are instanced, their display data is shared; however, their runtime information, such as position and orientation, are not.

1.5 Texture

The Character Pipeline Texture Palette format (TPL) is designed to encapsulate multiple texture images in a single file. The texture images in the file can be of any format supported by The GCN, and textures within the same file can have different formats. The GPL format references texture palettes for display objects to which textures have been applied. In the case of the geometry palette files, the loading of texture palettes is automated. The examples in this section are for developers who wish to use the texture palette format to build their own geometry formats.

1.5.1 Loading, accessing, and freeing a .tpl file

1.5.1.1 Loading true color textures using a GXTexObj

```
void Main ( void )
{
    TEXPalettePtr texPal = 0; // Declare a Texture Palette pointer
    GXTexObj texObj;          // Declare a GXTexObj

    // Load the Texture Palette file “test.tpl” off of disk
    TEXGetPalette(&texPal, "test.tpl");

    // Initialize the GXTexObj to be texture 0 of the texture palette
    TEXGetGXTexObjFromPalette(texPal, &texObj, 0);
    .
    .  // Use the texture
    .
    // Release the Texture Palette
    TEXReleasePalette(&texPal);
}
```

Code 4 Loading true color textures with GXTexObj

Code 4 loads the texture palette file “`test.tpl`” from the disk and into the texture palette pointer *texPal* using `TEXGetPalette`. The function `TEXGetGXTexObjFromPalette` then initializes the `GXTexObj` *texObj* from the texture palette to texture ID 0. At this point, *texObj* can be used to draw geometry. Once the program is done, the

texture palette is released with a call to `TEXReleasePalette`. Please note that the function `TEXGetGXTexObjFromPalette` only works for true-color textures.

1.5.1.2 Loading color-indexed textures using a GXTexObj

```
void Main ( void )
{
    TEXPalettePtr texPal = 0;
    GXTexObj texObj;
    GXTlutObj TlutObj;        // Declare a TLUT object

    TEXGetPalette(&texPal, "test.tpl");

    // Initialize the GXTexObj to be texture 0 of the texture palette
    // Also initialize the GXTlutObj to be the TLUT of texture 0
    TEXGetGXTexObjFromPaletteCI(texPal, texObj, TlutObj, GX_TLUT0, 0);
    .
    .
    .
    TEXReleasePalette (&texPal);
}
```

Code 5 Loading color-indexed textures with GXTexObj

This example performs the same function as Code 4, except that it uses `TEXGetGXTexObjFromPaletteCI` to initialize a color-indexed texture rather than a true-color texture.

1.5.1.3 Loading a texture using the texture descriptor

```
void Main ( void )
{
    TEXPalettePtr texPal = 0;
    GXTexObj texObj;
    GXTlutObj TlutObj;
    TEXDescriptorPtr tdp;
    GXBool mipMap;

    TEXGetPalette(&texPal, "test.tpl");
    tdp = TEXGet(texPal, 0);

    if(tdp->textureHeader->minLOD == tdp->textureHeader->maxLOD)
        mipMap = GX_FALSE;
    else
        mipMap = GX_TRUE;

    if(tdp->CLUTHeader)
    {
        GXInitTlutObj( TlutObj,
                       tdp->CLUTHeader->data,
                       (GXTlutFmt)tdp->CLUTHeader->format,
                       tdp->CLUTHeader->numEntries );

        GXInitTexObjCI( texObj,
                        tdp->textureHeader->data,
                        tdp->textureHeader->width,
                        tdp->textureHeader->height,
                        (GXCITexFmt)tdp->textureHeader->format,
                        tdp->textureHeader->wrapS,
                        tdp->textureHeader->wrapT,
```

```
                        mipMap,
                        GX_TLUT0);
    }
    else
    {
        GXInitTexObj( texObj,
                      tdp->textureHeader->data,
                      tdp->textureHeader->width,
                      tdp->textureHeader->height,
                      (GXTexFmt)tdp->textureHeader->format,
                      tdp->textureHeader->wrapS,
                      tdp->textureHeader->wrapT,
                      mipMap);
    }

    GXInitTexObjLOD( texObj, tdp->textureHeader->minFilter,
                     tdp->textureHeader->magFilter,
                     tdp->textureHeader->minLOD,
                     tdp->textureHeader->maxLOD,
                     tdp->textureHeader->LODBias,
                     GX_DISABLE,
                     tdp->textureHeader->edgeLODEnable,
                     GX_ANISO_1);
    .
    .
    .
    TEXReleasePalette(&texPal);
}
```

Code 6 Loading a texture with a texture descriptor

This example uses a texture descriptor pointer to first determine the type of texture requested (i.e., true-color or color-indexed) and the initializes a `GXTexObj` and `GXTlutObj` accordingly. The texture descriptor is retrieved by `TEXGet`. For detailed information on the `TEXDescriptor` structure, see section 2.2.5.2.

1.6 Hierarchy

The Character Pipeline Actor file format (ACT) enables a few key features. First, the ACT format creates a topology in which display objects from a geometry palette can be organized hierarchically. Second, it allows a single display object to be used as a deformable “skin” for a character. Third, it acts as a binding mechanism to link animation and display objects. The following examples show how to use the basic features of this format.

1.6.1 Loading, displaying, and freeing a .act file

```
void Main ( void )
{
    ACTActorPtr actor = 0;    // Declare an Actor pointer

    // Load the Actor file “test.act” off of disk and into actor
    ACTGet(&actor, “test.act”);

    // Do game loop here
    {
        .
        // Traverse the hierarchy and build all of its transformation matrices
        ACTBuildMatrices(actor);

        // Render the actor using no lights
        ACTRender(actor, cameraMatrix, 0);
```



```
    }

    // Release the actor
    ACTRelease(&actor);
}
```

Code 7 Loading, displaying, and freeing a .act file

Code 7 loads an ACT file from the disk and unpacks it into the actor pointer, *actor*, by calling `ACTGet`. In the game loop, a call to `ACTBuildMatrices` traverses the hierarchy and builds the necessary matrices to properly display the actor. If the hierarchy is not animated, you can save computation cycles by calling `ACTBuildMatrices` once from outside of the game loop. The application then renders the actor by calling the function `ACTRender` and passing it the actor and the camera matrix as arguments. The final parameter to this function is the number of per-vertex lights affecting the actor. Usage of per-vertex lights is discussed in detail is section 1.8.1, but they are not used here. Finally, after the game loop is through, the actor is released with a call to `ACTRelease`.

1.6.2 Controlling an actor using its global control

```
void Main ( void )
{
    ACTActorPtr actor = 0;
    CTRLControlPtr control;
    u32 angle = 0;

    ACTGet(&actor, "test.act");

    // Obtain the pointer to the Actor's global control
    control = ACTGetControl(actor);

    // Set the control's type
    CTRLInit(control);

    // Do game loop here
    {
        // Set the control's parameters to rotate the Actor around the X axis
        CTRLSetRotation(control, (f32)angle, 0.0F, 0.0F);

        ACTBuildMatrices(actor);
        ACTRender(actor, cameraMatrix, 0);

        angle++;
    }

    ACTRelease(&actor);
}
```

Code 8 Controlling an actor

This example starts by obtaining a pointer to the actor's global control with a call to `ACTGetControl` (control structure and usage are discussed more thoroughly in section 1.8.2). In the game loop, the control's *x* rotation parameter is animated, causing the actor to spin around its local *x*-axis.

1.6.3 Instancing an actor

```
void Main ( void )
```

```
{
    ACTActorPtr actor_1 = 0;
    CTRLControlPtr control_1;
    ACTActorPtr actor_2 = 0;
    CTRLControlPtr control_2;
    u32 angle = 0;

    // Create an instance of "test.act"
    ACTGet(&actor_1, "test.act");

    // Create a second instance of "test.act"
    ACTGet(&actor_2, "test.act");

    control_1 = ACTGetControl(actor_1);
    CTRLInit(control_1);

    control_2 = ACTGetControl(actor_2);
    CTRLInit(control_2);

    // Do game loop here
    {
        // Animate the two Actors differently to show they are independent instances
        // of the same Actor.

        CTRLSetRotation(control_1, (f32)angle, 0.0F, 0.0F);
        ACTBuildMatrices(actor_1);
        ACTRender(actor_1, cameraMatrix, 0);

        CTRLSetRotation(control_2, 0.0F, 0.0F, (f32)angle);
        ACTBuildMatrices(actor_2);
        ACTRender(actor_2, cameraMatrix, 0);

        angle++;
    }

    // Release both Actors
    ACTRelease(&actor_1);
    ACTRelease(&actor_2);
}
```

Code 9 Instancing an actor

Code 9 creates two instances of "`test.act`" using the `ACTGet` function. These actors are then animated independently using their global controls, just as in the previous example. As shown in this example, all of the display data is shared when an actor is instanced, but all of the runtime data (such as animation information) is not.

1.7 Animation

The Character Pipeline ANM file format enables the creation and playback of keyframed animation. An animation bank consists of multiple "sequences" which are defined as a complete set of animation data for an actor. An example of a sequence might be a complete run cycle for a character. Sequences are comprised of tracks that contain all the information a single bone within the actor needs to run through the animation sequence. A track is made up of keyframes. A keyframe is defined as a single instance of orientation data for a given bone in time.

1.7.1 Loading, accessing, and freeing a .anm file

```
void Main ( void )
{
    ANIMBankPtr     animBank = 0;
```



```
    ANIMSequencePtr sequence;
    ANIMTrackPtr    track;
    ANIMKeyFramePtr keyFrame1;
    ANIMKeyFramePtr keyFrame2;

    // Load an Animation Bank off of disk and unpack it into animBank
    ANIMGet(&animBank, "test.anm");

    // Retrieve the sequence "testAnim" from the Animation Bank
    sequence = ANIMGetSequence(animBank, "testAnim", 0);

    // Retrieve the Animation Track 0 from the Sequence
    track = ANIMGetTrackFromSeq(sequence, 0);

    // Retrieve the two closest keyframes to time 0.0F from the Animation Track
    ANIMGetKeyFrameFromTrack(track, 0.0F, keyFrame1, keyFrame2);

    // Release the Animation Bank
    ANIMRelease(&animBank);
}
```

Code 10 Loading, accessing, and freeing a .anm file

Code 10 illustrates how to load and unpack an animation bank using the `ANIMGet` function. the `ANIMGetSequence` function then retrieves the animation sequence "`testAnim`" from the animation bank. The function `ANIMGetTrackFromSeq` then extracts animation track 0 from the animation sequence.

Next, the two closest keyframes to time 0.0 are retrieved from the animation track by calling `ANIMGetKeyFrameFromTrack`. (The two closest keyframes are needed in case a keyframe does not occur exactly at the time given. If this happens, the two closest keyframes can be interpolated to yield correct data for the current time.)

Finally, `ANIMReleaseBank` releases the animation bank.

Please note that there is a specific API provided for binding actors to animation data. This data should only be accessed below the animation bank level, if the ANM format is being used independently from the actor.

1.7.2 Animating an actor with a .anm file

```
void Main ( void )
{
    ACTActorPtr actor = 0;
    ANIMBankPtr animBank = 0;

    // Load an Actor file off of disk and unpack it
    ACTGet(&actor, "test.act");

    // Load and unpack an animation bank
    ANIMGet(&animBank, "test.anm");

    // Bind the Actor to the "testAnim" Animation Sequence at time 0.0
    ACTSetAnimation(actor, animBank, "testAnim", 0, 0.0F);

    // Do game loop here
    {
        .
        // move the Actor's animation through one time step
        ACTTick(actor);

        // Build the Actor's matrices
        ACTBuildMatrices(actor);
```

```
        // Render the Actor
        ACTRender(actor, cameraMatrix, 0);
        .
    }

    // Release the data
    ANIMRelease(&animBank);
    ACTRelease(&actor);
}
```

Code 11 Animating an actor with a .anm file

This example binds the "`testAnim`" animation sequence to the actor with a call to `ACTSetAnimation`. Once the actor has been properly bound to a sequence, you can move it through the animation sequence with the `ACTTick` function. This function performs all of the animation for the actor for the current frame, then it updates the actor's current time counter to be used in the next frame.

1.8 Support

The Character Pipeline support libraries are small, simple libraries that add specific functionality to the Character Pipeline. Although all of these libraries can be used independently of the Character Pipeline with no additional changes, we discuss them within the context of the Character Pipeline for the purposes of this document.

1.8.1 Light library

The Character Pipeline light library provides a method of creating, managing, and manipulating per-vertex lights. This library is built directly on top of the GX library and mirrors its functionality in a slightly less cumbersome and more robust manner. This library also adds the ability to orient and animate lights easily.

1.8.1.1 Initializing lights and using them on an actor/display object

```
void Main ( void )
{
    LITLightPtr light = 0;
    GXColor color = {255, 255, 255, 255);

    // Load some Geometry Palette
    // Load and show some Display Object

    // Allocate a light
    LITAlloc(&light);

    // Initialize the light to be a point light at 0, 0, 100 in world space
    LITInitPos(light, 0.0F, 0.0F, 100.0F);

    // Initialize the light's color
    LITInitColor(light, color)

    // Do game loop here
    {
        .
        // Transform the light into camera space for processing
        LITXForm(light, cameraMatrix);

        // Render the Display Object using one light
```



```
        DORender(dispObj, cameraMatrix, 1, light);
        .
    }

    // Release Display Object and Geometry Palette

    // Free light
    LITFree(&light);
}
```

Code 12 Initializing lights and using them on an actor/display object

Code 12 illustrates the basic method in which lights are used. First, a light is allocated with a call to `LITAlloc`. Next, `LITInitPos` initializes the light's position to be (0, 0, 100) in world space. Please note that a light is considered to be a local point light if it only has position (i.e., no direction and angle attenuation). If it has position, direction, and angle attenuation, it is considered to be a local spotlight. Next, `LITInitColor` is called to initialize the light's color to white. The light is transformed into camera space with a call to `LITXForm` (lighting calculations are performed in camera space, so it is necessary to transform the light). The light is then passed as a parameter to the `DORender` function, causing the display object to be affected by the light. The method for using a light on an actor is exactly the same. Up to eight lights can be passed to the `DORender` and `ACTRender` functions. Finally, the light is freed with a call to the `LITFree` function.

1.8.1.2 Manipulating lights

```
void Main ( void )
{
    LITLightPtr light = 0;
    GXColor color = {255, 255, 255, 255);
    CTRLControlPtr control;
    u32 angle = 0;

    // Load some Geometry Palette
    // Load and show some Display Object

    // Allocate and initialize a light
    LITAlloc(&light);
    LITInitPos(light, 0.0F, 0.0F, 100.0F);
    LITInitColor(light, color)

    // retrieve the pointer to the light's control and set the type to SRT
    control = LITGetControl(light);
    CTRLInit(control);

    // Do game loop here
    {
        .
        // set the control to rotate the light around the X axis
        CTRLSetRotation(control, (f32)angle, 0.0F, 0.0F);

        LITXForm(light, cameraMatrix);
        DORender(dispObj, cameraMatrix, 1, light);
        angle++;
        .
    }

    // Release Display Object and Geometry Palette

    // Free light
    LITFree(&light);
```

```
}
```

Code 13 Manipulating lights

The example above shows how to manipulate a light using its control, just as Code 8 did with actors. First, the `LITGetControl` function retrieves a pointer to the light's control. This control is then set to rotate the light around the *x*-axis with a call to `CTRLSetRotation`. The `LITXForm` function transforms the light by the control-specified matrix, and then transforms it into camera space.

1.8.1.3 Animating lights using a .anm file

```
void Main ( void )
{
    LITLightPtr light = 0;
    GXColor color = {255, 255, 255, 255);
    u32 angle = 0;

    // Load some Animation Bank

    // Allocate a Light
    LITAlloc(&light);

    // Initialize the Light to be at 0, 0, 0 (default) and pointing down the -Z axiz
    LITInitDir(light, 0.0F, 0.0F, -1.0F);
    LITInitColor(light, color);

    // Set the light to animate using the "spin" Animation Sequence
    LITSetAnimSequence(light, animBank, "spin", 0.0F);

    // Do game loop here
    {
        .
        // Animate the light for this frame
        LITTick(light);

        LITXForm(light, cameraMatrix);
        DORender(dispObj, cameraMatrix, 1, light);
        .
    }

    // Release Animation Bank

    // Free light
    LITFree(&light);
}
```

Code 14 Animating lights using a .anm file

This example illustrates how to animate a light using the ANM format. First, `LITInitDir` initializes a light to be located at the default (0, 0, 0), and sets the light to point down the negative *z*-axis. The animation sequence "`spin`" is then bound to the light using `LITSetAnimSequence`. The light is animated for a frame with a call to the `LITTick` function, followed by the `LITXForm` function.

1.8.1.4 Attaching lights

```
void Main ( void )
{
```



```
    LITLightPtr light = 0;
    GXColor color = {255, 255, 255, 255};

    // Load some Geometry Palette
    // Load and show some Display Object

    // Allocate and initialize a Light to be a point light a the origin

    LITAlloc(&light);
    LITInitPos(light, 0.0F, 0.0F, 0.0F);
    LITInitColor(light, color);

    // Attach the Light to a Display Object
    LITAttach(light, (Ptr)dispObj, PARENT_DISP_OBJ);

    // Do game loop here
    {
        .
        LITXForm(light, cameraMatrix);
        DORender(dispObj, cameraMatrix, 1, light);
        .
    }
    // Release Display Object and Geometry Palette

    // Free light
    LITFree(&light);
}
```

Code 15 Attaching lights

The example above shows a light's ability to be attached to a display object and thus inherit its motion. Lights can also be attached to actors and arbitrary matrices by using the `LITAttach` function. Lights can be detached from objects by calling `LITDetach`.

1.8.2 Control library

The Character Pipeline control library provides a logical interface for manipulating the orientation, position, and scale of an object. These three parameters make up a control and, ultimately, yield a matrix. The control library is optimized to build matrices intelligently and arithmetically for efficiency, depending on whether position, rotation, or scale is enabled for a given control.

1.8.2.1 Using controls to yield matrices

```
void Main ( void )
{
    CTRLControl control;  // Declare a Control
    Mtx m;

    // Initialize the control
    CTRLInit(&control);

    // Set the control's rotation around the Z axis
    CTRLSetRotation(&control, 0.0F, 0.0F, 90.0F);

    // Set the control's position at (0,0,10) up on the Z axis
    CTRLSetTranslation(&control, 0.0F, 0.0F, 10.0F);
```

```
    // Yield a matrix which reflects the Control's parameters:
    // 1. Do not scale since scale was not set with CTRLSetScale
    // 2. Rotate around the Z axis by 90 degrees
    // 3. Translate the object up on Z axis by 10
    CTRLBuildMatrix(&control, m);
}
```

Code 16 Using controls to yield matrices

Code 16 illustrates the basic way in which to use the Character Pipeline control library. The `CTRLInit` function initializes the control, and the `CTRLSetRotation` function sets the Euler (*x*, *y*, *z*) rotation. Finally, the `CTRLBuildMatrix` function yields a matrix reflecting the current state of the control. Rotations can also be specified with a quaternion using the `CTRLSetQuat` function.

1.8.3 Shader library

The Character Pipeline shader library sits directly on top of the GX library and abstracts the Texture Environment (TEV). The function of the shader library is to allow the user to set up the TEV without worrying about resource restrictions and management.

A "shader" is defined as an expression the results in a color or alpha value. Taken in the context of the GCN system, there are a few defined shader types.

- Constant color.
- Rasterized color.
- Texture.
- Complex.

The process of using a shader is divided into three steps: compilation, data binding, and execution. This split optimizes shader processing at runtime. Most of the work occurs in the compilation stage at loadtime, rather than occurring dynamically at runtime.

In addition to providing TEV functionality, the shader library provides texture coordinate expressions to set up dynamic texture coordinate generation in a generic fashion. For instance, this allows you to write a sphere map shader that needs only a texture and a texture coordinate generation matrix bound to it at runtime.

1.8.3.1 Using texture shaders

```
SHDRCompiled    *CreateTextureShader ( void )
{
    SHDRShader *textureRGB;

    SHDRTexCoord *texcoord0;

    SHDRCompiled *temp;

     //Create texture coordinate expression – use tex coord 0
     texcoord0 = SHDRCreateTCPassThrough(SHADER_TG_TEX0);

    //create RGB side
    textureRGB = SHDRCreateTexture(SHADER_TEX0, texcoord0, SHADER_CHANNEL_RGB);

    //compile
    temp = SHDRCompile(textureRGB, ShaderOne);

    //free shaders
```



```
    SHDRFree(textureRGB);

    return temp;
}

void Main ( void )
{
    //Comile shader
    SHDRCompiled *textureShader = CreateTextureShader();

    //Bind a GXTexObj to the shader
    SHDRBindTexture(textureShader, SHADER_TEX0, MyTexObj);

    //Execute the shader
    SHDRExecute(textureShader);

}
```

Code 17 Creating texture shaders

The preceding example illustrates how to create a shader that returns the RGB component of a texture. The `CreateTextureShader` function declares a texture shader and compiles it using `SHDRCreateTexture` and `SHDRCompile`, respectively. `SHDRCreateTCPassThrough` creates a texture coordinate expression that simply returns the texture coordinate value passed in. This is equivalent to transforming the input coordinate by an identity matrix. Finally, the shade tree is freed with a call to `SHDRFree`.

In the main function, the shader is created and compiled with a call to our `CreateTextureShader` function. A texture is then bound to the shader with `SHDRBindTexture`. Finally, the shader is executed with `SHDRExecute`.

1.8.3.2 Using rasterized color shaders

```
SHDRCompiled    *CreateRasShader ( void )
{
    SHDRShader *rasRGB;

    SHDRCompiled *temp;

    //create RGB side
    rasRGB = SHDRCreateRasterized(SHADER_RAS0, SHADER_CHANNEL_RGB);

    //compile
    temp = SHDRCompile(rasRGB, ShaderOne);

    //free shaders
    SHDRFree(rasRGB);

    return temp;
}

void Main ( void )
{
    //Comile shader
    SHDRCompiled *rasShader = CreateRasShader();

    //Bind a rasterized color channel to the shader
    SHDRBindRasterized(rasShader, SHADER_RAS0, GX_COLOR0A0);

    //Execute the shader
    SHDRExecute(rasShader);
```

```
}
```

Code 18 Creating rasterized color shaders

Code 18 illustrates how to create a shader that returns the RGB component of a rasterized color. The `CreateRasShader` function declares a rasterized color shader and compiles it using `SHDRCreateRasterized` and `SHDRCompile`, respectively. Finally, a call to `SHDRFree` frees the shade tree.

In the main function, the shader is created and compiled with a call to our `CreateRasShader` function. A rasterized color channel is then bound to the shader with a call to `SHDRBindRasterized`. Finally, the shader is executed by `SHDRExecute`.

1.8.3.3 Using constant color shaders

```
SHDRCompiled    *CreateConstantColorShader ( void )
{
    SHDRShader *rgb;

    SHDRCompiled *temp;

    //create RGB side
    rgb = SHDRCreateColor(SHADER_COLOR0, SHADER_CHANNEL_RGB);

    //compile
    temp = SHDRCompile(rgb, ShaderOne);

    //free shaders
    SHDRFree(rgb);

    return temp;
}

void Main ( void )
{
    //Comile shader
    SHDRCompiled *rgbShader = CreateConstantColorShader();
    GXColor red = {255, 0, 0, 0};


    //Bind a constant color to the shader
    SHDRBindColor(rgbShader, SHADER_COLOR0, red);

    //Execute the shader
    SHDRExecute(rasShader);
}

}
```

Code 19 Creating constant color shaders

The example above illustrates how to create a shader that returns the RGB component of a constant color. The `CreateConstantColorShader` function declares a constant color shader and compiles it using the `SHDRCreateColor` and the `SHDRCompile` functions, respectively. Finally, a call to `SHDRFree` frees the shade tree.

In the main function, the shader is created and compiled with a call to our `CreateConstantColorShader` function. A constant color is then bound to the shader with a call to the `SHDRBindColor` function. Finally, the shader is executed with a call to the `SHDRExecute` function.

1.8.3.4 Using complex input shaders

```
SHDRCompiled    *CreateComplexInputShader ( void )
{
    SHDRShader *complexRGB;

    SHDRCompiled *temp;

    //create RGB side
    complexRGB = SHDRCreateComplexInput(SHADER_COMPLEX0, SHADER_CHANNEL_RGB);

    //compile
    temp = SHDRCompile(complexRGB, ShaderOne);

    //free shaders
    SHDRFree(complexRGB);

    return temp;
}

void Main ( void )
{
    //Comile shader
    SHDRCompiled *complexInputShader = CreateComplexInputShader ();

    //Bind a shader to the complex input shader
   SHDRBindComplexInput (complexInputShader, SHADER_COMPLEX0, SomeOtherShader);

    //Execute the shader
    SHDRExecute(complexInputShader);

}
```

Code 20 Creating complex input shaders

The example above creates a complex input shader with a call to the `SHDRCreateComplexInput` function. The shader created simply takes the RGB output from another shader and returns it. This may not seem too useful; however, this mechanism can be used to combine the output of two shaders, which can be very useful. The next example illustrates how to combine shader outputs.

1.8.3.5 Using complex shaders

```
SHDRCompiled    *CreateModulateShader ( void )
{
    SHDRShader *rasRGB;
    SHDRShader *textureRGB;
    SHDRShader *complexRGB;

    SHDRTexCoord *texcoord0;

    SHDRCompiled *temp;

    SHDRCompiled *temp;

    //Create texture coordinate expression – use tex coord 0
    texcoord0 = SHDRCreateTCPassThrough(SHADER_TG_TEX0);

    //Create RGB side
    //Create rasterized shader
    rasRGB = SHDRCreateRasterized(SHADER_RAS0, SHADER_CHANNEL_RGB);
```

```
    //Create texture shader
    textureRGB = SHDRCreateTexture(SHADER_TEX0, texcoord0, SHADER_CHANNEL_RGB);

    //Create multiply shader
    complexRGB = SHDRCreateComplex(ShaderZero, rasRGB, textureRGB, ShaderZero, SHADER_OP_ADD,
                                   SHADER_CLAMP_LINEAR_1023, SHADER_BIAS_ZERO, SHADER_SCALE_1,
                                   SHADER_CHANNEL_RGB);

    //compile
    temp = SHDRCompile(complexRGB, ShaderOne);

    //free shaders
    SHDRFree(complexRGB);

    return temp;
}

void Main ( void )
{
    //Comile shader
    SHDRCompiled *modulateShader = CreateModulateShader();

    //Bind a rasterized color channel to the shader
    SHDRBindRasterized(modulateShader, SHADER_RAS0, GX_COLOR0A0);
    SHDRBindTexture(modulateShader, SHADER_TEX0, MyTexObj);

    //Execute the shader
    SHDRExecute(modulateShader);

}
```

Code 21 Creating complex shaders

The only new function in this example is the `SHDRCreateComplex` function. This function takes four shaders as inputs, then performs some mathematical operation on them (such as add, subtract, multiply, or LERP). This example multiplies the RGB component of a rasterized color with the RGB component of a texture.

1.8.3.6 Using a shader with a display object

```
SHDRCompiled *textureShader;

void SetTextureShader ( SHDRCompiled *shader, DODisplayObjPtr dispObj, u32 combineSetting,
GXTexObj *texture, BOOL colorChanUsed, Ptr data )
{
    #pragma unused (data)
    #pragma unused (combineSetting)
    #pragma unused (texture)
    #pragma unused (colorChanUsed)
    #pragma unused (shader)

    SHDRBindTexture(textureShader, SHADER_TEX0, MyTexObj);
    SHDRExecute(textureShader);
}


void Main ( void )
{

    //Compile shader
    textureShader = CreateTextureShader();
```



```
    //Set callback function
    DOSetEffectsShader(MyDispObj, (Ptr)SetTextureShader, 0);

    //Render the display object
    DORender(MyDispObj, MyCameraMtx, 0);
}
```

Code 22 Creating a shader for use with a display object

This example declares the callback function `SetTextureShader` to bind and execute the global shader `textureShader`. The `CreateTextureShader` function comes from Code 17 above. The callback function is then attached to `MyDispObj` with a call to `DOSetEffectsShader`. As a result, the callback function is called before `MyDispObj` is rendered, therefore executing the desired shader. When `MyDispObj` is rendered, it will be mapped with whatever texture is specified by `MyTexObj`.

2 Runtime libraries in detail

2.1 Geometry

The Character Pipeline geometry palette file format and library form the foundation for the display of geometry. All data drawn by the Character Pipeline is processed by this library. The main function of the format and library is to provide a set of methods to display and manipulate artist-created data. The library provides basic support for loading, manipulating, displaying, and freeing geometry objects. It also supports features such as instancing, lighting, and the specification of programmable texture shaders.

2.1.1 Geometry palette

The geometry palette starts as a file on disk consisting of one or more display objects. A display object is the base unit of displayable geometry in the Character Pipeline. When a geometry palette is loaded into memory, it is unpacked to represent a simple list of one or more of these display objects. The display objects are individually accessible through an API call.

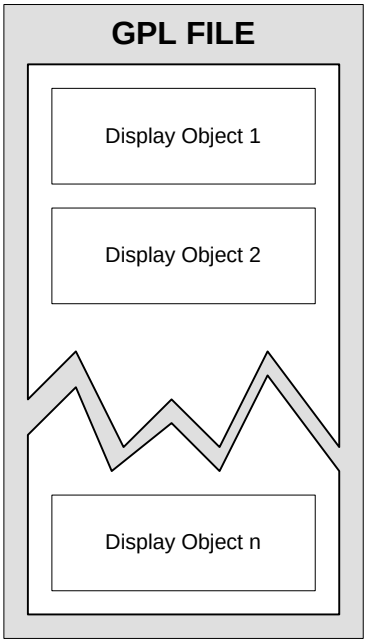


Figure 1 The GPL file

2.1.2 Display object

As mentioned above, the display object is the base unit of displayable geometry available in the Character Pipeline. A display object contains three main types of information:

- **Data Arrays** – Data arrays contain all the necessary data used to make up this display object. A display object can have positions, colors, texture coordinates, and normals as its data arrays. It is not necessary that a display object carry all of this data; a display object could be made up of only position and color, for instance.
- **Connectivity Information** – Connectivity information specifies how the data from the data arrays fits together to form a displayable mesh. This connectivity information comes in the form of hardware display lists that index into the data arrays to form geometric primitives that are then drawn by the hardware.
- **Display State** – A display object also carries with it some hardware state that must be set in order for the geometry to be drawn correctly. This state consists of things like vertex make-up, data quantization information, and texture combine modes.

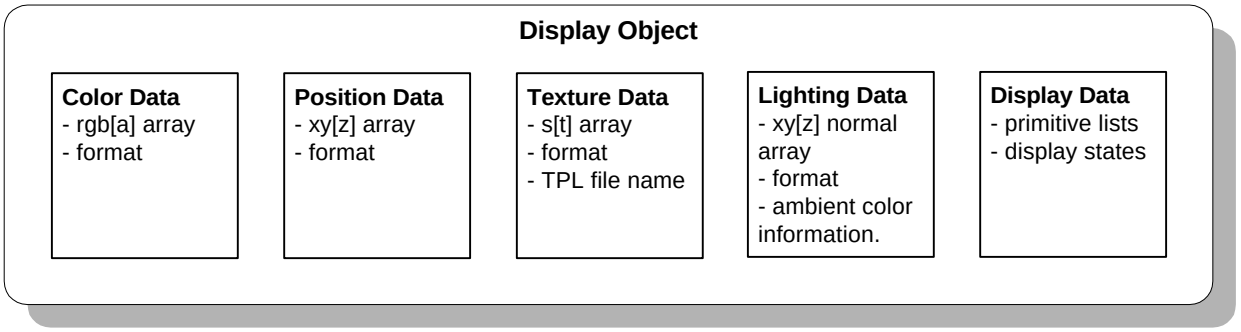


Figure 2 Components of a display object

2.1.2.1 Position data

Position data encompasses all of the state and data related directly to a display object's position array. A display object is required to have position data. The position data contains the actual quantized position array for an object, as well as information on how the data is quantized and what the data looks like (e.g., 2D or 3D).

2.1.2.2 Color data

Color data is comprised of all the data and state necessary to describe the per-vertex colors of an object. It is not necessary for a display object to contain color data; for example, it is possible that an object could have position and texture, but no color. The color data contains the actual color array for an object in addition to information about how the colors are quantized and what the data looks like (e.g., RGB or RGBA). This data also acts as the vertex material information if the object is lit by a per-vertex light.

2.1.2.3 Texture data

Texture data consists of all the data and state necessary to apply texture to a display object properly. Texture data is not required in a valid display object. It is possible, for instance, that an object could have position and color, but no texture data. Texture data carries the actual quantized texture coordinate array along with information about how the data is quantized and what the data looks like (e.g., 1D or 2D). Also, the texture data contains a pointer to a texture palette from which textures are loaded to draw the object. For more information on texture palettes, see section 2.2.

2.1.2.4 Lighting data

Lighting data contains information for performing per-vertex lighting on a display object. As a result, objects that will never be affected by a per-vertex light do not need to include lighting data. Lighting data includes a quantized normal array, information on how the normals are quantized, and the type of normal (e.g., 2D or 3D). Also, lighting data contains a percentage of the vertex color that will be considered ambient, which allows an ambient light effect without having to actually burn a light in hardware.

2.1.2.5 Display data

Display data is a bit more complex than the other data sets described above. Display data is comprised of two major components:

- **Display State List** – The display state list is a list of individual hardware states that must be set at a given time during the drawing of the object.
- **Primitive Lists** – Primitive lists are hardware display lists which define how the display object data (positions, normals, colors, and texture coordinates) fit together to form the object.

At runtime, the display state list is traversed in order. Each specified state is set in the hardware until a display state entry is encountered that points to a primitive list. When this happens, the primitive list is drawn, and the display routine resumes setting hardware state until the end of the state list is reached. At this time, the entire object should be drawn.

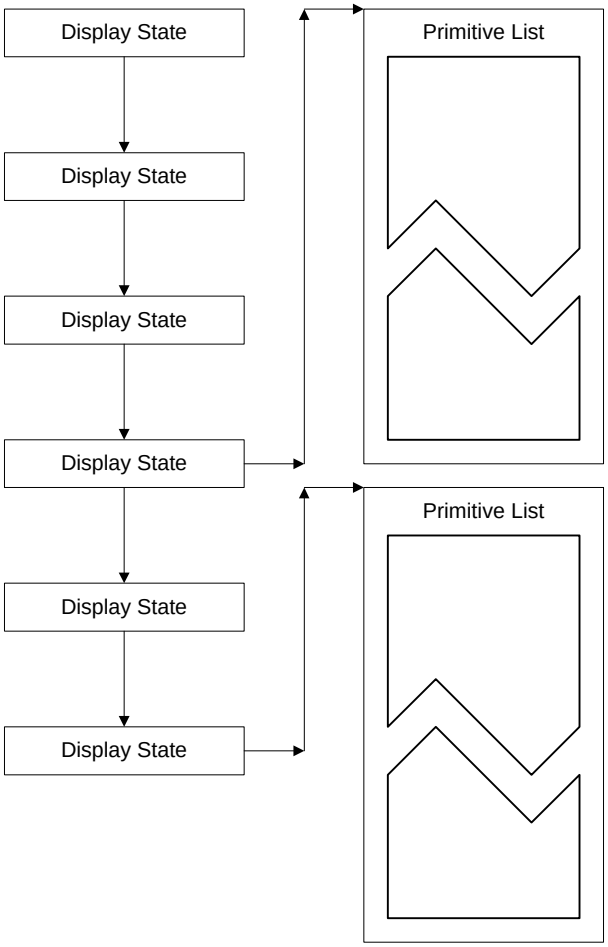


Figure 3 Display state and primitive lists

2.1.3 Display object layout

After a geometry palette is loaded and unpacked into memory, it is stored as a series of display object layouts. A display object layout acts as a sort of recipe for building a display object. The layout points to all of the necessary data to display a display object, but it does not contain any of the information (such as orientation information) that a useable runtime instantiation should have.

2.1.4 Creating an instance of a display object

When a display object is created from a display object layout, all of the data is instanced from the layout. However, additional runtime information is added to the display object that does not behave in an instanced manner. For example, if there are two instances of the same display object in a game, each instance can be positioned, oriented, and scaled differently, but they share the same data arrays. This means that if you were to change a data element in a data array of the first object, the second object would reflect this change as well.

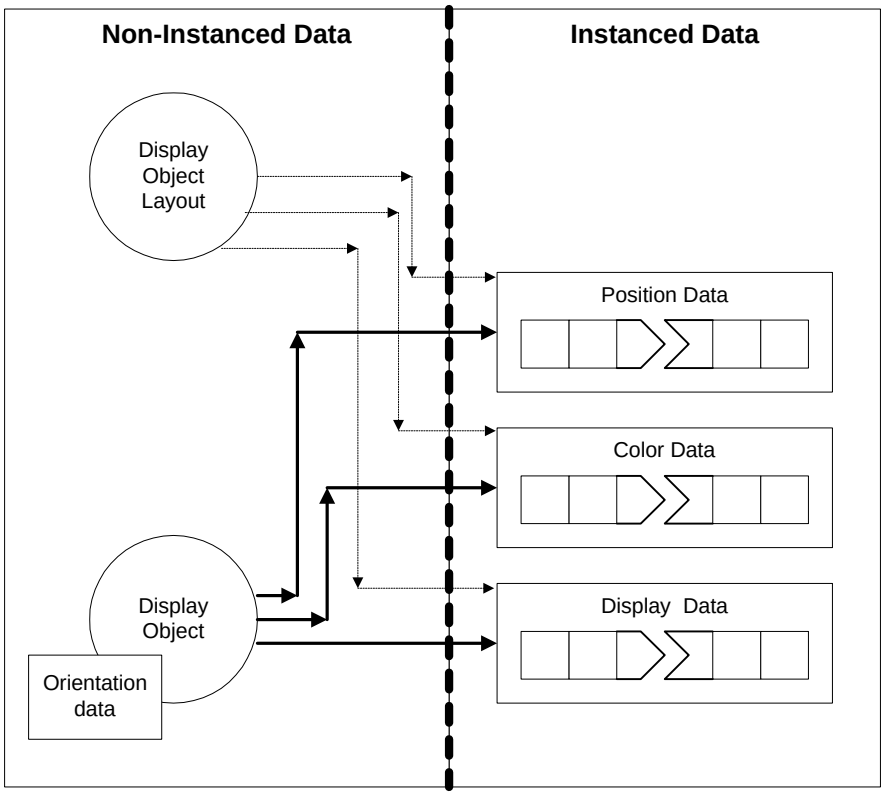


Figure 4 Display object instancing mechanism

2.1.5 Display object matrices

At runtime, a display object carries a world matrix which transforms the object from local space to world space. This world matrix is not instanced like data arrays; rather, it belongs to one—and only one—display object.

2.1.6 Display object visibility flag

Each display object has a unique visibility flag that determines whether or not the object should be rendered. That is, sending a display object that is tagged as "invisible" to the render routine will result in the display object not being rendered. This flag is provided as an easy way to "turn off" a display object in a scene without removing it from the database or freeing it from memory.

2.1.7 Processing the display object for display

When a display object is first sent to the rendering routine, a check is performed on the state of the display object's visibility flag. If this flag indicates that the display object is invisible, the routine simply returns. At this point, the display object's world matrix is concatenated with the given camera matrix to create the modelview matrix. The routine then begins to set up the hardware with the display object's global state.

2.1.7.1 Setting display object global state

Global state for a display object is defined as any hardware state that does not need to be changed over the course of drawing the object. The global state of a display object includes:

- Modelview matrix.
- Data array locations.
- Data quantization methods.
- Lighting state.

2.1.7.2 Processing the display state list

After the global state is set for a display object, the rendering routine processes the display state list contained in the display data section of the display object. Starting from the first entry in the list, the code parses the state entries and sets the proper hardware state to the specified values. The hardware state set by display state entries includes:

- Texture state.
- Color Combine state.
- Vertex Descriptor state.
- Matrix state.

Each state entry contains a pointer to a primitive list. If this pointer is not `NULL`, the rendering routine draws all the geometry specified in the primitive list and then processes the next state entry.

2.1.7.3 Processing primitive lists

A primitive list is simply a hardware display list that contains no state. All hardware state is specified from the display object rendering code via the immediate mode GX API, and therefore does not belong in the primitive list. It is important to note that these primitive lists only refer to data in an indexed manner. No direct data is allowed in a display object's primitive lists. Indexes can, however, be either 8-bit or 16-bit to aid compression. With the data organized in this manner, it is a simple matter to send this primitive list directly to the GCN hardware.

2.1.8 Display object lighting

The display object rendering routine can accept eight per-vertex lights by which the specified display object will be affected. These lights are specified as defined by the Character Pipeline light library (see section 2.5.3). If the given display object contains lighting data, these lights are loaded into hardware, and the display object's color array is used to provide per-vertex material information.

In addition to the ability to have an object affected by eight lights, it is also possible to simulate ambient lighting for free. Each display object carries an ambient percentage field in its lighting data that specifies the darkest level the object can ever appear. This field is specified as a floating point number in the range from 0.0 to 100.00, inclusive, and it can be modified at runtime. The ambient value for a display object is not instanced and can therefore be changed on a per-object basis.

2.1.9 Stitching a display object

Up until now, we've discussed a display object only as a series of primitives that are transformed by a single matrix. In actuality, a display object also has the ability to transform each of its vertices by a unique matrix. This technique, in which you can warp a mesh by transforming different vertices within it by different matrices, is called stitching.

A display object needs three sets of information to make stitching possible:

- At runtime, the display object must be passed two arrays of matrices that will transform its positions and normals into world space.
- The display object must have state entries in its display state list that instruct the hardware to load certain matrices from the supplied matrix arrays into the hardware matrix cache.
- The display object must have matrix indices in its primitive list that describe by which matrix in the hardware matrix cache each position is to be transformed.

2.1.9.1 Dealing with the matrix cache

A potentially confusing level of indirection is necessary when dealing with multiple matrices per display object. The GCN hardware contains a matrix cache in which all matrices used to transform geometry must reside. The problem occurs when the number of matrices needed to properly deform the given display object exceeds the hardware limit of ten.

The solution is simply to break up the display object into several primitive lists that each require ten or less matrices to draw. Then we simply load the matrices needed by any given primitive list into the matrix cache, draw the primitive list, then move on to the next primitive list.

To accomplish this, we add entries in the display state list instructing the hardware to load a matrix from the given matrix arrays into hardware. Once all the necessary matrices have been loaded into the matrix cache, the corresponding primitive list can be drawn. It is important to note that the primitive list indexes the matrices in the hardware cache, not the two arrays from which these matrices have been loaded.

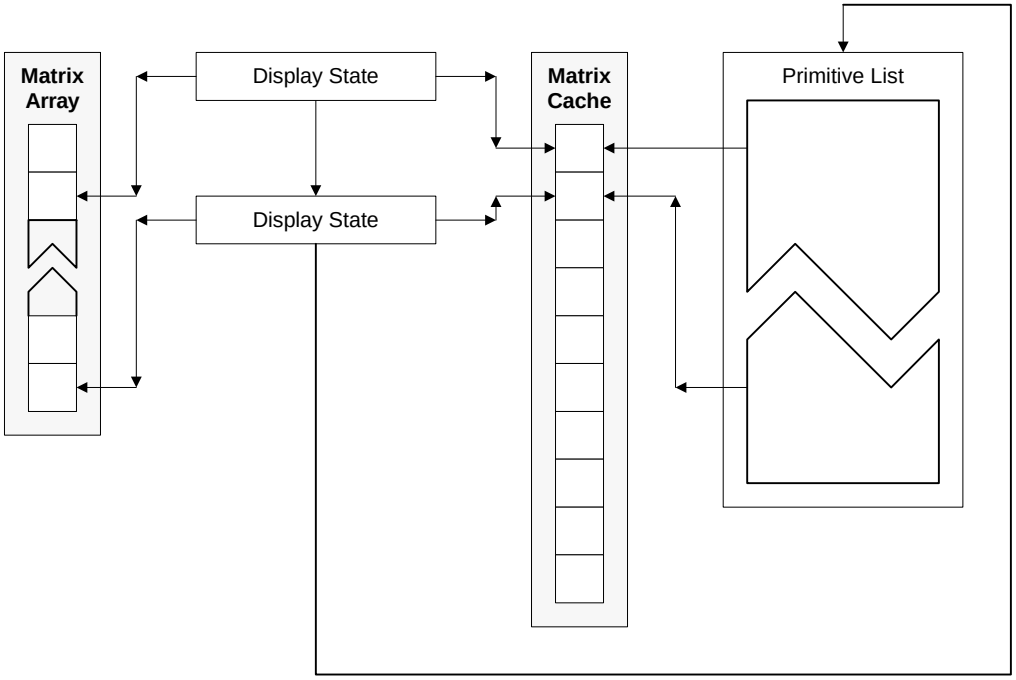


Figure 5 The hardware matrix cache

2.1.10 Geometry palette structures

2.1.10.1 GEOPalette

```
typedef struct
{
    u32               versionNumber;

    u32               userDataSize;
    void*             userData;

    u32               numDescriptors;
    GEODescriptorPtr  descriptorArray;

} GEOPalette, *GEOPalettePtr;
```

Code 23 GEOPalette

- *versionNumber* – Field to describe the version number of the file.
- *userDataSize* – Size of the GPL user-defined data.
- *userData* – Pointer to the GPL user-defined data.
- *numDescriptors* – Number of geometry descriptors in the file.
- *descriptorArray* – Array of geometry descriptors.

2.1.10.2 GEODescriptor

```
typedef struct
{
    DOLayoutPtr  layout;
    char         *name;

} GEODescriptor, *GEODescriptorPtr;
```

Code 24 GEODescriptor

- *layout* – Pointer to a display object layout.
- *name* – String name for this display object.

2.1.10.3 DOLayout

```
typedef struct
{
    DOPositionHeaderPtr     positionData;
    DOColorHeaderPtr        colorData;
    DOTextureDataHeaderPtr  textureData;
    DOLightingHeaderPtr     lightingData;
    DODisplayHeaderPtr      displayData;
    u8                      numTextureChannels;
    u8                      pad8;
    u16                     pad16;

} DOLayout, *DOLayoutPtr;
```

Code 25 DOLayout

- *positionData* – Pointer to the display object layout's position header.
- *colorData* – Pointer to the display object layout's color header.
- *textureData* – Pointer to the display object layout's texture header.
- *lightingData* – Pointer to the display object layout's lighting header.
- *displayData* – Pointer to the display object layout's display header.
- *numTextureChannels* – Number of texture headers in *textureData*.

2.1.10.4 DOPositionHeader

```
typedef struct
{
    Ptr  positionArray;
    u16  numPositions;
    u8   quantizeInfo;
    u8   compCount;

} DOPositionHeader, *DOPositionHeaderPtr;
```

Code 26 DOPositionHeader

- *positionArray* – Pointer to a quantized array of position data.
- *numPositions* – Number of positions in the position array. Note: a triplet of (*x*, *y*, *z*) values is considered *one* entry, not three.

- *quantizeInfo* – Bit field describing the quantization of the position data (see Figure 19).
- *compCount* – Number of components that make up an individual position; e.g., 2 for (*x*, *y*) pairs, 3 for (*x*, *y*, *z*) triples.

2.1.10.5 DOColorHeader

```
typedef struct
{
    Ptr  colorArray;
    u16  numColors;
    u8   quantizeInfo;
    u8   compCount;

} DOColorHeader, *DOColorHeaderPtr;
```

Code 27 DOColorHeader

- *colorArray* – Pointer to a quantized array of color data.
- *numColors* – Number of colors in the color array.
- *quantizeInfo* – Bit field describing the quantization of the color data (see Figure 19).
- *compCount* – Number of components that make up an individual color; e.g., 3 for RGB, 4 for RGBA.

2.1.10.6 DOTextureDataHeader

```
typedef struct
{
    Ptr  textureCoordArray;
    u16  numTextureCoords;
    u8   quantizeInfo;
    u8   compCount;

    char          *texturePaletteName;
    TEXPalettePtr  texturePalette;

} DOTextureDataHeader, *DOTextureDataHeaderPtr;
```

Code 28 DOTextureDataHeader

- *textureCoordArray* – Pointer to a quantized array of texture coordinate data.
- *numTextureCoords* – Number of texture coordinates in the texture coordinate array.
- *quantizeInfo* – Bit field describing the quantization of the texture coordinate data (see Figure 19).
- *compCount* – Number of components that make up an individual texture coordinate; e.g., 1 for 1D texture coordinates, 2 for 2D texture coordinates.
- *texturePaletteName* – String name of the texture palette file that is associated with this display object. **NOTE:** there is no path included with this name—the file is assumed to exist in the same directory as the geometry palette file.
- *texturePalette* – Pointer to the unpacked texture palette in memory.

2.1.10.7 DOLightingHeader

```
typedef struct
{
    Ptr  normalArray;
```

```
    u16  numNormals;
    u8   quantizeInfo;
    u8   compCount;
    f32  ambientPercentage;

} DOLightingHeader, *DOLightingHeaderPtr;
```

Code 29 DOLightingHeader

- *normalArray* – Pointer to a quantized array of normal data.
- *numNormals* – Number of normals in the normal array.
- *quantizeInfo* – Bit field describing the quantization of normal data (see Figure 19).
- *compCount* – Number of components that make up an individual texture coordinate; e.g., 1 for 1D texture coordinates, 2 for 2D texture coordinates.
- *ambientPercentage* – Floating point value that specifies the percentage of color that is assumed to be ambient.

2.1.10.8 DODisplayHeader

```
typedef struct
{
    Ptr                primitiveBank;
    DODisplayStatePtr  displayStateList;
    u16                numStateEntries;
    u16                pad16;

} DODisplayHeader, *DODisplayHeaderPtr;
```

Code 30 DODisplayHeader

- *primitiveBank* – Pointer to the first primitive list for the display object. (Primitive lists are described in detail in section 2.1.7.3.)
- *displayStateList* – Pointer to an array of display state structures. (Display state is described in detail in section 2.1.7.2.)
- *numStateEntries* – Number of state entries in the display state list.

2.1.10.9 DODisplayState

```
typedef struct
{
    u8   id;

    u8   pad8;
    u16  pad16;

    u32  setting;

    Ptr  primitiveList;
    u32  listSize;

} DODisplayState, *DODisplayStatePtr;
```

Code 31 DODisplayState

- *id* – Specifies which state this state entry sets. (For a complete list of possible IDs, see section 2.1.7.2.)
- *setting* – Specifies the setting for the state specified by *id*.

- *primitiveList* – Pointer to a primitive list which is to be drawn after the specified state is set. NULL is a valid value for this member.
- *listSize* – Size (in bytes) of the primitive list specified by *primitiveList*.

2.1.11 Geometry palette API

```
void GEOGetPalette      ( GEOPalettePtr *pal, char *name );
void GEOReleasePalette  ( GEOPalettePtr *pal );
u32  GEOGetUserDataSize ( GEOPalettePtr pal );
Ptr  GEOGetUserData     ( GEOPalettePtr pal );
```

Code 32 Geometry palette API

2.1.12 Display object structures

2.1.12.1 DODisplayObj

```
typedef struct
{
    DOPositionHeaderPtr     positionData;
    DOColorHeaderPtr        colorData;
    DOTextureDataHeaderPtr  textureData;
    DOLightingHeaderPtr     lightingData;
    DODisplayHeaderPtr      displayData;
    u8                      numTextureChannels;
    u8                      pad8;
    u16                     pad16;

    // Up to here, same as DOLayout
    u8     visibility;
    Mtx    worldMatrix;

    void*  (*shaderFunc)(SHDRCompiled *shader, struct DODisplayObj *dispObj, Ptr data);
    void*  shaderData;

}DODisplayObject, *DODisplayObjectPtr, DODisplayObj, *DODisplayObjPtr;
```

Code 33 DODisplayObj

- *positionData* – Pointer to the display object's position header.
- *colorData* – Pointer to the display object's color header.
- *textureData* – Pointer to the display object's texture header.
- *lightingData* – Pointer to the display object's lighting header.
- *displayData* – Pointer to the display object's display header.
- *numTextureChannels* – Number of texture headers in *textureData*.
- *visibility* – Flag to denote if the display object should be rendered.
- *worldMatrix* – Matrix used to transform the display object's positions into world space.
- *shaderFunc* – Pointer to a function which takes the default shader for this object, performs some shader operations on it, and returns a new shader.

2.1.13 Display object API

```
void   DOGet                 ( DODisplayObjPtr *dispObj, GEOPalettePtr pal, u16 id, char *name );
void   DORelease             ( DODisplayObjPtr *dispObj );
MtxPtr DOGetWorldMatrix      ( DODisplayObjPtr dispObj );
void   DOSetWorldMatrix      ( DODisplayObjPtr dispObj, Mtx m );
void   DOHide                ( DODisplayObjPtr dispObj );
void   DOShow                ( DODisplayObjPtr dispObj );
void   DORender              ( DODisplayObjPtr dispObj, Mtx camera, u8 numLights, ... );
void   DORenderSkin          ( DODisplayObjPtr dispObj, Mtx camera, MtxPtr mtxArray,
                               MtxPtr inverseTransposeMtxArray, u8 numLights, ... );
f32    DOSetAmbientPercentage ( DODisplayObjPtr dispObj, f32 percent );
```

Code 34 Display object API

2.2 Texture

The TPL format and library are designed to provide an easy method of texture storage and access. The library attempts to abstract away many of the format issues, allowing any of The GCN's myriad texture formats to be used in a consistent manner. The library provides the basic functionality of loading a file from disk and unpacking it into memory, retrieving information about an individual texture from the palette, and then loading that texture into hardware so that it can be used to draw geometry. The library provides support for true-color textures as well as color-indexed textures.

2.2.1 Texture palette in memory

When a texture palette file is loaded and unpacked into memory, it simply represents a list of texture descriptors. A texture descriptor maintains a pointer to a texture image and, if needed, a Color Lookup Table (CLUT).

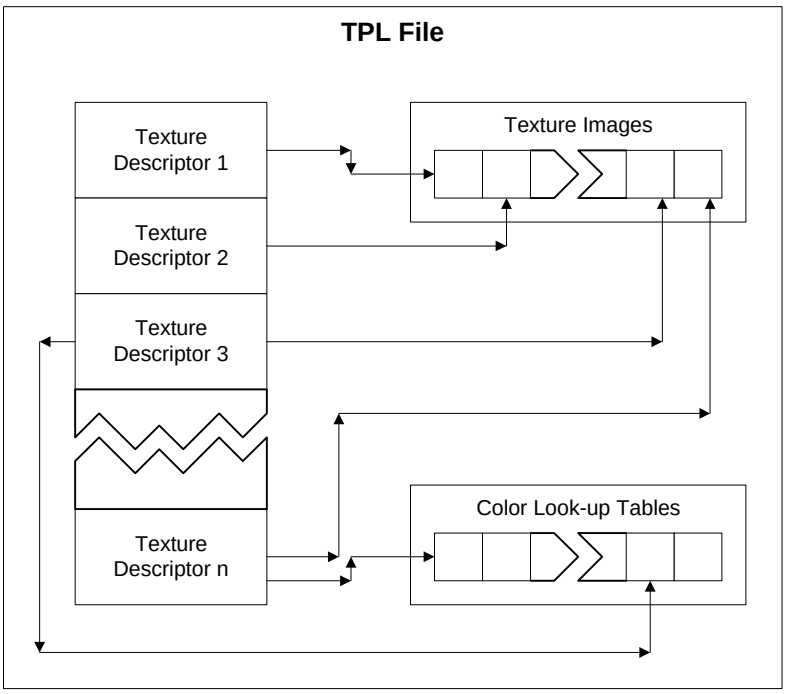


Figure 6 The texture palette in memory

2.2.2 Instancing CLUTs within a texture palette

Because texture images and CLUTs are referenced via pointer, you can instance them within a texture palette. Textures might share a CLUT, for example, or there might be one instance of a texture image using multiple CLUTs.

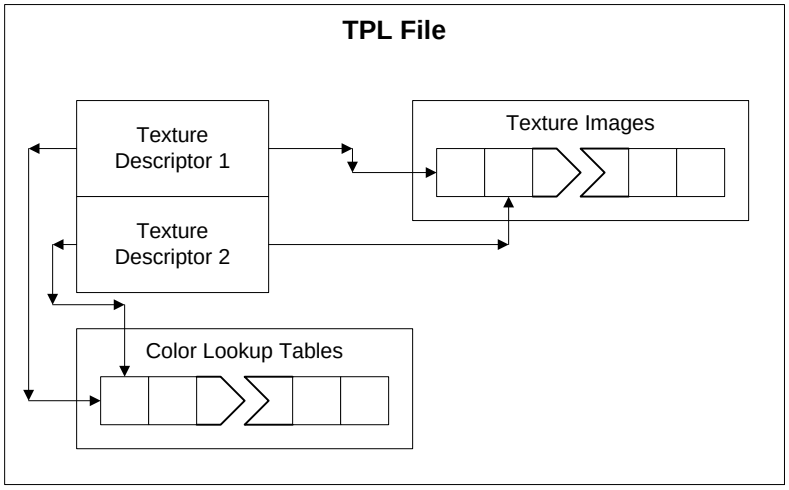


Figure 7 CLUT instancing within a texture palette

2.2.3 Retrieving texture information via a texture descriptor

Once a texture palette has been loaded and unpacked, individual textures within the palette can be accessed via ID. One method to retrieve texture information is via a texture descriptor pointer. The texture descriptor points to a texture image header and, if the texture is color-indexed, a CLUT header. The texture image header contains the actual bitmap for the image as well as format information, wrap and clamp information, size information, and mipmapping information. A CLUT header contains the actual CLUT data as well as format information and size information. The information from these structures can be used to initialize a `GXTexObj` and a `GXTlutObj`, respectively. These objects can then be sent to the hardware to perform necessary graphics operations.

2.2.4 Retrieving texture information via a GXTexObj and a GXTlutObj

In addition to providing texture information via a texture descriptor, the texture palette library also provides two functions to initialize a `GXTexObj` and a `GXTlutObj` directly. These functions simply look at the texture descriptor information and properly initialize the GX objects specified.

2.2.5 Texture palette structures

2.2.5.1 TEXPalette

```
typedef struct
{
    u32               versionNumber;
    u32               numDescriptors;
    TEXDescriptorPtr  descriptorArray;

} TEXPalette, *TEXPalettePtr;
```

Code 35 TEXPalette

- *versionNumber* – Field to describe the version number of the file.
- *numDescriptors* – Number of texture descriptors in the file.
- *descriptorArray* – Array of texture descriptors.

2.2.5.2 TEXDescriptor

```
typedef struct
{
    TEXHeaderPtr   textureHeader;
    CLUTHeaderPtr  CLUTHeader;

} TEXDescriptor, *TEXDescriptorPtr;
```

Code 36 TEXDescriptor

- *textureHeader* – Pointer to a texture header.
- *CLUTHeader* – Pointer to a Color Lookup Table header.

2.2.5.3 TEXHeader

```
typedef struct
{
    u16             height;
    u16             width;
    u32             format;
    Ptr             data;

    GXTexWrapMode   wrapS;
    GXTexWrapMode   wrapT;
    GXTexFilter     minFilter;
    GXTexFilter     magFilter;

    float           LODBias;
    u8              edgeLODEnable;
    u8              minLOD;
    u8              maxLOD;
    u8              unpacked;

} TEXHeader, *TEXHeaderPtr;
```

Code 37 TEXHeader

- *height* – Height (in pixels) of texture image.
- *width* – Width (in pixels) of texture image.
- *format* – Format of the texture image.
- *Data* – Pointer to the actual texture image data.
- *wrapS* – Describes how texture coordinates will be wrapped in the *s* direction. Accepted values: `GX_CLAMP, GX_REPEAT, GX_MIRROR.`
- *wrapT* – Describes how texture coordinates will be wrapped in the *t* direction. Accepted values: `GX_CLAMP, GX_REPEAT, GX_MIRROR.`
- *minFilter* – Filter mode to use when the texel/pixel ratio is <= 1.0. Accepted values: `GX_NEAR`, `GX_LINEAR`, `GX_NEAR_MIP_NEAR`, `GX_LIN_MIP_NEAR`, `GX_NEAR_MIP_LIN`, `GX_LIN_MIP_LIN`.
- *magFilter* – Filter mode to use when the texel/pixel ratio is > 1.0. Accepted values: `GX_NEAR`, `GX_LINEAR`.
- *LODBias* – Bias to add to computed LOD value.
- *edgeLODEnable* – Compute LOD using adjacent texels when `GX_TRUE`, else use diagonal texels.
- *minLOD* – Minimum LOD value. The hardware will use `MAX(min_lod, lod)`. Range is 0.0 to 10.0.
- *maxLOD* – Maximum LOD value. The hardware will use `MIN(max_lod, lod)`. Range is 0.0 to 10.0.
- *unpacked* – Internal flag used for file unpacking.

2.2.5.4 CLUTHeader

```
typedef struct
{
    u16         numEntries;
    u8          unpacked;
    u8          pad8;
    GXTlutFmt   format;
    Ptr         data;

} CLUTHeader, *CLUTHeaderPtr;
```

Code 38 CLUTHeader

- *numEntries* – Number of palette entries.
- *unpacked* – Internal flag used for file unpacking.
- *format* – Format of the CLUT data.
- *data* – Pointer to CLUT data.

```
void             TEXGetPalette               ( TEXPalettePtr *pal, char *name );
TEXDescriptorPtr TEXGet                      ( TEXPalettePtr pal, u32 id );
void             TEXReleasePalette           ( TEXPalettePtr *pal );
void             TEXGetGXTexObjFromPalette   ( TEXPalettePtr pal, GXTexObj *to, u32 id );
void             TEXGetGXTexObjFromPaletteCI ( TEXPalettePtr pal, GXTexObj *to,
                                               GXTlutObj *tlo, GXTlut tluts, u32 id );
```

Code 39 Texture palette API

2.3 Hierarchy

The Character Pipeline Actor (ACT) format and library provide a topology for grouping display objects together hierarchically. This topology also provides a logical binding to animation. Using the ACT format and library, you can create and manipulate hierarchical characters and databases as a whole, rather than having to access each display object individually and manipulate it independently of another.

We refer to the global database as an *actor*, while each node in the hierarchy is called a *bone*. A display object can be attached to an individual bone, inheriting its animation parameters completely, or, in the case of stitching, it can be attached to multiple bones, having individual vertices affected by different bones.

2.3.1 Actor layout in memory

The actor database behaves in much the same manner as the display objects we have seen in previous sections. When an actor file is loaded from the disk and into memory, it is unpacked into a *layout* structure that contains some, but not all, of the information necessary to use an actor at runtime.

Layout structures contain information about how the database organizes hierarchically, and about which geometry palette/display objects the actor affects. While the actor is being unpacked, the corresponding geometry palette is loaded or instanced based on the state of the Character Pipeline cache. Display objects that are used in this actor are instanced and bound to the actor layout structures.

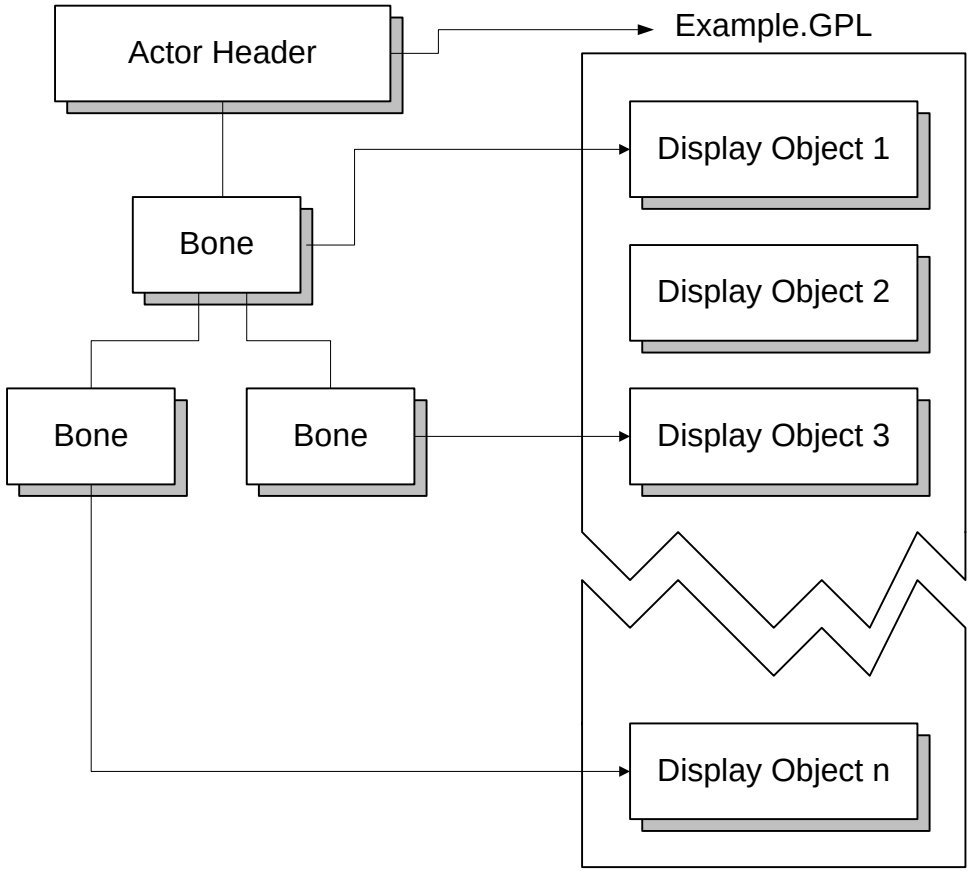


Figure 8 Actor layout

2.3.2 Bone layout in memory

Within the actor layout, each bone has a packet of information called the bone layout. This layout contains information about the bone, such as what display object is attached, the bone's display priority, and the bone's default orientation in relation to the root of the hierarchy.

2.3.3 Actor instancing mechanism

When an instance of an actor is created, each display object to which the actor refers behaves in the exact same instanced manner as described in section 2.1.4. The actor structure is so small that the overhead of instancing actually becomes more costly than the overhead of copying; therefore, all of the static data is copied—not instanced—from the actor layout structures. Additional runtime information, such as animation control data, is added to the actor and bone structures.

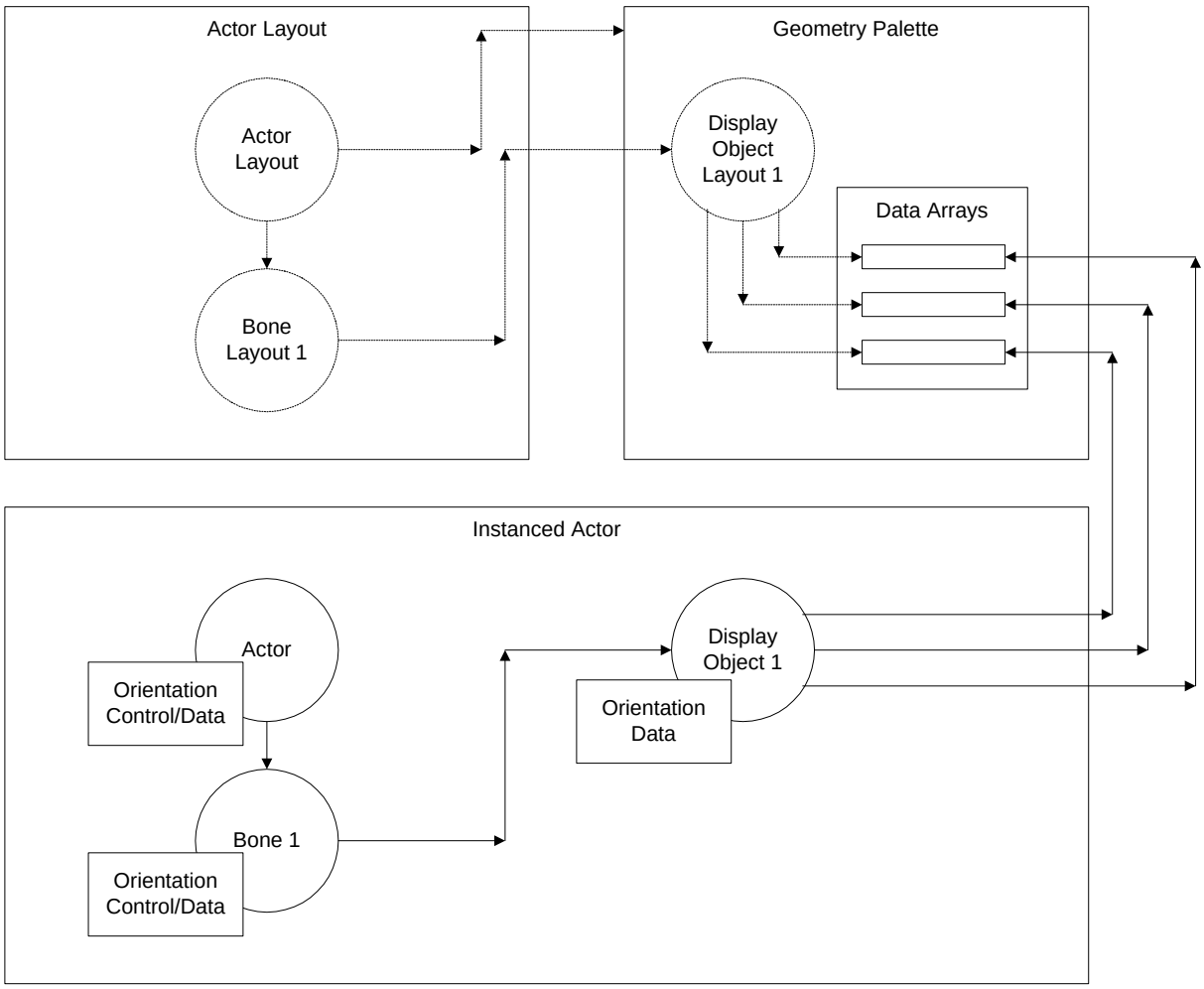


Figure 9 Instancing an actor

2.3.4 Actor at runtime

The actor at runtime maintains a relatively complex topology. At the top level, the actor header contains all of the global state for the entire actor, as well as a tree of bone structures that make up the hierarchy. For each bone in the hierarchy, there is one bone header that contains local information for that bone.

2.3.4.1 The actor header

The actor header serves as a depository for all of the global state for an actor. Key members of the actor header data structure include:

- **Actor Global Control** – The actor's global control yields a matrix used to manipulate the actor as a whole. This matrix will be inherited by all of the bones in the hierarchy regardless of their inheritance state. This allows the user to position, orient, and scale an actor relative to the environment in which it exists.
- **Bone Tree** – The actor's bone tree organizes all of the actor's bones into a hierarchical database. This representation is used to perform matrix concatenations at runtime because bones often inherit matrix information from their parents.

- **Bone List** – The actor's bone list is a linked list containing all of the bones in the hierarchy sorted in order of display priority. (Display priority is a property of each bone and is discussed below.) This representation is used to process each bone's display object (if there is one present) for display.
- **Skin Object** – The actor's skin object points to a display object that represents a deformable mesh that has been attached to the actor. Currently, there can be only one skin object attached to an actor.
- **Geometry Palette** – The actor's geometry palette is the geometry palette from which all display objects attached to the actor are instanced. This geometry palette is loaded when the actor is unpacked.
- **Forward Matrix Array** – The forward matrix array represents an array of matrices into which each bone places its final matrix. In other words, there is a one-to-one correspondence between the number of matrices in this array and the number of bones in the hierarchy. When a bone builds its final matrix from its orientation, animation, and inheritance, that matrix is stored in the bone's matrix in this array.
- **Skinning Matrix Array** – The skinning matrix array is present only in actors that have a skin object attached to them. This array performs the same function as the forward matrix array, except that the matrices contained within this array are used only on skin objects.
- **Skinning Inverse Transpose Matrix Array** – This array performs the same function as the skinning matrix array, except that the matrices stored within this array are used to transform a skin object's normals rather than its vertices. Also, this array is not present in actors that do not have a skin object attached.

2.3.4.2 The bone header

Each bone within an actor is represented by a bone header structure. The key elements in this structure are as follows:

- **Orientation Control** – A bone's orientation control defines a default orientation, position, and scale for a bone within an actor. All animation performed on this bone should be relative to these default parameters. The bone is always transformed by the matrix yielded by these default parameters.
- **Animation Control** – The animation control for a bone defines some animation relative to the bone's default orientation, position, and scale that should be factored into the final matrix for the bone. The animation control is manipulated by the animation pipe structure (described next).
- **Animation Pipe** – The animation pipe is a structure that binds a bone to animation. A bone that has never been animated will not contain an animation pipe structure—it is allocated when the bone is first bound to animation.
- **Bone Priority** – Each bone has a display priority that indicates when its corresponding display object should be drawn in relation to the other bones in the hierarchy.
- **Display Object** – A bone can have only one display object attached to it at any given time. A display object attached to a bone will completely inherit the bone's motion. This display object is instanced from the geometry palette attached to the actor in the actor header structure.

2.3.5 Processing an actor for display

When an actor is rendered, the code first draws the skin object attached to the actor (if one is present), then it simply traverses the bone list and draws the display object attached to each bone in the order determined by the list. The actor can also be lit by up to eight per-vertex lights. These lights get passed to each call to the display object rendering routine.

2.3.6 Actor structures

2.3.6.1 ACTLayout

```
typedef struct
{
```

```
    u32     versionNumber;

    u16     actorID;
    u16     totalBones;

    DSTree  hierarchy;

    char   *geoPaletteName;

    u16     skinFileID;
    u16     pad16;

    u32     userDataSize;
    Ptr     userData;

} ACTLayout, *ACTLayoutPtr;
```

Code 40 ACTLayout

- *versionNumber* – Field to describe the version number of the file.
- *actorID* – ID number for this actor.
- *totalBones* – Total number of bones in the actor.
- *hierarchy* – `DSTree` structure which describes the hierarchical relationship of the bones in the actor.
- *geoPaletteName* – String name of a geometry palette file that contains the geometry to be attached to the actor's bones.
- *skinFileID* – ID number of the actor's skin (deformable mesh) geometry within the geometry palette specified by *geoFileName*. A value of `ACT_NO_DISPLAY_OBJECT` indicates that the actor has no skin geometry.
- *userDataSize* – Size of the ACT user-defined data.
- *userData* – Pointer to the ACT user-defined data.

2.3.6.2 ACTBoneLayout

```
typedef struct
{
    CTRLControlPtr  orientationControl;
    DSBranch        branch;
    u16             geoFileID;
    u16             boneID;
    u8              inheritanceFlag;
    u8              drawingPriority;
    u16             pad16;

} ACTBoneLayout, *ACTBoneLayoutPtr;
```

Code 41 ACTBoneLayout

- *orientationControl* – Pointer to a control which specifies the bone's default orientation. (See section 2.5.2 for more information on controls.)
- *branch* – `DSBranch` structure which specifies the bone's position within the hierarchy.
- *geoFileID* – ID number of the display object associated with this bone within the geometry palette specified in the actor layout. A value of `ACT_NO_DISPLAY_OBJECT` indicates that the bone has no geometry attached to it.
- *boneID* – Unique ID number for the bone within the actor.
- *inheritanceFlag* – Flag which specifies whether the matrix from the bone's parent should be concatenated with the bone's matrix before rendering it.
- *DrawingPriority* – Number stating when this bone's geometry should be drawn in relationship to the other bones in the actor. A lower value for this field indicates a higher drawing priority.

2.3.6.3 ACTActor

```
typedef struct
{
    ACTLayoutPtr        layout;
    u16                 actorID;

    u16                 totalBones;
    DSTree              hierarchy;

    GEOPalettePtr       pal;
    DODisplayObjectPtr  skinObject;

    ACTBonePtr*         boneArray;
    CTRLControl         worldControl;

    MtxPtr              forwardMtxArray;
    MtxPtr              skinMtxArray;
    MtxPtr              skinInvTransposeMtxArray;
    MtxPtr              orientationInvMtxArray;

    DSList              drawPriorityList;

    SKHeader*           skHeader;

} ACTActor, *ACTActorPtr;
```

Code 42 ACTActor

- *layout* – Pointer to the actor layout from which this actor is instanced.
- *actorID* – ID number for this actor.
- *totalBones* – Total number of bones in the actor.
- *hierarchy* – `DSTree` structure which describes the hierarchical relationship of the bones in the actor.
- *pal* – Pointer to the actor's geometry palette in memory.
- *skinObject* – Pointer to a display object which is attached to this actor as its skin. A `NULL` value indicates that no skin is attached to the actor.
- *boneArray* – Array of bones which comprise the hierarchy.
- *worldControl* – Top-level control for the actor. (See section 2.5.2 for more information on controls.)
- *forwardMtxArray* – An array of matrices used by the bones to control rigid display objects. Each bone references one unique matrix in this array.
- *skinMtxArray* – An array of matrices used by the bones to control deformable display objects. Each bone references one unique matrix in this array.
- *skinInvTransposeMtxArray* – An array of matrices used by the bones to control the normals of deformable display objects. Each bone references one unique matrix in this array.
- *orientationInvMtxArray* – An array of precomputed inverse orientation matrices. This array allows for better efficiency when building actor matrices for a skinned object.
- *drawPriorityList* – `DSList` structure describing a list containing all the bones in the actor sorted by display priority.
- *skHeader* – Pointer to skinning header in SKN format if actor is skinned.

2.3.6.4 ACTBone

```
typedef struct
{
    u16                 boneID;
```

```
    u8                  inheritanceFlag;
    u8                  drawingPriority;

    DSBranch            branch;

    DODisplayObjectPtr  dispObj;

    CTRLControl         orientationCtrl;
    CTRLControl         animationCtrl;

    ANIMPipePtr         animPipe;

    MtxPtr              forwardMtx;
    MtxPtr              skinMtx;
    MtxPtr              skinInvTransposeMtx;
    MtxPtr              orientationInvMtx;

    DSLink              drawPriorityLink;

} ACTBone, *ACTBonePtr;
```

Code 43 ACTBone

- *boneID* – Unique ID number for the bone within the actor.
- *inheritanceFlag* – Flag which specifies whether the matrix from the bone's parent should be concatenated with the bone's matrix before rendering it.
- *drawingPriority*– Number stating when this bone's geometry should be drawn in relationship to the other bones in the actor. A lower value for this field indicates a higher drawing priority.
- *branch* – `DSBranch` structure which specifies the bone's position within the hierarchy.
- *dispObj* – Pointer to the display object associated with this bone. A value of `NULL` indicates no geometry is attached to this bone.
- *orientationCtrl* – A control that specifies the bone's default orientation. (See section 2.5.2 for more information).
- *animationCtrl* – A control that specifies the bone's current animation. (See section 2.5.2 for more information).
- *animPipe* – A pointer to an animation pipe. This structure is allocated dynamically when animation is applied to an actor. (For more information on animation pipes, see section 2.5.1.)
- *forwardMtx* – Pointer to a matrix in the actor's *forwardMtxArray*. This matrix is used to control rigid geometry attached to this bone.
- *skinMtx* – Pointer to a matrix in the actor's *skinMtxArray*. This matrix is used to control deformable geometry attached to this bone.
- *skinInvTransposeMtx* – Pointer to a matrix in the actor's *skinInvTransposeMtxArray*. This matrix is used to control the normals of deformable geometry attached to this bone.
- *orientationInvMtx* – Pointer to a matrix in the actor's *orientationInvMtxArray*. This matrix is precomputed when the actor is initialized so that building actor matrices for a skinned object is more efficient.
- *displayPriorityLink* – Structure to fit the bone into the actor's *drawPriorityList*.

2.3.7 Actor API

```
void            ACTGet                  ( ACTActorPtr *actor, char *name );
void            ACTRelease              ( ACTActorPtr *actor );
void            ACTRender               ( ACTActorPtr actor, Mtx camera,  u8 numLights, ... );
void            ACTHide                 ( ACTActorPtr actor );
void            ACTShow                 ( ACTActorPtr actor );
void            ACTBuildMatrices        ( ACTActorPtr actor );
CTRLControlPtr  ACTGetControl           ( ACTActorPtr actor );
void            ACTSetInheritance       ( ACTActorPtr actor, u8 inheritanceFlag );
void            ACTSetAmbientPercentage ( ACTActorPtr actor, f32 percent );
void            ACTSort                 ( ACTActorPtr actor );
```



```
void            ACTSetEffectsShader     ( ACTActorPtr actor, Ptr shaderFunc, Ptr data );
u32             ACTGetUserDataSize      ( ACTActorPtr actor );
Ptr             ACTGetUserData          ( ACTActorPtr actor );
```

Code 44 Actor API

2.3.8 ActorAnim API

```
void ACTSetAnimation  ( ACTActorPtr actor, ANIMBankPtr animBank,
                        char *sequenceName, u16 seqNum, float time );
void ACTSetTime       ( ACTActorPtr actor, float time );
void ACTSetSpeed      ( ACTActorPtr actor, float speed );
void ACTTick          ( ACTActorPtr actor );
```

Code 45 ActorAnim API

2.4 Animation

The Character Pipeline Animation (ANM) format and library provide simple methods to store and retrieve keyframed animation. This library does not perform any of the interpolation or playback features of animation—those are handled at a different level. This library simply provides a way to access the information necessary to properly animate an object.

An animation bank is made up of one or more animation sequences that define whole animations for an actor in the world. These animations can be accessed and used to perform the necessary calculations to perform smooth animation playback in a game.

2.4.1 Animation bank in memory

When an animation bank is unpacked into memory, it comprises three major components in a hierarchical fashion. The animation *bank* contains animation sequences, an animation *sequence* contains animation tracks, and an animation *track* contains *keyframes* that are organized in order of when, in time, they occur.

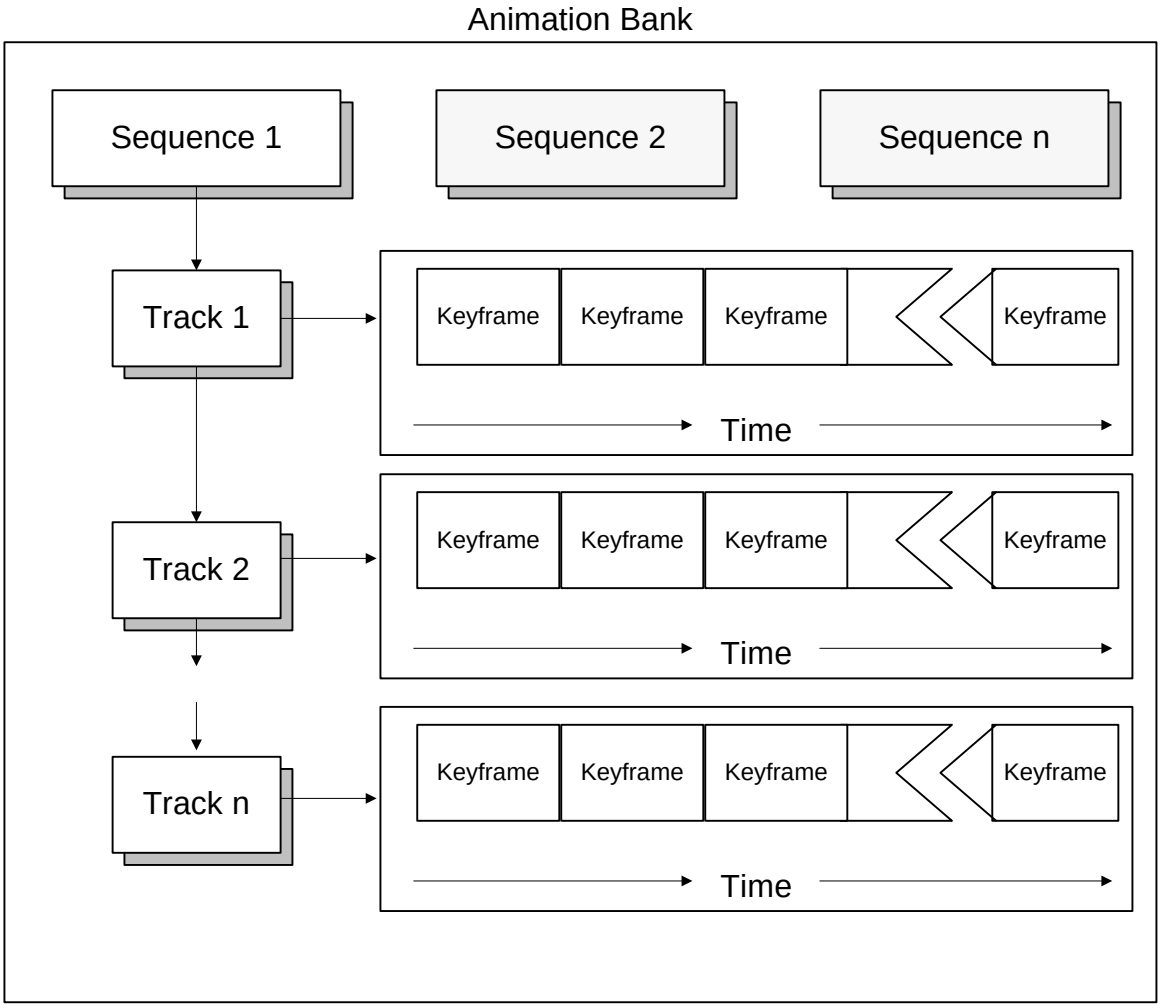


Figure 10 Layout of animation bank in memory

2.4.1.1 Animation sequences

An animation sequence is defined as an entire animation loop for a character. For instance, the animation of a character running is considered to be an animation sequence. A character will probably have multiple sequences such as “run,” “jump,” and “punch.” Animation sequences can be referenced by string name or by sequence ID with the animation bank.

2.4.1.2 Animation tracks

An animation track is defined as all of the information needed to move a single bone through an animation sequence. In general, each bone in an actor should have an animation track for each animation sequence.

2.4.1.3 Keyframes

A keyframe is on instance of animation data within an animation track. A track is made up of multiple keyframes causing the corresponding bone to move through these keyframes as time passes. A keyframe carries information specifying the time at which it occurs in the animation track.

2.4.2 Keyframe data

A keyframe can specify any combination of rotation, translation, and scale. Also, given the type of animation, keyframes can be interpolated differently. The only limitation is that all of the keyframes within any given animation track must carry the same type of information and be interpolated by the same method.

2.4.2.1 Setting data

The setting data for a keyframe specifies the actual values for the given combination of rotation, translation, and scale. This data can be specified in a few different forms:

- Rotation can be specified as Euler angles or quaternions.
- Scale and translation can be supplied as individual values.
- Rotation, scale, and translations can be specified together as a matrix.

2.4.2.2 Interpolation data

Interpolation data for a keyframe is the information needed to interpolate between two consecutive keyframes. Different types of interpolation and interpolation data can be associated with different setting data.

- **Euler Rotation, Translation, Scale** – Euler rotation, translation, and scale can be interpolated using linear, Bezier, or Hermite interpolation. Linear interpolation needs no interpolation data, but Bezier and Hermite require extra parameters to specify how keyframes are interpolated.
- **Quaternion Rotation** – Quaternions can be interpolated using either spherical linear, SQUAD, or SQUAD with ease-in and ease-out interpolation (SQUADEE). SLERP requires no extra interpolation data, but SQUAD and SQUADEE require extra parameters to specify how keyframes are interpolated.
- **Matrix** – Matrices cannot be interpolated and therefore do not require any interpolation data.

2.4.3 Quantization

Both keyframe data and interpolation data can be quantized from floating point to 16-bit or 8-bit fixed point to save memory. Normals and quaternions are assumed to be 2.6 or 2.14 depending on whether they are quantized to 8-bit or 16-bit, respectively. For all other data, a shift value must be explicitly specified. Note that all interpolation data must be quantized in the same way. Setting data must also all be quantized in the same way.

2.4.4 AnimBank structures

2.4.4.1 ANIMBank

```
typedef struct
{
    u32              versionNumber;

    ANIMSequencePtr  animSequences;

    u16              bankID;
    u16              numSequences;
    u16              numTracks;
    u16              numKeyFrames;

    u32              userDataSize;
    void*            userData;

} ANIMBank, *ANIMBankPtr;
```

Code 46 ANIMBank

- *versionNumber* – Field to describe the version number of the file.
- *animSequences* – Array of animation sequences.
- *bankID* – ID number for the animation bank.
- *numSequences* – Number of sequences contained within the animation bank.
- *numTracks* – Number of tracks contained within the animation bank.
- *numKeyFrames* – Number of keyframes contained within the animation bank.
- *userDataSize* – Size of the ANM user-defined data.
- *userData* – Pointer to the ANM user-defined data.

2.4.4.2 ANIMSequence

```
typedef struct
{
    char         *sequenceName;

    ANIMTrackPtr  trackArray;

    u16           totalTracks;
    u16           pad16;

} ANIMSequence, *ANIMSequencePtr;
```

Code 47 ANIMSequence

- *sequenceName* – String name uniquely identifying the sequence within the animation bank.
- *trackArray* – Array of animation tracks.
- *totalTracks* – Total number of tracks for this sequence.

2.4.4.3 ANIMTrack

```
typedef struct
{
    float            animTime;

    ANIMKeyFramePtr  keyFrames;

    u16              totalFrames;
    u16              trackID;

    u8               quantizeInfo;
    u8               animType;
    u8               interpolationType;
    u8               replaceHierarchyCtrl;

} ANIMTrack, *ANIMTrackPtr;
```

Code 48 ANIMTrack

- *animTime* – Total time duration for this animation track.
- *keyFrames* – Keyframe array.
- *totalFrames* – Total number of keyframes for this animation track.
- *trackID* – ID number for this animation track.
- *quantizeInfo* – Information about how the keyframe data is quantized. (For more information on keyframe quantization, see section 2.4.3.)
- *animType* – Specifies which control parameters are animated by this animation track. (For more information on this field, see section 2.4.2.1.)
- *interpolationType* – Specifies the method of interpolation for keyframe data. (For more information on keyframe interpolation, see section 2.4.2.2.)
- *replaceHierarchyCtrl* – Specifies whether the resulting animation control should replace the hierarchy control or be concatenated with it (making animation relative to orientation). This flag depends on how animation data is converted from the CG tools; currently, it should always be TRUE. Note that this flag is maintained for backward compatibility, since the earlier animation library was more inefficient and restrictive in making animation relative to hierarchy matrices.

2.4.4.4 ANIMKeyFrame

```
typedef struct
{
    float  time;
    Ptr    setting;
    Ptr    interpolation;

} ANIMKeyFrame, *ANIMKeyFramePtr;
```

Code 49 ANIMKeyFrame

- *time* – Time during the animation at which this keyframe occurs.
- *setting* – Pointer to the actual keyframe data. This data is made up of animation data for the control parameters specified by *animType* in the animation track that contains this keyframe. This data is quantized according to *quantizeInfo* in the animation track that contains this keyframe. (For more information on keyframe setting data, see section 2.4.2.1.)

- *interpolation* – Pointer to the interpolation data needed to interpolate the setting data by the method specified by *interpolationType* in the animation track that contains this keyframe. (For more information on keyframe interpolation data, see section 2.4.2.2.

2.4.5 AnimBank API

```
void             ANIMGet                 ( ANIMBankPtr *animBank, char *name );
void             ANIMRelease             ( ANIMBankPtr *animBank );
void             ANIMGetKeyFrameFromTrack ( ANIMTrackPtr animTrack, float time,
                                            ANIMKeyFramePtr *currentFrame,
                                            ANIMKeyFramePtr *nextFrame );
ANIMSequencePtr  ANIMGetSequence         ( ANIMBankPtr animBank, char *sequenceName,
                                            u16 seqNum );
ANIMTrackPtr     ANIMGetTrackFromSeq     ( ANIMSequencePtr animSeq, u16 animTrackID );
u32              ANIMGetUserDataSize     ( ANIMBankPtr animBank );
Ptr              ANIMGetUserData         ( ANIMBankPtr animBank );
```

Code 50 Animation bank API

2.5 Support

The Character Pipeline support libraries are libraries that are used by the major libraries (geometry, actor, animation, and texture), but might not be readily visible at the top level.

2.5.1 Animation pipe

The Character Pipeline animation pipe library provides the backbone for the Character Pipeline's animation capabilities. In the Character Pipeline, any object that is to be animated using an animation bank must contain an animation pipe.

2.5.1.1 Function of the animation pipe

The function of the animation pipe is to bind an animation control to an animation track, and then correctly update the control with interpolated data from the track as necessary. The animation pipe maintains state relevant to the animation, such as the current time and speed. Also, this library keeps and executes all of the code to interpolate keyframe data.

2.5.1.2 How an animation pipe binds to a control/animation track

When an animation pipe is bound to a control and a track, it sets the type of parameters the control accepts (see section 2.5.2) based on the type of animation track to which it is bound. Once this binding has occurred, the animation pipe may be "ticked," meaning that it can be told to update the control's parameters with interpolated animation information from the animation track.

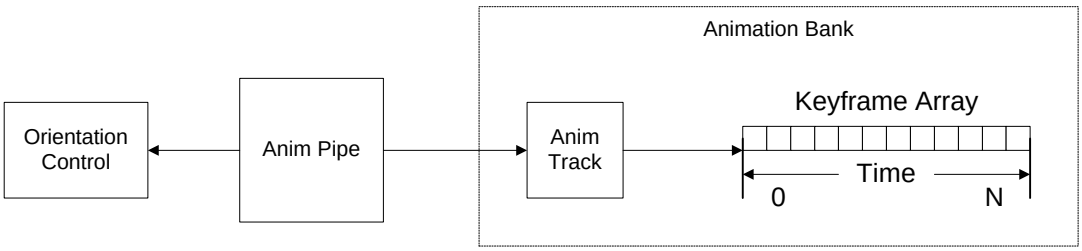


Figure 11 Animation pipe bound to control and animation track

2.5.1.3 Understanding an animation pipe's parameters

The animation pipe keeps a few parameters that are key to the playback of animation. It is good to know about the following when using the animation pipe:

- **Time** – This is the current time stamp of the animation within the animation track. The animation pipe processing will handle wrapping this time back around to the beginning of the track when it reaches the end.
- **Speed** – The speed parameter specifies the current time increment at which the animation is playing. Make this increment smaller and the animation will play more slowly; make it greater and the animation will play faster. This increment is added to the animation pipe's time after a frame of animation is processed.

2.5.1.4 How an animation pipe interpolates keyframe data

When an animation pipe is instructed to update its associated control with animation data, it starts by finding the start frame and end frame from the animation track as discussed above. The setting data from these two keyframes is then interpolated using the interpolation data from the two keyframes, and based on how the data is specified as being interpolated in the animation track. Once this data has been properly interpolated, it is stored in the correct manner in the control.

2.5.1.4.1 Currently supported interpolation methods

The animation pipe library currently supports the following types of interpolation:

- **None** – It is possible to not interpolate keyframe data at all. In this case, the data from the start frame is used regardless of where the start and end frames occur relative to the current time. Usually, this method is used only if the keyframe setting data uses entire matrices to describe animation.
- **Linear** – Linear interpolation can be used on data for scale, translation, and Euler rotation. Linear interpolation requires no extra interpolation data.
- **Bezier** – Bezier interpolation smoothly interpolates values along a curve specified by an in-tangent and an out-tangent for each keyframe. Bezier interpolation can be used for scale, translation, and Euler rotation data. The in-tangent and out-tangent are specified for each data member in the interpolation data section of a keyframe.
- **Hermite** – Hermite interpolation smoothly interpolates Kochanek-Bartel cubic splines, otherwise known as TCB (tension, continuity, bias) cubic splines. In this implementation, Hermite interpolation also carries two extra parameters in the interpolation data; these are ease-in and ease-out values for each data member to be interpolated. Hermite interpolation can be used for scale, translation, and Euler rotation data.
- **SLERP** – Spherical, Linear Interpolation is used to interpolate quaternions linearly. SLERP is used only to interpolate quaternion values and requires no extra interpolation data in the keyframe.

- **SQUAD** – Spherical, Quadratic Interpolation is, effectively, the quaternion equivalent of Bezier interpolation for Euler angles. SQUAD interpolation smoothly interpolates quaternions along a specified curve, requiring an in-quaternion and an out-quaternion for each quaternion keyframe.
- **SQUADEE** – SQUADEE interpolation is, effectively, the quaternion equivalent of Hermite interpolation for Euler angles. In addition to an in-quaternion and an out-quaternion, SQUADEE requires an ease-in and ease-out value for each parameter being interpolated.

2.5.1.5 Animation pipe structures

2.5.1.5.1 ANIMPipe

```
typedef struct ANIMPipe
{
    float           time;
    float           speed;

    ANIMTrackPtr    currentTrack;

    CTRLControlPtr  control;

} ANIMPipe, *ANIMPipePtr;
```

Code 51 ANIMPipe

- *time* – The current time for the animation being played by the animation pipe.
- *speed* – The current speed of the animation being played by the animation pipe.
- *currentTrack* – Pointer to the animation track which is currently bound to the animation pipe.
- *control* – Pointer to the control to which the animation pipe is currently bound.

2.5.1.6 Animation pipe API

```
void  ANIMBind      ( ANIMPipePtr animPipe, CTRLControlPtr control,
                      ANIMTrackPtr animTrack, float time );
void  ANIMSetTime   ( ANIMPipePtr animPipe, float time );
void  ANIMSetSpeed  ( ANIMPipePtr animPipe, float speed );
void  ANIMTick      ( ANIMPipePtr animPipe );
```

Code 52 Animation pipe API

2.5.2 Control

The Character Pipeline control library provides some abstraction on top of the MTX library so that you can easily build and control a matrix. The control library is also more efficient than MTX in two ways:

- Matrices are built arithmetically instead of using costly matrix concatenations to combine translation, rotation, and scale components.
- Matrices are built selectively; for example, if rotation is not set in the control, then the code to integrate rotation is not called.

2.5.2.1 Understanding ways to use a control

A control can be used in three different forms to compute a matrix:

1. Scale, Euler (*x, y, z*) rotation, and/or translation, using the following functions:
   - `CTRLSetScale`
   - `CTRLSetRotation`
   - `CTRLSetTranslation`

2. Scale, quaternion rotation, and/or translation, using the following functions:
   - `CTRLSetScale`
   - `CTRLSetQuat`
   - `CTRLSetTranslation`

3. Matrix using the following function:
   - `CTRLSetMatrix`

The type of the control is set automatically depending on which of the three preceding forms you use. The control structure is a union of the structures that represent these three methods of using a control. Before a control is used, however, it must be initialized with `CTRLInit` to clear the control type data structures.

2.5.2.2 Yielding a matrix from the control

Once the control parameters are set with one of the three forms specified in section 2.5.2.1, a control can be instructed to build the matrix using `CTRLBuildMatrix`. If no parameters are set, the control will build an identity matrix. These data values are not cleared or re-initialized until the type of the control is reset. This means that data remains resident in a control until it is specifically overwritten. Matrices are produced in the order of scale, rotation (Euler (*x, y, z*) or quaternion), then translation. Inverse matrices can also be built arithmetically using the function `CTRLBuildInverseMatrix`.

- For a code example, see section 1.8.2.1.

2.5.2.3 Control structures

2.5.2.3.1 CTRLControl

```
typedef struct
{
    u8     type;
    u8     pad8;
    u16    pad16;

    union
    {
        CTRLSRTControl    srt;
        CTRLMTXControl    mtx;

    } controlParams;

} CTRLControl, *CTRLControlPtr;
```

Code 53 CTRLControl

- *type* – Specifies the current type of the control; also shows which parameters in the control are valid for building a matrix. There are four valid types which are set automatically by the control library:

  1. `CTRL_NONE`, which yields an identity matrix.
  2. `CTRL_SCALE | CTRL_ROT_EULER | CTRL_TRANS`
  3. `CTRL_SCALE | CTRL_ROT_QUAT | CTRL_TRANS`
  4. `CTRL_MTX`

- *controlParams* – Union containing data with which a matrix can be built.

2.5.2.3.2 CTRLMTXControl

```
typedef struct
{
    Mtx m;

} CTRLMTXControl, *CTRLMTXControlPtr;
```

Code 54 CTRLMTXControl

- *m* – A matrix as defined in `mtx.h`.

2.5.2.3.3 CTRLSRTControl

```
typedef struct
{
    Vec        s;
    Quaternion r;       // Also used for Euler (x, y, z) parameters (typecast as Vec)
    Vec        t;

} CTRLSRTControl, *CTRLSRTControlPtr;
```

Code 55 CTRLSRTControl

- *s* – Floating point values describing scale in (*x*, *y*, *z*), respectively.
- *r* – Floating point values describing quaternion or Euler rotation in (*x*, *y*, *z*), respectively.
- *t* – Floating point values describing translation in (*x*, *y*, *z*), respectively.

2.5.2.4 Control API

```
void CTRLInit               ( CTRLControlPtr control );                 // This is a macro
void CTRLDisable            ( CTRLControlPtr control, u8 type );        // This is a macro
void CTRLEnable             ( CTRLControlPtr control, u8 type );        // This is a macro

void CTRLSetScale           ( CTRLControlPtr control, float x, float y, float z );
void CTRLSetRotation        ( CTRLControlPtr control, float x, float y, float z );
void CTRLSetQuat            ( CTRLControlPtr control, float x, float y, float z, float w );
void CTRLSetTranslation     ( CTRLControlPtr control, float x, float y, float z );
void CTRLSetMatrix          ( CTRLControlPtr control, Mtx m );

void CTRLGetScale           ( CTRLControlPtr control, float *x, float *y, float *z );
void CTRLGetRotation        ( CTRLControlPtr control, float *x, float *y, float *z );
void CTRLGetQuat            ( CTRLControlPtr control, float *x, float *y, float *z, float *w );
void CTRLGetTranslation     ( CTRLControlPtr control, float *x, float *y, float *z );
```



```
void CTRLGetMtx             ( CTRLControlPtr control, MtxPtr mtxPtr );  // This is a macro
void CTRLBuildMatrix        ( CTRLControlPtr control, Mtx m );
void CTRLBuildInverseMatrix ( CTRLControlPtr control, Mtx m );
```

Code 56 Control API

2.5.3 Light

The Character Pipeline Light library serves to mirror the GX API while adding extra functionality, such as logical positioning, manipulation, and animation of lights. The light library only models local lights, not infinite lights. If you want a light to be infinite, simply place it very far from the scene with no attenuation. Also, the Light library supports both distance and angle attenuation.

2.5.3.1 Controlling a light

A light structure carries a control structure so that the light may be manipulated by a matrix to produce animation. The matrix yielded by this control transforms the light's position and direction before the light is transformed by the camera matrix. This allows the user to specify a light in local space, then position and orient it based on the values set in the control.

2.5.3.2 Understanding the parameters of a light

A light structure carries many parameters necessary to the GX API for the proper description of a light. The important data members include:

- **Position** – Position data determines where in space the light exists. The position will be transformed by the matrix specified by the light's control (see the previous section), then concatenated with the camera matrix before it is sent to the GX API. If the light's control specifies an identity matrix, this position can be considered as specified in world space; otherwise, this position can be considered as being in the light's local space.
- **Direction** – Direction data determines the direction the light is pointing. This direction will be transformed by the inverse-transpose of the matrix specified by the light's control (see the previous section), then concatenated with the camera matrix before it is sent to the GX API. If the light's control specifies an identity matrix, this direction can be considered as specified in world space; otherwise, this direction can be considered as being in the light's local space.
- **Color** – Color data simply specifies the color of the light.

2.5.3.3 Transforming a light

Before the GCN hardware can use a light, it must be transformed into camera space. The `LITXForm` function takes a light pointer and a camera matrix and performs this calculation for you. Also, prior to transforming the light by the camera matrix, this function builds a matrix from the light's control and transforms the light by it. The application should call this function for a light before passing it to any rendering routine.

2.5.3.4 Animating a light with an animation bank

In addition to manipulating a light by hand at runtime, it is possible to bind an animation track to the light's control and play it back using the standard Character Pipeline animation mechanisms. Note that when an animation sequence is bound to a light, the light will use animation track 0 from the sequence.

2.5.3.5 Light structures

2.5.3.5.1 LITLight

```
typedef struct
{
    GXLightObj   lt_obj;

    Vec          position;
    Vec          worldPosition;

    Vec          direction;
    Vec          worldDirection;

    GXColor      color;

    CTRLControl  control;
    MtxPtr       parent;

    ANIMPipePtr  animPipe;

} LITLight, *LITLightPtr;
```

Code 57 LITLight

- *lt_obj* – Structure used to pass light information to hardware.
- *position* – Position of the light in local space.
- *worldPosition* – Position of the light in world space (after transformation by the matrix specified by control and parent).
- *direction* – Direction of the light in local space.
- *worldDirection* – Direction of the light in world space (after transformation by the inverse transpose of the matrix specified by control and parent).
- *color* – Color of the light.
- *control* – Control used to position and orient the light in world space.
- *parent* – Pointer to the matrix of the light's parent. This matrix is concatenated with the matrix specified by the control to yield the final light matrix.
- *animPipe* – Pointer to the animation pipe used to animate the light. This pointer remains `NULL` until the light is first bound to animation, at which point memory is dynamically allocated for the structure.

2.5.3.5.2 CPParentType

```
typedef enum
{
    PARENT_BONE,
    PARENT_DISP_OBJ,
    PARENT_MTX

} CPParentType;
```

Code 58 CPParentType

2.5.3.6 Light API

```
void            LITAlloc            ( LITLightPtr *light );
void            LITFree             ( LITLightPtr *light );
void            LITInitAttn         ( LITLightPtr light, f32 a0, f32 a1, f32 a2,
                                      f32 k0, f32 k1, f32 k2 );
void            LITInitSpot         ( LITLightPtr light, f32 cutoff, GXSpotFn spot_func );
void            LITInitDistAttn     ( LITLightPtr light, f32 ref_distance,
                                      f32 ref_brightness, GXDistAttnFn dist_func );
void            LITInittPos         ( LITLightPtr light, f32 x, f32 y, f32 z );
void            LITInittDir         ( LITLightPtr light, f32 nx, f32 ny, f32 nz );
void            LITInitColor        ( LITLightPtr light, GXColor color );
void            LITXForm            ( LITLightPtr light, Mtx view );
void            LITAttach           ( LITLightPtr light, Ptr parent, CPParentType type );
void            LITDetach           ( LITLightPtr light );
CTRLControlPtr  LITGetControl       ( LITLightPtr light );
void            LITSetAnimSequence  ( LITLightPtr light, ANIMBankPtr animBank,
                                      char *seqName, float time );
void            LITTick             ( LITLightPtr light );
```

Code 59 Light API

2.5.4 Shader

The Character Pipeline Shader library sits on top of the GX API and serves to abstract much of the complexity behind the Texture Environment (TEV). The goal of this library is to allow users to define complex color blending equations without worrying about the resource limitations of the TEV. Also, the Shader library allows you to build complex operations into complex shaders that you can easily apply to multiple objects in the scene without having to completely redefine the effect.

2.5.4.1 Definition of a shader

A *shader* is defined simply as some expression that results in an RGB or alpha value. This means that a shader may be as simple as returning a constant color, or as complex as combining eight textures, two rasterized colors, and three constant colors to produce some desired result.

2.5.4.2 Shader channels

As mentioned above, a shader results in either an RGB or an alpha value; therefore, the application must specify which one a shader will return. When creating shaders, simply specify to which of these two channels (RGB or alpha) the shader belongs. RGB shaders can take input from other RGB shaders as well as alpha shaders, but alpha shaders can take input only from other alpha shaders.

2.5.4.3 Shader types

Because there are a few different types of data that can be used in shaders, we've defined a few of the simple ones to aid understanding. These simple shader types are:

- **Trivial –** A trivial shader simply returns a constant value. The possible constant values are 1.0, 0.0, 0.5, and 0.25. The constant values 0.5 and 0.25 cannot be used in an alpha shader. Trivial shaders, unlike all other

shaders, cannot be created. Instead, they are statically allocated globally. The global names for these shaders are `ShaderZero`, `ShaderOne`, `ShaderHalf`, and `ShaderQuarter`, respectively.

- **Constant –** A constant shader takes a `GXColor` as input when it is created, and simply returns that same color during triangle rasterization.
- **Rasterized –** A rasterized shader takes a color channel as input when it is created, and returns the interpolated color for the current triangle at the current position.
- **Texture –** A texture shader takes a `GXTexObj` and a texture coordinate expression as input when it is created. It returns the processed color for the current triangle at the current position.
- **Complex Input –** A complex input shader takes a compiled shader as input and returns the result.

2.5.4.4 Complex shaders

At the heart of the Shader library is the complex shader. The complex shader allows other shaders to be combined in some mathematical manner. A complex shader takes input from four other shaders and receives some combination information when it is created, then combines the results of those individual shaders in the manner specified. The following diagram shows how the inputs into a complex shader may be combined:

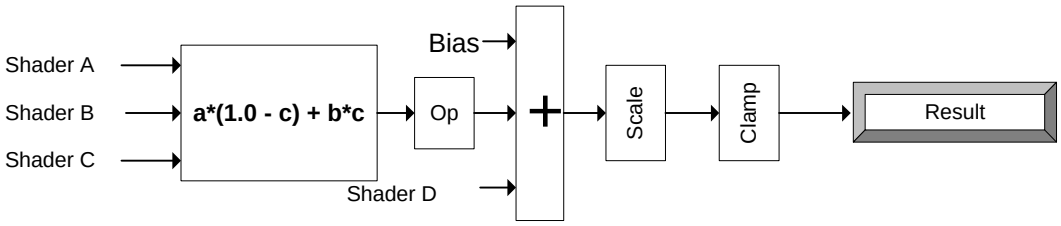


Figure 12 Shader combination method

As shown in Figure 12, here are the combination parameters that need to be given to a complex shader:

- **Op** – Plus or minus.
- **Bias –** Some value by which the result of the equation is biased.
- **Scale –** Some value by which the result of the biased equation is scaled.
- **Clamp –** A method by which the scaled, biased equation is clamped.

As mentioned earlier, complex shaders take four other shaders as inputs. It is important to note that any or all of these inputs can be other complex shaders. This allows the user to build up complex expression trees.

2.5.4.5 Compiling shaders

When shaders are specified, their inputs are specified generically. Real data is bound to these generics inputs later so that specific operations can be performed. Shaders are meant to be compiled at loadtime and bound and executed at runtime.

2.5.4.6 Binding shaders

At runtime, real application data must be bound to the generic inputs of a shader. A shader will not execute unless all of its generic inputs are properly bound.

2.5.4.7 Executing shaders

Once an entire shader has been compiled and bound to valid application data, it must be executed to set the proper hardware state in order to carry out the specified expression. When the execute function is called, the shader's compiled hardware tokens are passed to the GX API.

2.5.4.8 Shader API

```
typedef enum
{
    SHADER_TG_POS,
    SHADER_TG_NRM,
    SHADER_TG_BINRM,
    SHADER_TG_TANGENT,
    SHADER_TG_TEX0,
    SHADER_TG_TEX1,
    SHADER_TG_TEX2,
    SHADER_TG_TEX3,
    SHADER_TG_TEX4,
    SHADER_TG_TEX5,
    SHADER_TG_TEX6,
    SHADER_TG_TEX7,
    SHADER_TG_COLOR0,
    SHADER_TG_COLOR1

} ShaderTexGenSrc;
```

```
typedef enum
{
    SHADER_TG_MTX3x4,
    SHADER_TG_MTX2x4,
    SHADER_TG_BUMP0,
    SHADER_TG_BUMP1,
    SHADER_TG_BUMP2,
    SHADER_TG_BUMP3,
    SHADER_TG_BUMP4,
    SHADER_TG_BUMP5,
    SHADER_TG_BUMP6,
    SHADER_TG_BUMP7,
    SHADER_TG_SRTG

} ShaderTexGenType;
```

```
typedef enum
{
    SHADER_MTX0,
    SHADER_MTX1,
    SHADER_MTX2,
    SHADER_MTX3,
    SHADER_MTX4,
    SHADER_MTX5,
```

```
    SHADER_MTX6,
    SHADER_MTX7,
    SHADER_IDENTITY

} SHADER_MTXInput;
```

```
typedef enum
{
    SHADER_OP_ADD,
    SHADER_OP_SUB

} SHADER_OP;
```

```
typedef enum
{
    SHADER_CLAMP_LINEAR_1023,
    SHADER_CLAMP_LINEAR_255,
    SHADER_CLAMP_GE_255,
    SHADER_CLAMP_GE_0,
    SHADER_CLAMP_EQ_255,
    SHADER_CLAMP_EQ_0,
    SHADER_CLAMP_LE_255,
    SHADER_CLAMP_LE_0

} SHADER_CLAMP;
```

```
typedef enum
{
    SHADER_BIAS_ZERO,
    SHADER_BIAS_ADDHALF,
    SHADER_BIAS_SUBHALF

} SHADER_BIAS;
```

```
typedef enum
{
    SHADER_SCALE_1,
    SHADER_SCALE_2,
    SHADER_SCALE_4,
    SHADER_SCALE_DIVIDE_2

} SHADER_SCALE;
```

```
typedef enum
{
    SHADER_CHANNEL_RGB,
    SHADER_CHANNEL_RRR,
    SHADER_CHANNEL_GGG,
    SHADER_CHANNEL_BBB,
    SHADER_CHANNEL_A

}SHADER_CHANNEL;
```

```
typedef enum
{
    SHADER_TEX0,
```



```
    SHADER_TEX1,
    SHADER_TEX2,
    SHADER_TEX3,
    SHADER_TEX4,
    SHADER_TEX5,
    SHADER_TEX6,
    SHADER_TEX7

} SHADER_TexInput;
```

```
typedef enum
{
    SHADER_RAS0,
    SHADER_RAS1

} SHADER_RasInput;
```

```
typedef enum
{
    SHADER_COLOR0,
    SHADER_COLOR1,
    SHADER_COLOR2,
    SHADER_COLOR3

} SHADER_ColorInput;
```

```
typedef enum
{
    SHADER_COMPLEX0,
    SHADER_COMPLEX1,
    SHADER_COMPLEX2,
    SHADER_COMPLEX3,
    SHADER_COMPLEX4,
    SHADER_COMPLEX5,
    SHADER_COMPLEX6,
    SHADER_COMPLEX7

} SHADER_ComplexInput;
```

```
SHDRTexCoord *SHDRCreateTexCoordExpression   ( ShaderTexGenSrc src,
                                               SHDRTexCoord *shadSrc,
                                               ShaderTexGenType texGenType,
                                               SHADER_MTXInput mtxInput );

SHDRTexCoord *SHDRCreateTCPassThrough        ( ShaderTexGenSrc src );

SHDRShader *SHDRCreateTexture        ( SHADER_TexInput tex, SHDRTexCoord *texCoordShader,
                                                            SHADER_CHANNEL channel );

SHDRShader *SHDRCreateRasterized     ( SHADER_RasInput rasColor, SHADER_CHANNEL channel );
SHDRShader *SHDRCreateColor          ( SHADER_ColorInput color, SHADER_CHANNEL channel );
SHDRShader *SHDRCreateComplexInput   ( SHADER_ComplexInput input, SHADER_CHANNEL channel );

SHDRShader *SHDRCreateComplex        ( SHDRShader *input1, SHDRShader *input2,
                                       SHDRShader *input3, SHDRShader *input4,
                                       SHADER_OP op, SHADER_CLAMP clamp,
                                       SHADER_BIAS bias, SHADER_SCALE scale,
                                       SHADER_CHANNEL channel );

void    SHDRFree                     ( SHDRShader *shader );
```

```
SHDRCompiled    *SHDRCompile          ( SHDRShader *rgbShader, SHDRShader *aShader );

void SHDRBindTexture          ( SHDRCompiled *shader, SHADER_TexInput tex,
                                GXTexObj *texObj );

void SHDRBindRasterized       ( SHDRCompiled *shader, SHADER_RasInput rasColor,
                                GXChannelID channel );

void SHDRBindColor            ( SHDRCompiled *shader, SHADER_ColorInput colorInput,
                                GXColor color );

void SHDRBindComplexInput     ( SHDRCompiled *shader, SHADER_ComplexInput input,
                                SHDRCompiled *inputShader );

void SHDRBindTexGenMtx        ( SHDRCompiled *shader, SHADER_MTXInput input, Mtx mtxData );

void SHDRExecute              ( SHDRCompiled *shader );
```

Code 60 Shader API

3 Formats

3.1 Display object

This section presents the GPL in its byte-wise file format. In the following figures, the figure caption text within parentheses shows the structure used to access the GPL format in memory. For more details, see section 2.1.10.

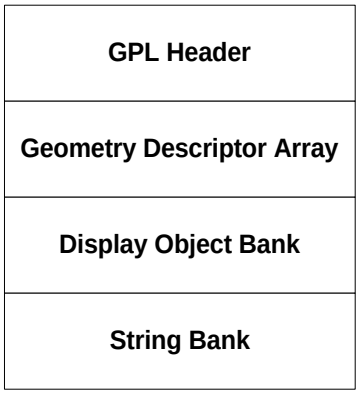


Figure 13 GPL overview

Size (in bytes)	Meaning
4	Version number.
4	User-defined data size.
4	Offset to user data (32-byte aligned).
4	Number of geometry descriptors.
4	Offset to geometry descriptor array (8).

Table 2 GPL header (GEOPalette)

Size (in bytes)	Meaning
4	Offset to display object.
4	Offset to the string name of the display object.

Table 3 Geometry descriptor (GEODescriptor)

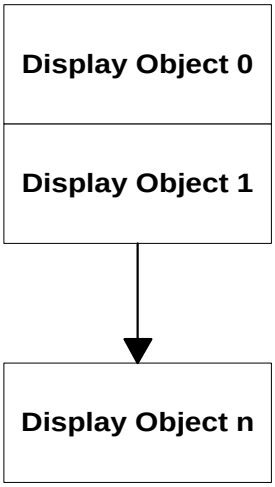


Figure 14 Display object bank

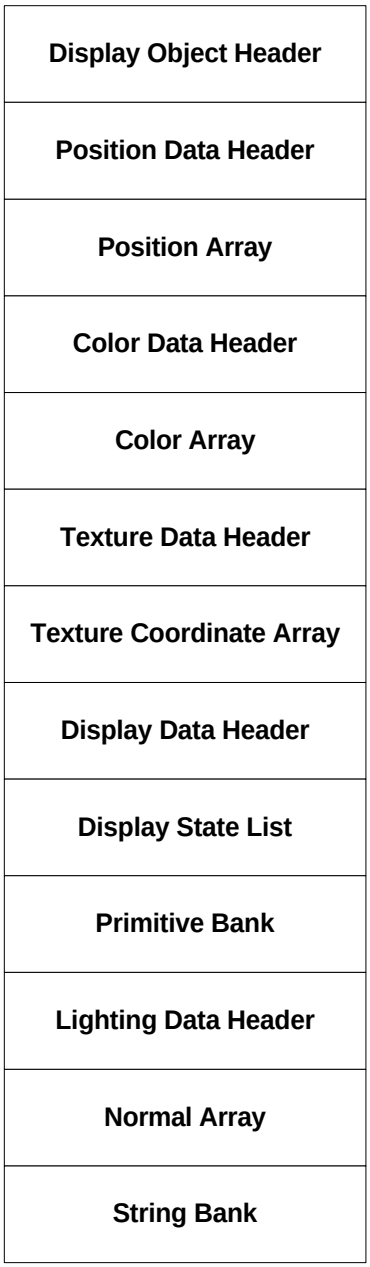


Figure 15 Display object overview

Size (in bytes)	Meaning
4	Offset to position data header from the beginning of the display object (24).
4	Offset to color data header from the beginning of the display object.
4	Offset to texture data header from the beginning of the display object.
4	Offset to lighting data header from the beginning of the display object.
4	Offset to display data header from the beginning of the display object.
1	Number of texture headers (for multitexturing).
1	Pad.
2	Pad.

Table 4 Display object header (DODisplayObjectLayout or DOLayout)

Size (in bytes)	Meaning
4	Offset to position array from the beginning of the display object.
2	Number of positions.
1	Quantize info (see Figure 19).
1	Number of components (2 or 3).

Table 5 Position data header (DOPositionHeader)

Size (in bytes)	Meaning
4	Offset to color array (from beginning of display object).
2	Number of colors.
1	Quantize info (see Figure 19).
1	Number of components (3 or 4).

Table 6 Color data header (DOColorHeader)

Size (in bytes)	Meaning
4	Offset to texture coordinate array (from beginning of display object).
2	Number of texture coordinates.
1	Quantize info (see Figure 19).
1	Number of components (1 or 2).
4	Offset to texture palette file name (from beginning of display object)
4	Pad (reserved for runtime usage).

Table 7 Texture data header (DOTextureDataHeader)

Size (in bytes)	Meaning
4	Offset to primitive bank (from beginning of display object).
4	Offset to display state list (from beginning of display object).
2	Number of display state list entries.
2	Pad.

Table 8 Display data header (DODisplayHeader)

Size (in bytes)	Meaning
4	Offset to normal array (from beginning of display object).
2	Number of normals.
1	Quantize info (see Figure 19).
1	Number of components (2 or 3).
4	Ambient percentage (0.0 – 100.0).

Table 9 Lighting data header (DOLightingHeader)

Size (in bytes)	Meaning
1	State ID.
3	Pad.
4	Setting.
4	Offset to primitive list from the beginning of the display object.
4	Length of primitive list (in bytes).

Table 10 Display state entry (DODisplayState)

State ID	Setting
`DISPLAY_STATE_TEXTURE`	See Figure 16.
`DISPLAY_STATE_TEXTURE_COMBINE`	`GXTevMode`.
`DISPLAY_STATE_VCD`	See Figure 17.
`DISPLAY_STATE_MTXLOAD`	See Figure 18.

Table 11 Display state settings

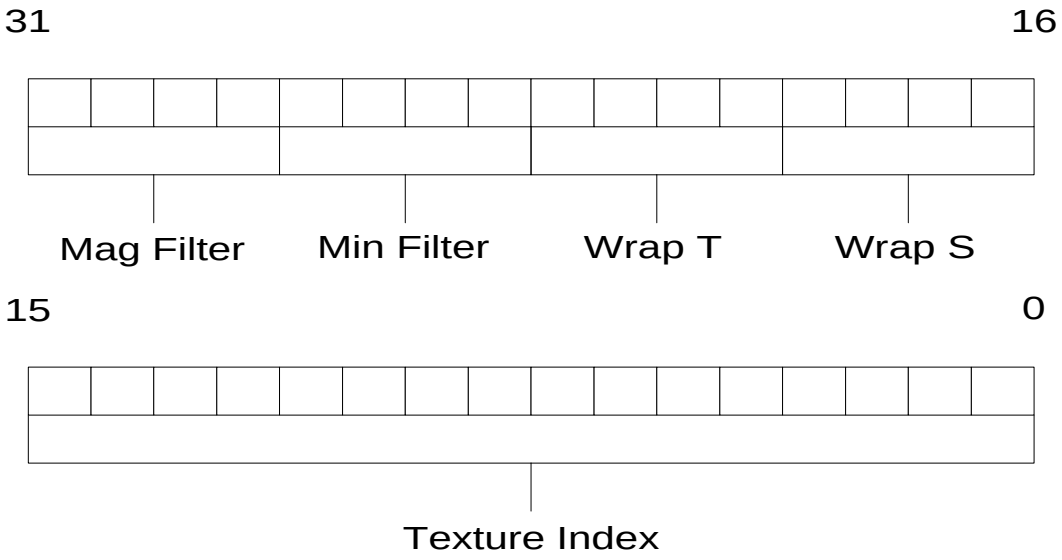


Figure 16 Setting for DISPLAY_STATE_TEXTURE

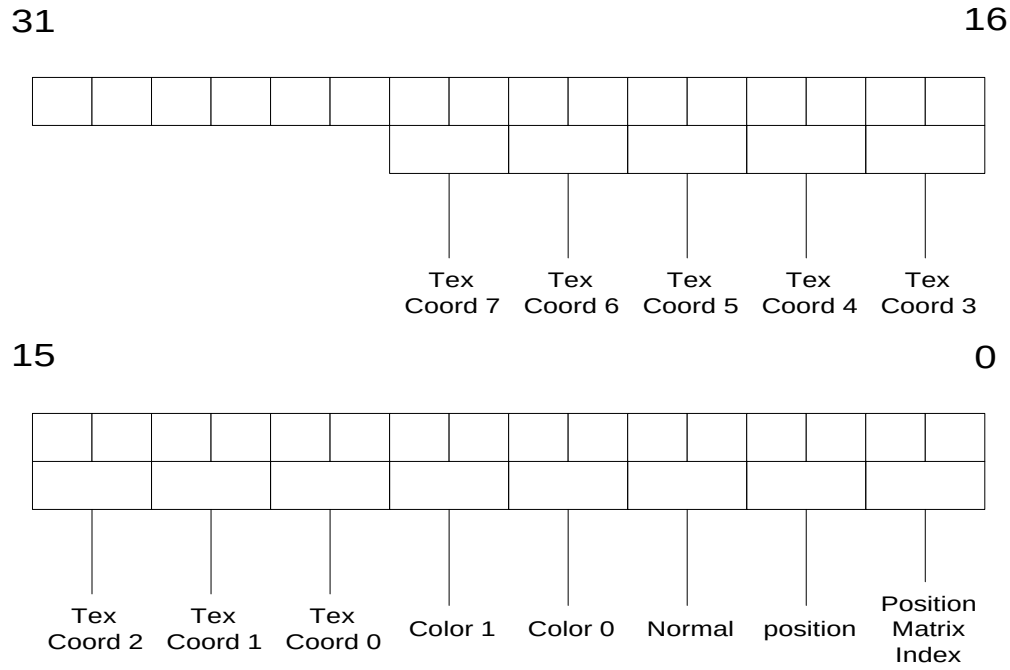


Figure 17 Setting for DISPLAY_STATE_VCD

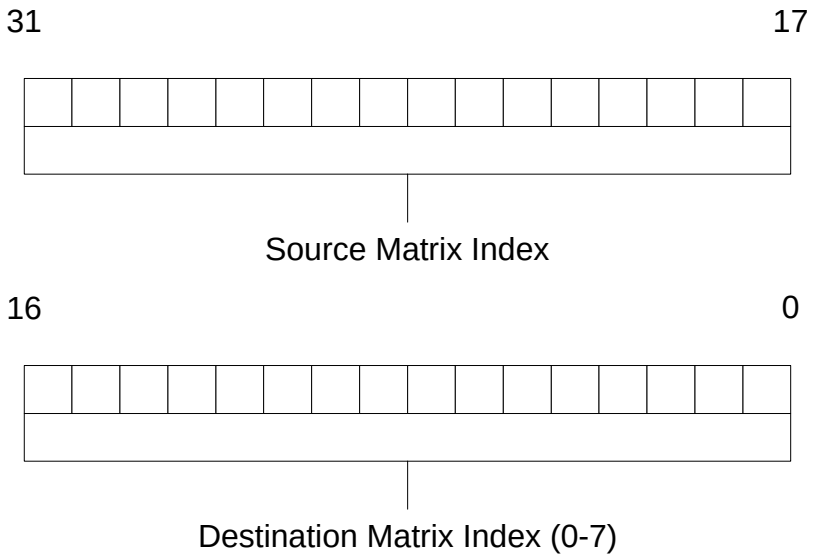


Figure 18 Setting for DISPLAY_STATE_MTXLOAD

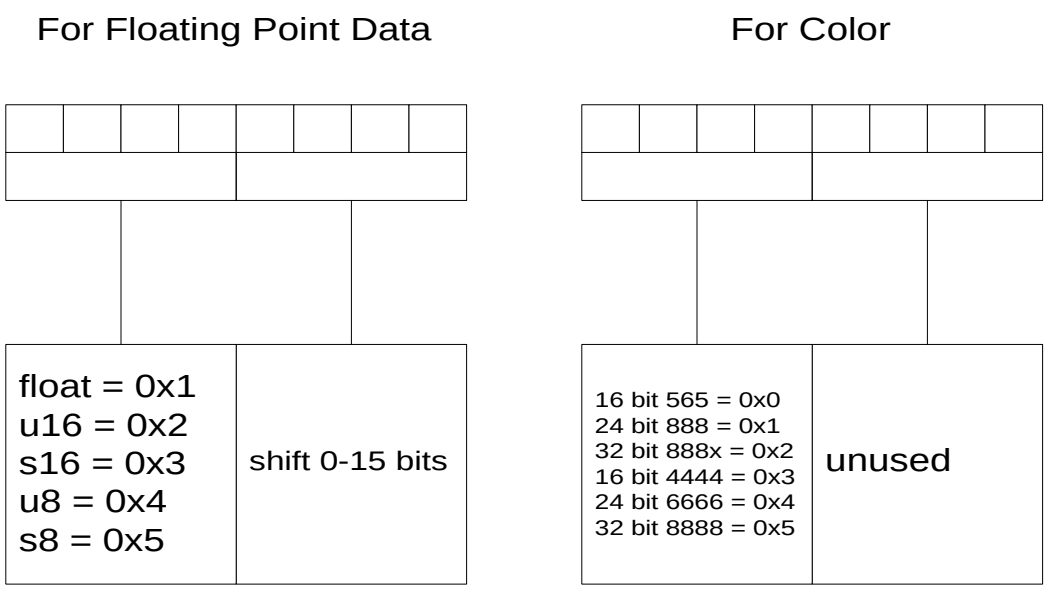


Figure 19 Quantization data layout

3.2 Texture

This section presents the TPL in its byte-wise file format. In the following figures, the figure caption text within parentheses shows the structure used to access the TPL format in memory. For more details, refer to section 2.2.5.

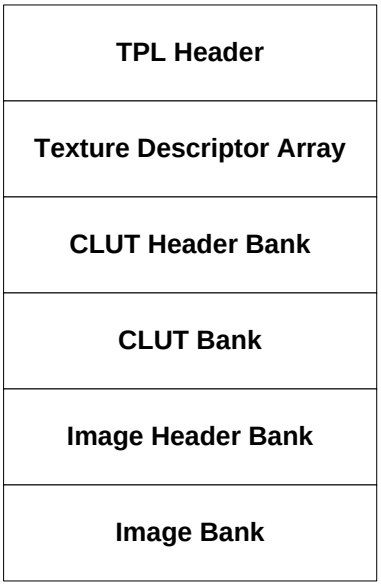


Figure 20 TPL overview

Size (in bytes)	Meaning
4	Version number.
4	Number of texture descriptors.
4	Offset to texture descriptor array (8).

Table 12 TPL header (TEXPalette)

Size (in bytes)	Meaning
4	Offset to image header.
4	Offset to CLUT header.

Table 13 Texture descriptor (TEXDescriptor)

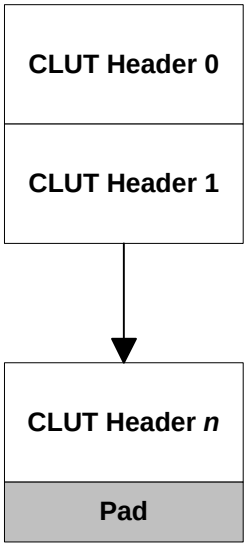


Figure 21 CLUT header bank

The CLUT header bank should pad enough bytes to align to a 32-byte boundary from the top of the TPL file.

Size (in bytes)	Meaning
2	Number of entries.
2	Pad.
4	Pixel format.
4	Offset to CLUT data.

Table 14 CLUT header (CLUTHeader)

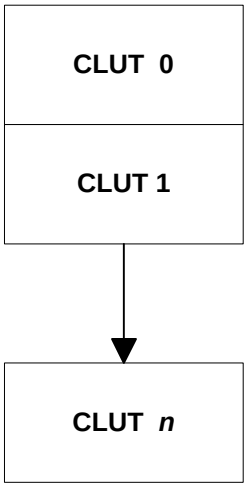


Figure 22 CLUT bank (optional)

The CLUT format is described in “Graphics Library (GX)” in the *NINTENDO GAMECUBE Graphics Programmer’s Guide*.

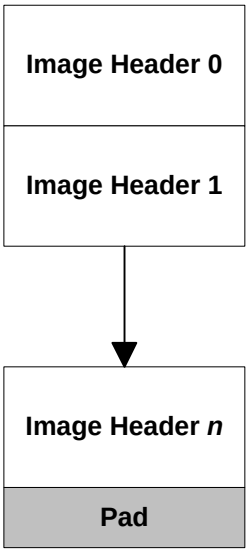


Figure 23 Image header bank

The image header bank should pad enough bytes to align to a 32-byte boundary from the top of the TPL file.

Size (in bytes)	Meaning
2	Height.
2	Width.
4	Color format.
4	Offset to image data.
4	WrapS.
4	WrapT.
4	MinFilter.
4	MagFilter.
4	LODBias.
1	EdgeLODEnable.
1	Image min. LOD.
1	Image max. LOD.
1	Pad.

Table 15 Image header (TEXHeader)

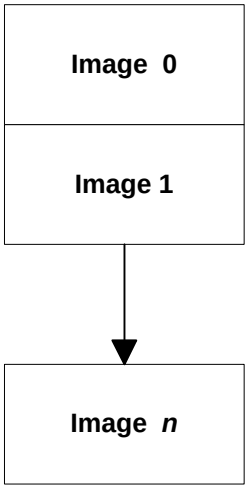


Figure 24 Image bank

The texture image format is described in “Graphics Library (GX)” in the *NINTENDO GAMECUBE Graphics Programmer’s Guide*.

3.3 Actor

This section presents the ACT in its byte-wise file format. In the following figures, the figure caption text within parentheses shows the structure used to access the ACT format in memory. For more details, refer to section 2.3.6.

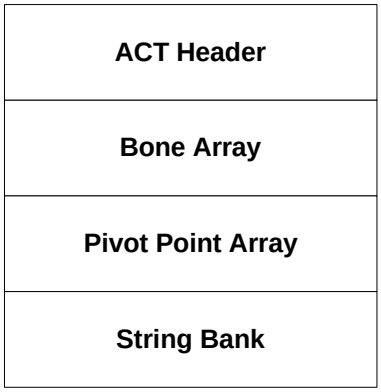


Figure 25 ACT overview

Size (in bytes)	Meaning
4	Version number.
2	Actor ID.
2	Number of bones.
4	Branch offset (`Tree.Offset`).
4	Offset to root bone (`Tree.Root`).
4	Offset to .GPL file name.
2	Skin file ID (0xFFFF = not present).
2	Pad.
4	User-defined data size.
4	Offset to user data (32-byte aligned).

Table 16 Actor header (ACTLayout)

Size (in bytes)	Meaning
4	Offset to orientation control.
4	Offset to previous sibling (`Branch.Prev`).
4	Offset to next sibling (`Branch.Next`).
4	Offset to parent (`Branch.Parent`).
4	Offset to children (`Branch.Children`).
2	Display Object ID (0xFFFF = not present).
2	Bone ID.
1	Inheritance flag.
1	Drawing priority.
2	Pad.

Table 17 Bone (ACTBoneLayout)

3.4 Animation

This section presents the ANM in its byte-wise file format. In the following figures, the figure caption text within parentheses shows the structure used to access the ANM format in memory. For more details, refer to section 2.4.4.

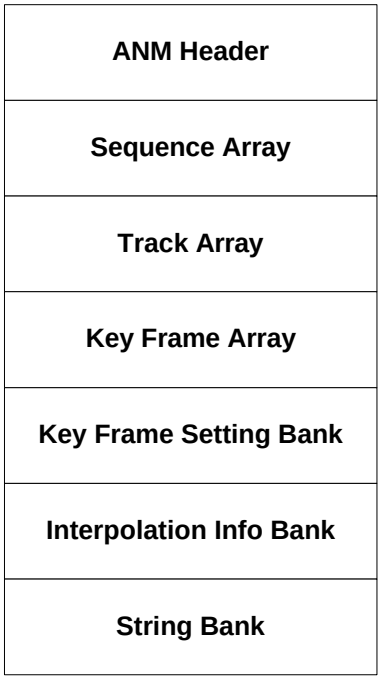


Figure 26 ANM overview

Size (in bytes)	Meaning
4	Version number.
4	Offset to sequence array.
2	Bank ID.
2	Number of sequences.
2	Number of tracks.
2	Number of keyframes.
4	User-defined data size.
4	Offset to user data (32-byte aligned).

Table 18 ANM header (ANIMBank)

Size (in bytes)	Meaning
4	Offset to sequence name.
4	Offset to track array.
2	Number of tracks.
2	Pad.

Table 19 Sequence (ANIMSequence)

Size (in bytes)	Meaning
4	Animation time.
4	Offset to keyframe array.
2	Number of keyframes.
2	Track ID.
1	Parameter quantization info (see Figure 19).
1	Animation type (see Figure 27).
1	Interpolation type (see Figure 28).
1	Replace hierarchy control (always true).

Table 20 Track (ANIMTrack)

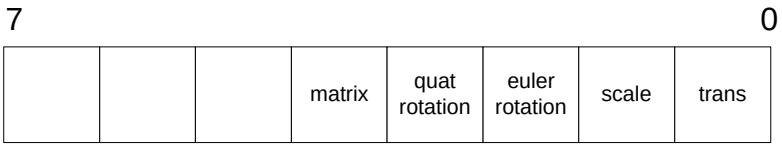
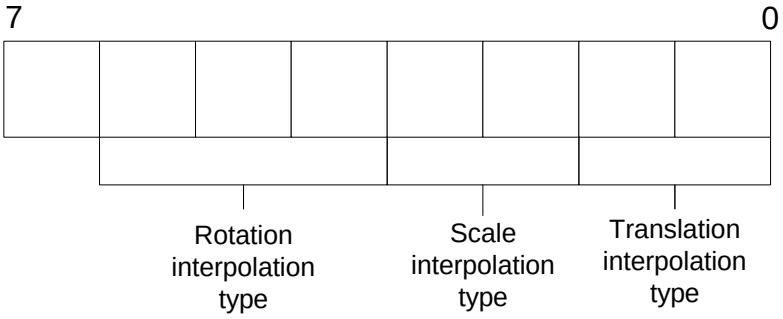


Figure 27 Animation type



None = 00 (Euler, scale, and trans)
Linear = 01 (Euler, scale, and trans)
Bezier = 10 (Euler, scale, and trans)
Hermite = 11 (Euler, scale, and trans)
SQUAD = 100 (quat rotation only)
SQUADEE = 101 (quat rotation only)
SLERP = 110 (quat rotation only)

Figure 28 Interpolation type

Size (in bytes)	Meaning
4	Time.
4	Offset to setting bank.
4	Offset to interpolation info.

Table 21 Keyframe (ANIMKeyFrame)

Size (in bytes)	Meaning
12	In control.
12	Out control.

Table 22 Bezier interpolation (Euler, scale, and translation)

Size (in bytes)	Meaning
12	In control.
12	Out control.
2	Ease-in (2.14).
2	Ease-out (2.14).

Table 23 Hermite interpolation (Euler, scale, and translation)

Size (in bytes)	Meaning
16	In quaternion.
16	Out quaternion.

Table 24 SQUAD interpolation (quaternion only)

Size (in bytes)	Meaning
16	In quaternion.
16	Out quaternion.
2	Ease-in (2.14).
2	Ease-out (2.14).

Table 25 SQUADEE interpolation (quaternion only)

3.5 Skinning

This section presents the SKN in its byte-wise file format. In the following figures, the figure caption text in parentheses indicates the structure used to access the SKN format in memory. For more details, refer to *NINTENDO GAMECUBE Development News* Issue #2 (1/23/2001).

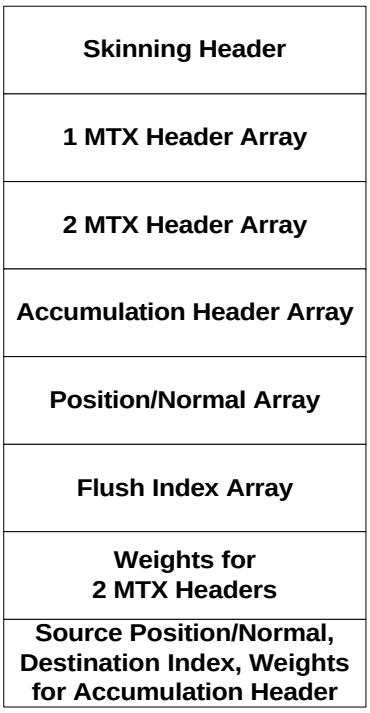


Figure 29 SKN overview

Size (in bytes)	Meaning
2	Number of 1 MTX lists.
2	Number of 2 MTX lists.
2	Number of accumulation lists.
1	Position/normal quantization.
1	Pad.
4	Offset to 1 MTX header array.
4	Offset to 2 MTX header array.
4	Offset to accumulation header array.
4	Offset to top of memory to clear.
4	Byte size of memory to clear.
4	Offset to flush index array.
4	Number of flush indices.

Table 26 SKHeader

Size (in bytes)	Meaning
48	ROMtx (matrix at runtime)
4	Offset to source position/normal array
4	Offset to destination position/normal array (in GPL file)
2	Bone index into ACT file
2	Number of vertices in this list
1	Offset to correct start of position/normal array since it may not be 32-byte aligned.
1	Pad.
2	Pad.

Table 27 SK1List Header

Size (in bytes)	Meaning
48	ROMtx (Matrix 1 at runtime).
48	ROMtx (Matrix 2 at runtime).
4	Offset to source position/normal array.
4	Offset to weight array.
4	Offset to destination position/normal array (in GPL file).
2	Bone 1 index into ACT file.
2	Bone 2 index into ACT file.
2	Number of vertices in this list.
1	Offset to correct start of position/normal array (since it may not be 32-byte aligned).
1	Pad.

Table 28 SK2List Header

Size (in bytes)	Meaning
48	ROMtx (matrix at runtime).
4	Offset to source position/normal array.
4	Offset to destination index array.
4	Offset to destination position/normal array (in GPL file).
4	Offset to weight array.
2	Bone index into ACT file.
2	Number of vertices in this list.

Table 29 SKAccList Header

Appendix A. Building source code

This appendix describes how to build the runtime CP libraries and demos. To learn how to build tools (export plug-ins and texture converter), refer to Appendix A of "CG Tools Programming" in this guide.

A.1 Building runtime libraries

The Character Pipeline runtime libraries are built using a makefile system similar to that of the GCN SDK. For more detailed information, please refer to "Build System" and "SDK Roadmap" in the *NINTENDO GAMECUBE Programmer's Guide* included with the **GCN** SDK.

The GCN SDK must be installed before the CP SDK can be built. Since the GCN SDK ships with pre-built libraries, those CP libraries not included in the CP SDK need not be rebuilt unless you modify them. The CP makefile system will use the `DOLPHIN_ROOT` environment variable to automatically link in the necessary libraries in the GCN SDK. Some CP libraries are still kept in the GCN SDK because of the dependence that GCN demos have on the texPalette library (the TPL format is very useful).

Library	Purpose	Makefile Location
TexPalette (GCN SDK)	Accesses texture objects from TPL file.	`/dolphin/build/charPipeline/texPalette/makefile`
Structures (GCN SDK)	Link list, trees, hashtable data structures.	`/dolphin/build/charPipeline/structures/makefile`
FileCache (GCN SDK)	File access caching.	`/dolphin/build/charPipeline/fileCache/makefile`
GeoPalette	Accesses display objects from GPL file.	`/cp/build/libraries/geoPalette/makefile`
Control	Typical 3D object control/manipulation structures.	`/cp/build/libraries/control/makefile`
Actor	Accesses hierarchy from ACT file.	`/cp/build/libraries/actor/makefile`
Anim	Accesses animation from ANI file.	`/cp/build/libraries/anim/makefile`
Skinning	Provides skinning data for CPU processing.	`/cp/build/libraries/skinning/makefile`

Table 30 Runtime libraries

To build an individual library, go to the directory of the makefile location and in type

```
make –r PLATFORM=...
```

in the Cygnus bash shell. For a non-debug build, type

```
make –r NDEBUG=TRUE PLATFORM=...
```

The `PLATFORM` variable should be one of the following: `HW2`, `HW1`, `MAC`, or `MAC2`. The "`-r`" option in not necessary, but speeds up build time by bypassing built-in makefile rules.

To build all libraries, use the following commands in a Cygnus bash shell:

```
cd /cp/build
make –r libraries NDEBUG=TRUE PLATFORM=...
```

A.2 Building demos

Several demos and utilities are available in the Character Pipeline:

- **previewer:** previews GPL, ACT, ANM files w/ SKN and STP optional files.
- **texPrev2:** previews TPL files.
- **perfView:** performance viewer.
- **cardemo:** demo.
- **sketchdemo:** demo.

These demos are also built using the makefile system. You can build all demos with the following commands in a Cygnus bash shell:

```
cd /cp/build
make –r demos NDEBUG=TRUE PLATFORM=...
```

Like libraries, demos can be built individually. Use the `make` command in any subdirectory of `/cp/build/demos`.

