Nanosaur Game Engine Information

V7/15/98

OVERVIEW

QUICKDRAW 3D

The Nanosaur game engine uses QuickDraw 3D as it's core 3D geometry processing engine, therefore, the bulk of the Nanosaur code deals with generating the terrain and doing skinned animation on the dinosaur models. A complete understanding of QuickDraw 3D is necessary to use this engine and will not be covered in this documentation.

THE CODE

The Nanosaur code is well documented and the functions should be fairly clear. I don't use cryptic variable or function naming conventions, and I generally put functions in C files with logical names. For example, all of the code pertaining to the terrain generation resides in the files Terrain.c & Terrain2.c. Likewise, all of the code dealing with the T-Rex enemy is in the file Enemy_Rex.c.

In the CodeWarrior project, the C files are grouped according to their functionality. The following outlines the meanings of the various groups:

- System
  This group contains all of the C files dealing with basic system operation. Everything from the boot code to random number generation to the master linked list is contained in these files.

- Terrain
  This group contains all of the files responsible for generating the 3D terrain.

- Player
  The 3 files in here deal with the player. All of the code to control the player to managing the weapons is in this group.

- Enemies
  All of the enemies in the game have their own C file and have been grouped here.

- Items
  Everything from powerups to time portals to the lava rocks are in this group.

- Skeleton
  This group contains the most complex C files in the entire engine. These files handle the animation and rendering of all of the dinosaurs in the game.

- QD3D
  Most of the core QuickDraw 3D code is kept in this group of C files. Everything from loading a 3DMF file to setting up a Draw Context is here.

- Screens
  Each of the files in this group handles one of the screens (such as title screen, high scores screen, main menu, and even the infobar) in the game.

This document is not going to explain every single function in every single C file. Most of the functions are self-explanatory either by the name of the function or the comments in the code. Instead, this document will try to explain the fundamental workings of the key components of the Nanosaur engine. Any specific questions can easily be answered by sending an email to brian@pangeasoft.net.

THE TOOLS

To generate the data for the Nanosaur engine, several custom tools must be used:

- BioOreo Pro

This is the keyframe animation tool which is used to create the animation scripts for all of the skinned models. In Nanosaur, the main character plus all of the enemy dinosaurs were animated with this tool.

- OreoTerrain
  This tool is tile-based terrain editing tool which is used to create the Nanosaur landscapes and is also used to place all of the objects (enemies, powerups, etc.) on the terrain.

- 3DMF Optimizer
  All models to be used in the game or by the other tools first need to be optimized using this tool. This tool takes 3DMF geometry files and optimizes them and saves them in a simple 3DMF format that the other tools and the game can work with easily.

- 3DMF Linker
  Nanosaur uses large 3DMF files which contain all of the 3D models used in the game rather than creating a separate file for each object. 3DMF Linker is used to take your original 3DMF models for each object and merge them all into a single large file which the Nanosaur engine can reference into easily.

- OreoSprite
  This tool is not used extensivly by the Nanosaur engine. It is only used to create sprites for the game's status bar (the score, weapon icons, etc.).

- 3DMF Mapper
  This tool is not required by the Nanosaur engine, but it does make certain tasks easier. Most 3D modeling applications have lousy texture mapping features. 3DMF Mapper is a great texture mapping tool for doing organic models such as dinosaurs and other animals.

Instructions for using these tools are not included in this document. Each tool comes with its own set of documentation which explains how to use the tool and the file format(s) which it outputs.

You have also been provided the source code to BioOreo Pro and OreoTerrain since these tools are specific for Nanosaur and you may wish to add features or change the file formats of these tools for your specific application.

USING THE ENGINE

Once you have licensed the Nanosaur engine from Pangea Software, you are free to do whatever you want with it except sub-license it to anyone else or start giving it away to people. You may modify the code as much as you like and may use it in as many applications as you like – all royalty free!

** Remember, however, that only the code and tools have been licensed to you. The artwork to Nanosaur is the property of the artist who drew it and is therefore not part of the licensing agreement. You should not use any artwork or audio from Nanosaur in your own applications since the original owner of the art or sound may sic his lawyers on you.

GAME OBJECTS

OBJECTS == GAME ITEMS

The first thing you need to learn about this game engine is that the entire thing essentially runs off of a gigantic linked list of "objects." Don't worry, this isn't some mystical object like you think of in C++. These objects are more like "real" objects in that each is simply a big data structure which contains all of the data for one item in the game. Each dinosaur is an object, each rock, each bullet, each shadow is an object.

THE OBJECT DATA STRUCTURE

An object is simply a node in a linked list. The code often refers to objects as nodes and vice versa. The structure which defines an object is called **ObjNode** (object node) and its definition as seen in Structs.h is as follows:

```
struct ObjNode
{
	struct ObjNode	*PrevNode;	// ptr toprevious node in linked list
	struct ObjNode	*NextNode;	// ptr to next node in linked list
	struct ObjNode	*ChainNode;
	struct ObjNode	*ChainHead;// a chain's head (link back to 1
						obj in chain)

	short		Slot;		// sort value
	Byte		Genre;		// obj genre
	Byte		Type;		// obj type
```

```
Byte        Group;      // obj group
void        (*MoveCall)(struct ObjNode *); // ptr to obj's move routine
TQ3Point3D Coord;       // coord of object
TQ3Point3D OldCoord;    // coord @ previous frame
TQ3Vector3DDelta;       // delta velocity of object
TQ3Vector3DRot;         // rotation of object
TQ3Vector3DRotDelta;    // rotation delta
TQ3Vector3DScale;       // scale of object
float       Speed;      // speed: sqrt(dx2 * dz2)
float       Accel;      // current acceleration value
TQ3Vector2DTerrainAccel;    // force added by terrain slopes
TQ3Point2D TargetOff; // target offsets
TQ3Point3D AltCoord;        // alternate misc usage coordinate
unsigned long   CType;      // collision type bits
unsigned long   CBits;      // collision attribute bits
Byte            Kind;       // kind
signed char     Mode;       // mode
signed char     Flag[6];
long            Special[6];
float           SpecialF[6];
float           Health;     // health 0..1
float           Damage;     // damage

unsigned long   StatusBits;// various status bits

struct ObjNode  *ShadowNode;     // ptr to node's shadow (if any)
struct ObjNode  *PlatformNode;   // ptr to obj which it on top of.
Struct ObjNode  *CarriedObj;     // ptr to obj being carried/pickedup

Byte            NumCollisionBoxes;
CollisionBoxType*CollisionBoxes;// Ptr to array of collision
                                    rectangles

                                 // box offsets (only used by simple
                                 objects with 1 collision box)
short LeftOff,RightOff,FrontOff,BackOff,TopOff,BottomOff;

TriangleCollisionList *CollisionTriangles; // ptr to triangle
                                             collision data

short       StreamingEffect;     // streaming effect (-1 = none)

TQ3Matrix4x4    BaseTransformMatrix; // matrix which contains all
                                     of the transforms for the
                                     object as a whole
TQ3TransformObject BaseTransformObject;// extra LEGAL object ref to
                                     BaseTransformMatrix (other
                                     legal ref is kept in
                                     BaseGroup)
TQ3Object  BaseGroup;                // group containing all
                                     geometry,etc. for this object
                                     (for drawing)
float      Radius;                   // radius use for object
                                     culling calculation

SkeletonObjDataType   *Skeleton;     // pointer to skeleton
                                     record data
```

```
	TerrainItemEntryType *TerrainItemPtr;	// if item was from terrain,
						   then this pts to entry in
						   array
};
typedef struct ObjNode ObjNode;
```

The ObjNode structure is pretty massive, but it contains everything you'd ever need to know about any object in the entire game. I'm not going to go into the gritty details of every single record in the ObjNode structure since the comments in the code explain most of it, but the fundamental ones will be covered in this document.

The most important records in the ObjNode structure are the following:

- PrevNode
  Points to the previous node in the linked list.

- NextNode
  Points to the next node in the linked list.

- Slot
  Determines the position to insert the node in the linked list. This linked list is ordered from smallest to largest Slot value.

All of the other records are used by specific parts of the game. Only these three basic records are used to manage the objects in the linked list.

OBJECT GENRES

The basic principle behind the Nanosaur engine is to create an ObjNode for each game item as it is needed and then get rid of the node when the item is not needed anymore. Generally, new ObjNode's are created as the player runs along the terrain and a new enemy or item comes into range. As other enemies and items go out of range, their ObjNode's are deleted. So, at any given time, the Object linked list contains all of the currently active items in the game. When the player fires a bullet, a new ObjNode is created for the bullet and when the bullet explodes its ObjNode is deleted.

There are several different "Genres" of ObjNodes for performing different tasks.

- SKELETON_GENRE

This ObjNode will contain information describing an animated character based on a skeleton file created with BioOreo Pro.

- DISPLAY_GROUP_GENRE,
  Simple geometry models are described by the data in this ObjNode.

- EVENT_GENRE
  Even generes don't actually contain any geometry like the other two. These objects simply describe events such as timers or "particle emitters." For example, a special Event object generates the random lava balls that appear in the lava fields. Each lava ball generated by the event object is actually a display group genre object.

CREATING NEW OBJECTS

There are specific functions for creating Skeleton Objects or Display Group Objects, but they all in turn call one core ObjNode initialization function MakeNewObject and take a pointer to an "object definition" data structure as input. The NewObjectDefinitionType contains some default information about the object you want to create. This data is then passed to one of the Object creation routines to actually setup the ObjNode for the new object.

```
typedef struct
{
        Byte            genre,group,type,animNum;
        TQ3Point3D      coord;
        unsigned long   flags;
        short           slot;
        void            (*moveCall)(ObjNode *);
        float           rot,scale;
}NewObjectDefinitionType;
```

Not all fields are used by each genre of Object, but here's what they all mean:

- genre
  Obviously defines the genre of Object we are about to create. Only needs to be set if calling MakeNewObject directly.

- group
  For Display Group objects this refers to the 3DMF group containing the model we want to use.

- type
  Refers to the specific model inside the 3DMF group which we want to use for this object.

- animNum
  Used by Skeleton objects to determine which animation sequence the object should be using upon creation.

- coord
  Defines the 3D coordinate of the object in the game.

- flags
  Contains bits which define special features of the object.

- slot
  Where to put this new Object in the linked list

- moveCall
  This is a pointer to the Object’s “move” routine. This function will automatically be called by the MoveObjects function each frame of the game.

- rot
  The initial y-axis rotation of a Skeleton or Display Group Object.

- scale
  The initial scale of a Skeleton or Display Group Object.

As mentioned above, there are different functions for creating the different genres of objects:

- MakeNewDisplayGroupObject
  This function is used to create a new Display Group Object. For example, the following code is used to create a new “spore pod” in Nanosaur:

```
NewObjectDefinitionType     gNewObjectDefinition;
ObjNode                     *newObj;

gNewObjectDefinition.group = LEVEL0_MGroupNum_Pod;
gNewObjectDefinition.type  = LEVEL0_MObjType_Pod;
gNewObjectDefinition.coord.x = x;
gNewObjectDefinition.coord.y = GetTerrainHeightAtCoord_Planar(x,z);
gNewObjectDefinition.coord.z = z;
gNewObjectDefinition.flags = 0;
gNewObjectDefinition.slot  = 100;
gNewObjectDefinition.moveCall = MoveSporePod;
gNewObjectDefinition.rot   = 0;
gNewObjectDefinition.scale = .5;
newObj = MakeNewDisplayGroupObject(&gNewObjectDefinition);
```

As you can see, the required fields of the NewObjectDefinitionType structure are filled out and then passed to MakeNewDisplayGroupObject. MakeNewDisplayGroupObject automatically sets the genre field and takes care of doing everything else needed to initialize the Object.

- MakeNewSkeletonObject
  This function is used to create a new Skeleton Object. The code to create a new skeleton object looks very similar to the above code with a few exceptions. Here's the code to put the Nanosaur on the game's Main Menu:

```
gNewObjectDefinition.type  = SKELETON_TYPE_DEINON;
gNewObjectDefinition.animNum= 1;
gNewObjectDefinition.scale = .8;
gNewObjectDefinition.coord.x= -350;
gNewObjectDefinition.coord.y= 00;
gNewObjectDefinition.coord.z= 630;
gNewObjectDefinition.slot  = 10;
gNewObjectDefinition.flags = STATUS_BIT_HIGHFILTER|STATUS_BIT_DONTCULL;
gNewObjectDefinition.moveCall = nil;
gNewObjectDefinition.rot   = 0
newObj = MakeNewSkeletonObject(&gNewObjectDefinition);
```

This code tells the game to create the new Skeleton Object and start it playing animation #1. It also sets a few of the status bits in the flags field. These status bits tell the game engine to use the best texture filtering and not to bother cull-testing the model when rendering it.

This is all that is needed to put an animating character on the screen. Once the ObjNode has been created and put into the linked list like this, everything else is automatically handed by the game engine and you don't need to worry about a thing.

PROCESSING OBJECTS

Once you have created a new ObjNode as in above, all processing is handled pretty much automatically. In your game loop you only need to call two functions: MoveObjects and DrawObjects. When these functions are called, the ObjNode linked list is traversed and each object is processed.

The MoveObjects function will automatically call the "MoveCall" for each Object. Remember the MoveCall was set in the definition structure you filled out when you created the Object. Skeleton Objects also have their animation automatically updated here. You don't have to do a thing – it's all automatic!

Your move function can do whatever it likes to the input ObjNode. The one thing to know is that you must call UpdateObjectTransforms at the end of the function to update the Object's transform matrices if you have moved, scaled, or rotated the object. To move the Object you just fiddle with the Coord field of the ObjNode (ie. myObj->Coord.x = 100). Or, to rotate it you would do something like myObj->Rot.z += 1.2. These actions alone don't actually do anything. You must call UpdateObjectTransforms to update the transform matrices to these new orientation values.

When you want to remove an Object from the linked list (say the enemy gets blown up or walks too far away) all you need to do is call DeleteObject which takes care of disposing of all memory allocated by the ObjNode and removes the ObjNode from the linked list. It is very important that you never try to access fields of an ObjNode which you have just Deleted since the memory may now be in use by something else and modifying it could cause a crash.

SKELETON OBJECTS

Later in this document I will give more details how the actual Skeleton animation code works, but for now there are some things to know about Skeleton Objects in general.

Skeleton Objects automatically take care of updating the skeleton's animation from frame to frame. You use the following functions to change the Skeleton's animation:

- SetSkeletonAnim
  Use this function to set the animation number of a Skeleton Object.

- MorphToSkeletonAnim

Causes the animation to morph from the current frame to the 1st frame of the indicated animation. You can set the speed of the morph.

Also, keep in mind that Skeleton Objects do use up a lot of memory in addition to just the ObjNode structure. Notice that many of the fields in the ObjNode structure are there for Skeleton Objects. When a new Skeleton Object is created, a bunch of other memory is allocated to the ObjNode for these fields.

NOTES ABOUT OBJECTS

There are many additional functions you can use to manage ObjNodes, and most of them are found in Objects.c and Objects2.c. Most of these functions are self explanatory, therefore, their functionality is not covered here.

QD3D SUPPORT FUNCTIONS

VIEW CREATION

Normally, creating a new View object in QuickDraw 3D is a very tedious and annoying process to endure. The QD3D_Support.c file contains a bunch of functions I have written to assist with working with QD3D.

To create a new Draw Context / View Object with the Nanosaur engine, you can do something like the following:

```
QD3DSetupInputType      viewDef;

 QD3D_NewViewDef(&viewDef, gCoverWindow);

 viewDef.camera.hither       = 10;
 viewDef.camera.yon          = 500;
 viewDef.camera.fov          = 1.0;
 viewDef.lights.fogHither    = .3;

 QD3D_SetupWindow(&viewDef, &gGameViewInfoPtr);
```

The function QD3D_NewViewDef is called to initialize a QD3DSetupInputType structure to its default values. The QD3DSetupInputType structure is a large data structure which contains all of the settings which you wish to base your new View from. In the code above, we are modifying some of the default values that were set in QD3D_NewViewDef. We are changing some of the camera's values (hither, yon, fov) and one of the light values (fogHither). You can look in QD3DSupport.h for the description of the full data structure. You are free to modify any of the default values after QD3D_NewViewDef has been called.

Once you have filled out your QD3DSetupInputType structure, you simply pass it to QD3D_SetupWindow. QD3D_SetupWindow automatically creates a new View object based on the information in the input data structure. No work on your part is required – it is all done automatically!

QD3D_SetupWindow does just return you a reference to a View object. Instead it fills out another data structure: QD3DsetupOutputType. This new structure looks like this:

```
typedef struct
{
 Boolean                      isActive;
	TQ3ViewObject			viewObject;
	TQ3ShaderObject			shaderObject;
	TQ3ShaderObject			nullShaderObject;
	TQ3StyleObject			interpolationStyle;
	TQ3StyleObject			backfacingStyle;
	TQ3StyleObject			fillStyle;
	TQ3CameraObject			cameraObject;
	TQ3GroupObject			lightGroup;
	TQ3DrawContextObject		drawContext;
	WindowPtr			window;
	Rect				paneClip;
	TQ3Point3D			currentCameraCoords;
	TQ3Point3D			currentCameraLookAt;
	float				hither,yon;
}QD3DSetupOutputType;
```

This structure contains everything you'll ever need to know about the new View that was created. Here you have access to the camera, the lights, the draw Context, etc. Since this structure contains everything important about the View, this is the structure that you will usually be passing to most of the rendering functions. For example, you pass this data to DrawObjects when you are about to draw the objects in the linked list.

UTILITY FUNCTIONS

In QD3DSupport.c you will see that there are a lot of other utility functions which do everything from creating new lights, to updating the camera position, to creating texture shaders. All of these functions are fairly self-explanatory but if something does not make sense, find out where it is being called in the Nanosaur code to get a better idea how it should be used.

GEOMETRY FUNCTIONS

There is also a file called QD3DGeometry.c which contains a bunch of useful geometry related functions. Here there are functions for calculating bounding boxes, exploding geometries into small triangles, and animating the UV coordinates on models. All of these functions are used in Nanosaur, so once again, look thru the code to learn how they are used.

THE TERRAIN ENGINE

FROM 2D TO 3D

The terrain in Nanosaur is actually a 2D tiled map like you would find in any old scrolling game from years past. The thing that makes it 3D is that there is a “heightmap” layer which is used to provide height information for each corner of each tile in the map. So, we are essentially taking a 2D map and extruding it up to make it 3D.

The OreoTerrain documentation explains how to create the terrain files and use the editor. This documentation will explain the general functionality of the terrain engine inside Nanosaur. It will explain how the OreoTerrain data files are actually processed by this engine.

TILES AND SUPERTILES

The tiles that make up a terrain map in Nanosaur are 32x32 pixels in size. For speed and efficiency, however, the Nanosaur engine also uses what are called “SuperTiles.” A SuperTile is nothing more than a 5x5 matrix of tiles. Each SuperTile is a single piece of geometry which is built on-the-fly as the player moves along the terrain. The texture for the SuperTile is also built on-the-fly by simply drawing the textures for the 25 tiles into a buffer and then assigning that to the SuperTile’s geometry.

Since each tile is 32 pixels wide and each SuperTile is 5 tiles. That would make each SuperTile’s texture 160 pixels in dimension. Since 3D accelerators only work with textures which are powers of 2, the Nanosaur terrain engine actually shrinks the 160x160 SuperTile texture down to 128x128 pixels. The 2MB version of the game actually shrinks the textures down to 64x64.

As the player moves along the terrain, new SuperTiles are created and old ones are removed. The Nanosaur engine knows which SuperTiles are not visible by the camera and culls them from the scene before rendering.

PARAMETERS

The terrain.h header file contains various values for adjusting the terrain parameters. For example, TERRAIN_POLYGON_SIZE determines the size of a single tile in world units. Even though a tile may be 32 pixels in size, you can make the geometry be any size you want. Nanosaur defaults to 140 units, but you can set this to whatever works for your application.

The SUPERTILE_SIZE value determines the actual size of a SuperTile in terms of tiles. Keep in mind that changing this value or the SUPERTILE_TEXMAP_SIZE value will mean that you need to rewrite the texture shrinking function since it's custom hard-coded for these values. In general, you should not need to change most of the values in this header file.

The SUPERTILE_ACTIVE_RANGE value is of particular interest. This value is normally set to 3. It indicates how many SuperTiles to the left, right, front, and back of the camera to make the visible scene. If you increase your camera's yon value, you will need to increast the SUPERTILE_ACTIVE_RANGE value so that more SuperTiles will be visible off in the distance. With the value of 3, there are 6x6 (or 36) active SuperTiles at any given time. Note that the more SuperTiles that are active, the more VRAM you will need to hold their 36 textures.

Another very important value is HEIGHT_EXTRUDE_FACTOR. This determines how much to extrude the vertices of the terrain. Each vertex has a height value of 0 to 255 which it gets from the OreoTerrain data file. This height value is then multiplied by the HEIGHT_EXTRUDE_FACTOR to get the actual y-coordinate of the vertex. Therefore, the larger a number you assign to HEIGHT_EXTRUDE_FACTOR, the taller and steeper the terrain will appear.

BASIC TERRAIN FUNCTION CALLS

At boot, you need to call the function InitTerrainManager to initialize the terrain manager. This will allocate memory for the terrain SuperTiles and do other buffer allocation.

Next, you will need to load the terrain data files. The code to do this should look something like this:

```
FSMakeFSSpec(gDataSpec.vRefNum, gDataSpec.parID, "\p:terrain:level1.trt", &spec)
LoadTerrainTileset(&spec);
FSMakeFSSpec(gDataSpec.vRefNum, gDataSpec.parID, "\p:terrain:level1.ter", &spec)
LoadTerrain(&spec);
```

Note that the LoadTerrain function also scans for the player's starting coordinate which is one of the map "items" in the data.

Once the data files are loaded you need to “prime” the terrain by calling PrimeInitialTerrain. This will initialize all of the SuperTiles which need to be visible when the first frame of animation is rendered. At this point calling DrawTerrain is all you need to do to draw the terrain.

When the player moves, the function DoMyTerrainUpdate needs to be called. This will tell the terrain manager to scroll the terrain, create new SuperTiles, and delete old SuperTiles. It also takes care of initializing any new items or enemies which may have scrolled into range as well.

The best way to understand how these functions work is to look at the Nanosaur code which calls them. It’s probably not important that you understand exactly how each of the functions works. The code is commented well, but the terrain engine is fairly complex and unless you have a really good reason for doing so, it is not advisable to mess with it.

GETTING THE TERRAIN HEIGHT

Perhaps the most often called terrain manager function is GetTerrainHeightAtCoord_Planar. This call returns a y-coordinate based on the input x and z coodinates. Given any x and z coordinate in the world, this function will return the y coordinate (the height) of the terrain at that point.

This function is extremely accurate when the point is on top of an active SuperTile. If the point is outside of range where no SuperTile is currently active, then the function returns a “rough” result based on the OreoTerrain data file’s height map information. The function GetTerrainHeightAtCoord_Quick is used to get this value (GetTerrainHeightAtCoord_Planar automatically calls this if the point is out of range).

You will notice that every moving object in the game from the player to the bullets to the enemies calls GetTerrainHeightAtCoord_Planar after each move to update it’s y-coordinate or to see if it has hit the “floor.”

TERRAIN ITEMS

As mentioned above, new rows and columns of SuperTiles are created as the player moves along the ground. In addition to new SuperTiles, any terrain items which exist on those SuperTiles are also initialized at this time. However, as SuperTiles are deleted off of the trailing edge of the scroll “window”, items are not deleted. Items are generally deleted when they are significantly out of range. The function TrackTerrainItem() is used to determine if an ObjNode is out of the default range. If so, then the object’s move handling function will delete it.

The OreoTerrain documentation explains how terrain items are identified by item numbers with other sub-parameters. As the Nanosaur Terrain manager scans for new items to initialize as you scroll along the terrain, it uses a big jump table to handle the items. In Terrain2.c, there is a long jump table called gTerrainItemAddRoutines[]. When the Terrain manager encounters an item, it looks up the item’s initialization or “Add” function in this table and then calls it. The Add function then takes care of making the new ObjNode for that terrain item. You will notice that the Nanosaur code is very consistent in that all terrain item initialization routines start with “Add…” (i.e. AddGasVent, AddBush, AddEnemy_Stego, etc.). Most of these functions look very similar: they create a new ObjNode and set custom information about the object.

COLLISION DETECTION

This section explains how the collision detection in the game engine works. There are basically 3 different types of collision:

- Object bounding box collision
- Object triangle collision
- Tile collision

The file Collision.c contains all of the core collision detection routines, but much of the custom handling is done in MyGuy.c and Enemy.c,

OBJECT COLLISIONS

Object collision is the collision between two ObjNode's. There are several fields in the ObjNode structure which are important to the collision detecting process:

- **CType**
  "Collision Type" for use in classifying an object type.
- **CBits**
  "Collision Bits" specifies collision options.
- **NumCollisionBoxes**
  the number of collision boxes assigned to this ObjNode.
- **CollisionBoxes**
  an array of CollisionBoxType's which contains the world-space collision box coordiantes.
- **CollisionTriangles**
  pointer to collison triangle information for doing triangle-accurate collision detection.

CTYPE & CBITS

The CType field determines the collision classification of an object. When a collision function is called a CType value is passed in which indicates what objects to perform collision on. An object is only tested if one or more of it's CType bits match the input CType's bits. In other words, if an object is an enemy, then it will have the CTYPE_ENEMY bit set. If you want to see if a bullet has hit an enemy then you pass the CTYPE_ENEMY bit to the collision function. If you also want to check if the bullet has hit the player, you also pass the CTYPE_PLAYER (**CTYPE_ENEMY | CTYPE_PLAYER**)

All of the CType values are defined in Structs.h:

```
enum
{
	CTYPE_PLAYER     =     1,          // Me
	CTYPE_ENEMY      =     (1<<1),     // Enemy
	CTYPE_MYBULLET   =     (1<<2),     // Player's bullet
	CTYPE_BONUS      =     (1<<3),     // Bonus item
	CTYPE_TRIGGER    =     (1<<5),     // Trigger
	CTYPE_SKELETON   =     (1<<6),     // Skeleton
	CTYPE_MISC       =     (1<<7),     // Misc
	CTYPE_BLOCKSHADOW =    (1<<8),     // Shadows go over it
	CTYPE_HURTIFTOUCH=     (1<<9),     // Hurt if touched
	CTYPE_PORTAL     =     (1<<10),    // time portal
	CTYPE_BGROUND2   =     (1<<11),    // Terrain BGround 2 path tiles
	CTYPE_PICKUP     =     (1<<12),    // Pickup
	CTYPE_CRYSTAL    =     (1<<13),    // Crystal
	CTYPE_HURTME     =     (1<<14),    // Hurt Me
	CTYPE_HURTENEMY  =     (1<<15),    // Hurt Enemy
	CTYPE_BGROUND    =     (1<<16),    // Terrain BGround path tiles
	CTYPE_PLAYERTRIGGERONLY = (1<<17)  // combined with _TRIGGER, this..
	                                   // ..trigger is only triggerable by player
};
```

You will notice that when most ObjNode's are created in an item's Add function, the CType and CBits are set to tell the collision system what kind of object it is and how collision should be performed on it. For example, when a Time Portal terrain item is initialized, the following code sets this collision type and bits:

```
	newObj->CType = CTYPE_PORTAL;
	newObj->CBits = CBITS_TOUCHABLE;
```

COLLISION BOX

Every ObjNode which you plan on colliding with needs a collision box. The collision box is a bounding box which closely approximates the bounds of an object. If an object is moving, this bounding box will need to be updated on every frame. Nanosaur allows for an ObjNode to have multiple collision boxes, but in most cases only 1 box is needed (as is such in the Nanosaur code).

To easily set an ObjNode’s collision box, simply call the following function:

```
	SetObjectCollisionBounds(theObjNode, top, bottom, left, right, front, back)
```

The top, bottom, left, right, front, and back values are the offsets from the object’s origin for its bounding box. The Time Portal’s collision box is defined as:

```
	SetObjectCollisionBounds(newObj,500,0,-60,60,60,-60);
```

Once you have set an ObjNode’s Ctype, CBits, and collision box, you’ve done all you need to do to setup its collision information for doing simple bounding box collision detection. Just remember that if the object moves, you need to be sure that the bounding box gets updated. To do this you should call CalcObjectBoxFromNode() which will automatically recalculate the bounding box’s coordinates based on the ObjNode’s new coordinate and the old top, bottom, etc., offsets that you fed it at init time.

Note, that in most cases you will call a function called UpdateObject at the end of its “move” function. UpdateObject will automatically call CalcObjectBoxFromNode for you so you don’t really need to think about it.

COLLISION TRIANGLES

Occasionally, you may want to be able to collide more accurately with an ObjNode, and when this happens you need to calculate Collision Triangles for the object. Objects with this kind of collision still have bounding boxes, but you do not need to specify them explicity like we did before. Instead, the function CreateCollisionTrianglesForObject() takes care of everything for us. It scans the object and creates a list of triangles for later collision detection and also builds the collison box for internal use.

Collision Triangles can only be assigned to non-moving objects! Do not try to do this on something that moves because it just won’t work.

In Nanosaur, the gray boulders are an example of an object with collision triangles. The boulder’s collision information is setup with the following code:

```
	newObj->CType = CTYPE_MISC;
```

```
newObj->CBits = CBITS_ALLSOLID;

CreateCollisionTrianglesForObject(newObj);
```

That's all that needs to be done to make the collision detection on this object work on a polygon-accurate level. You can run around and jump on boulders in a very accurate collision scheme.

TILE COLLISIONS

Tile collisions are somewhat complex when explained in detail, so I will try to explain just the important points here since it is unlikely that you will ever need to modify this part of the Nanosaur engine.

In the Nanosaur engine, the terrain maps have 3 layers: the texture layer, height layer, and the path layer. The path layer contains collision tiles which are used to determine where solid areas of the map are. See the OreoTerrain documentation for more information.

The collision manager will automatically take care of detecting collisions with tiles, but suffice to say that the function GetTileCollisionBitsAtRowCol() is called to get the collision information about the tiles which the ObjNode in question is colliding against.

PERFORMING COLLISIONS

Most generic Objects can call a single collision function to automatically handle all collision detection: HandleCollisions(). This function takes an ObjNode and a CType as input values, but also relies on some global values for doing collision detection:

- gCoord
  Contains the current global coordinate of the ObjNode. This function does not use the Coord field in the ObjNode structure!

- gDelta
  Contains the motion delta values that the ObjNode just now moved. This function does not use the Delta field in the ObjNode structure!

The HandleCollisions function takes care of performing collisions against all ObjNodes with matching Ctypes, and the collision information is stored in the array gCollisionList[]. You can use gNumCollisions and gCollisionList to manually find out information about what kinds of collisions just occurred, but it is more likely that you will usually let the collision manager handle these events on its own.

In the case that your object is not generic and you need to handle collisions in a specific way, you will want to call the function CollisionDetect. All this functions does is build the gCollisionList, but unlike HandleCollisions, nothing is done about these collisions. Both the player and enemy manager code have specialized collision handling logic, so they only call CollisionDetect and then parse through the data in gCollisionList manually. The player's collision handling function is called DoPlayerCollisionDetect, and the enemy's collision function is DoEnemyCollisionDetect.

It is best to examine the existing Nanosaur code to learn more about how these functions are called and used. The important thing to remember is that most of the time, all collisions are handled automatically and it is not important that you understand everything about how the code works.

TRIGGERS

Triggers are a specific type of object which does something when the player touches it. PowerUps, Exploding Crystals, and Lava Stepping Stones are examples of Triggers – they all respond to being touched by the player.

The player's collision detection handler calls the Trigger collision handler function called HandleTrigger(). This function determines if the collision requires action by the Trigger Object and if so, it calls the Trigger's "DoTrig_" function.

SKELETONS

The Skeleton Manager is really the core part about the Nanosaur engine which makes it unique. This is the section of code which handles all of the character animation for the dinosaurs in Nanosaur. The skeleton data is created with BioOreo Pro, and the documentation for that tool explains the workings of the Skeleton in more detail than is covered here. In this document, I will cover the specifics of the function calls you need to use to work with Skeleton models in your application.

SKELETON FILES

The first thing you need to do in order to use Skeletons in your application is to load the .skeleton file(s) that you are going to use. A single Skeleton can be used to simultaneously animate an infinite number of Skeleton objects on the screen. So, to load the T-Rex Skeleton, we do the following:

```
LoadASkeleton(SKELETON_TYPE_REX);
```

This function takes a "SKELETON_TYPE" as input. These values are enumerated in SkeletonObj.h:

```
enum
{
 SKELETON_TYPE_PTERA,
 SKELETON_TYPE_REX,
 SKELETON_TYPE_STEGO,
 SKELETON_TYPE_DEINON,
 SKELETON_TYPE_TRICER,
 SKELETON_TYPE_SPITTER
};
```

You can easily add or remove Skeleton Types from this list, just be sure to always update the MAX_SKELETON_TYPES value to indicate the correct quantity of Types you have.

LoadASkeleton() calls a function named LoadSkeletonFile(). This function has a big switch statement which handles the loading of the specific skeleton file indicated by the Type you have passed in. LoadSkeletonFile() then takes care of actually loading in the data for the Skeleton and also loads the Skeleton's 3DMF model file. Basically, calling LoadASkeleton will do all memory allocations and geometry parsing that needs to be done. You really don't have to worry about much – it's all automatic.

SKELETON 3DMF DECOMPOSITION

When a Skeleton’s 3DMF file is loaded via LoadASkeleton(), the geometry in the file is “decomposed” into data which the Nanosaur engine can use to do the animation. Bascially, this decomposition simply creates lists of vertices and normals for the model while removing all duplicates. When the engine animates a skeleton, it uses this decomposed data to construct the actual TriMesh which is eventually displayed on the screen.

Remember that the Skeleton animation in Nanosaur is done by deforming the original 3DMF geometry of the dinosaur. We keep an original copy of the geometry in our decomposed lists, deform the geometry and store the result into a temporary TriMesh, and then submit the TriMesh in immediate mode to render it. Sounds relatively straight-forward, but believe me, it is a very complex process and luckily for you, you don’t need to worry about how it works.

During decomposition, the following functions are used to determine if two points or vectors are identical or “close enough”:

```
PointsAreCloseEnough()
VectorsAreCloseEnough()
```

Notice how one of these functions appears:

```
Boolean VectorsAreCloseEnough(TQ3Vector3D *v1, TQ3Vector3D *v2)
{
 if (fabs(v1->x - v2->x) < 0.02f)
      if (fabs(v1->y - v2->y) < 0.02f)
           if (fabs(v1->z - v2->z) < 0.02f)
                return(true);

 return(false);
}
```

There is an identical function in BioOreo Pro which does the same thing. The “tolerance” values used in both the Nanosaur engine and BioOreo Pro must match exactly or bad things will happen.

In the above function, the tolerance values are set to 0.02. In the PointsAreCloseEnough function, the tolerance is 0.001. The tolerance values in BioOreo Pro are identical. If you change them in one app, you must change them in the other.

SKELETON OBJECTS

The third genre of Objects in the Nanosaur engine is the SKELETON_GENRE. Just like the other genre (DISPLAY_GROUP_GENRE and EVENT_GENRE), there is a special function call used to create a new Skeleton Object:

```
MakeNewSkeletonObject()
```

Just as with the other genres, this function takes a NewObjectDefinitionType structure as input. You fill out the usual data plus a few additional fields. The following code is used to create the player's Skeleton Object in Nanosaur:

```
gNewObjectDefinition.type  = SKELETON_TYPE_DEINON;
gNewObjectDefinition.animNum = PLAYER_ANIM_FALLING;
gNewObjectDefinition.coord.x = gMyStartX;
gNewObjectDefinition.coord.y = y + 400;
gNewObjectDefinition.coord.z = gMyStartZ;
gNewObjectDefinition.slot = PLAYER_SLOT;
gNewObjectDefinition.flags = STATUS_BIT_DONTCULL|STATUS_BIT_HIDDEN;
gNewObjectDefinition.moveCall = MoveMe;
gNewObjectDefinition.rot = (float)gMyStartAim * (PI2/8);
gNewObjectDefinition.scale = MY_SCALE;
gPlayerObj = MakeNewSkeletonObject(&gNewObjectDefinition);
```

Most of this should look pretty familiar from earlier in this document. The main difference is the use of the animNum field. A .skeleton file contains not only the skeleton geometry information, but also the animation information for all of the skeleton's animation sequences. When you create a new Skeleton Object you specify which animation to start with by filling in the animNum field.

Everything else behaves just as it does for a DISPLAY_GROUP_GENRE Object, but as we'll see later, the Object's Move function needs to do a little more work to keep the Skeleton Object updated.

In the ObjNode structure, there is a field called Skeleton. When you create a Skeleton Object, this field will point to a large block of data which contains all of the additional information needed to describe a Skeleton Object:

```
typedef struct
{
 Byte            AnimNum;         // animation #
```

```
    Boolean          IsMorphing;       // flag set when morphing
    float            MorphSpeed;       // speed of morphing (1.0 = normal)
    float            MorphPercent;     // % morph from kf1 to kf2 (0.0 - 1.0)

    JointKeyframeType JointCurrentPosition[MAX_JOINTS]; // for each joint,
                          holds current interpolated keyframe values
    JointKeyframeType MorphStart[MAX_JOINTS]; // morph start & end keyframes for
                                        each joint
    JointKeyframeType MorphEnd[MAX_JOINTS];

    float            CurrentAnimTime;// current time index for animation
    float            LoopBackTime;     // time to loop or zigzag back to
    float            MaxAnimTime;      // duration of current anim
    float            AnimSpeed;        // time factor for speed of executing
                                          current anim (1.0 = normal time)
    Byte             AnimEventIndex;   // current index into anim event list
    Byte             AnimDirection;    // forward or backward
    Byte             EndMode;          // what to do when reach end of anim
    Boolean          AnimHasStopped;   // flag gets set when anim has reached
                                       end of sequence (looping anims don't set
                                       this!)

    TQ3Matrix4x4     jointTransformMatrix[MAX_JOINTS]; // holds matrix xform
                                                for each joint

    SkeletonDefType  *skeletonDefinition; // point to skeleton's shared data

    TQ3TriMeshData   localTriMeshes[MAX_DECOMPOSED_TRIMESHES]; // the triMeshes
                                        to submit for this ObjNode
   }SkeletonObjDataType
```

This structure holds information ranging from the status of the current animation to the TriMeshes used to render the model. The structure is large and complicated, but the commenting in the code is fairly clear about how everything operates. For the most part, you should never have to worry about anything in this structure that is not documented below.

SKELETON ANIMATION

A looping animation will continue to play and will automatically be updated each time through the game's main loop. The MoveObjects() function calls UpdateSkeletonAnimation() which does the deformation on the skeleton's geometry for the current frame. It handles all of the interpolation of the keyframes to get the exact state of each vertex in the mesh. To change from one animation to another or to morph to a different animation, there are two functions which can be called inside the Skeleton's Move function:

```
   SetSkeletonAnim()
   MorphToSkeletonAnim()
```

Use SetSkeletonAnim() to immediately switch to a different animation sequence. The following code shows how the player starts the death sequence in Nanosaur:

```
    SetSkeletonAnim(theNode->Skeleton, PLAYER_ANIM_DEATH);
```

This will cause the animation to start at time = 0 of the given animation sequence. This function also resets the animation playback speed to 1.0 (normal) by setting the AnimSpeed field in the ObjNode's SkeletonObjDataType structure.

By default, animations and morphs always play at "normal" speed. Normal speed simply means the speed that the animation appears in BioOreo Pro. You can speed up and slow down any animation simply by changing the AnimSpeed field. A value of 2.0 would make the animation run 2x faster, and a value of 0.5 would make it run half as fast. In Nanosaur, I tweak the animation playback speed for the running animation depending on how fast the player is moving. The faster the player moves, the faster I play the animation so he doesn't appear to moon-walk.

A morph causes the current frame of animation to morph to the first frame of animation in the target animation before continuing to play the target animation. It's really the same as a "SetAnim" except that it doesn't immediately start the next animation. Instead, it morphs to the animation before playing it.

In Nanosaur, when the player stops running, he morphs back to the stand position. This makes the transition much smoother than a SetAnim:

```
    MorphToSkeletonAnim(theNode->Skeleton,PLAYER_ANIM_STAND,3.0);
```

This looks just like SetSkeletonAnim() except that there is a third parameter which is the morph speed parameter. The bigger the number, the faster the morph will be. There's no real meaning to this number, just use whatever gives you the desired visual effect. In general, I use numbers between 2 and 6 for most of the morphing in Nanosaur.

JOINT COORDINATES & ORIENTATION

Very often you may need to know the world-space coordinate of a Skeleton's head, foot, or whatever. For example, in Nanosaur I need to know the coordinate and orientation of the player's gun so I know where to start the bullets and which direction to fire them in.

Getting this information is very easy because there are a few utility functions in the Nanosaur engine for extracting the coordinates and matrix associated with any of the Skeleton's joints:

```
FindCoordOfJoint()
FindCoordOnJoint()
FindJointFullMatrix()
```

FindCoordOfJoint()

This function will return the current world-space coordinates of the joint in question.

Nanosaur calls this function for a kick animation that never made it into the final game. I needed to know the coordinates of the player's foot so that I could see if he had successfully kicked an enemy:

```
FindCoordOfJoint(theNode, MY_FOOT_JOINT_NUM, &footCoord);
```

You need to know the joint # of the limb you are finding. This is done by looking at the Animation Timeline window in BioOreo Pro. The footCoord value will contain the world-space coordinates of this joint upon return.

FindCoordOnJoint()

This function is similar to FindCoordOfJoint() except that it returns the coordinates of a point relative to the joint. In other words, if I'm shooing a bullet out of the tip of my gun, I need the coordinates of the tip of the gun, not the origin of the gun. FindCoordOfJoint() only returns me the coordinates of the origin of a joint, but FindCoordOnJoint() gives me the coordinates of any point relative to the origin.

So, the code for finding the coordinate of the tip of the player's gun looks like this:

```
static  TQ3Point3D gGunTipOffset = {0,0,-65};
```

```
FindCoordOnJoint(theNode, MYGUY_LIMB_GUN, &gGunTipOffset, &gunTipCoord);
```

I know by looking at my model in Form*Z that the tip of the gun is –65 units down the z-axis from the origin. Therefore, I set the gGunTipOffset to (0,0,-65). When I pass this to FindCoordOnJoint(), it returns the world-space coordinates of this offset.

FindJointFullMatrix ()

The other two functions return coordinates, but not any orientation information. I still need to know what direction my gun's tip is facing so I can know which direction to shoot my bullets. Actually, to be fair, I don't actually use this function to get the bullet's trajectory in Nanosaur. Since the gun doesn't really move much relative to the player, I cheat and just assume its pointing straight and I use the player's y-axis rotation information.

But… if I were to do this correctly, I would do the following:

```
FindJointFullMatrix(playerObj, MYGUY_LIMB_GUN, &jointMat);
```

This would return to me the full 4x4 transformation matrix for that joint.

CONCLUSION

So, hopefully, you have at least a rough understanding of how the Nanosaur engine works. Most of the code is fairly easy to understand and is well commented. My advice for working with this code is to start with the Nanosaur engine as a shell and gradually take things out and add your own things in. Going slow will avoid any catastrophic events.

Should have run into problems or have questions that are not answered in this document, please don’t hesitate to contact me (pangea@bga.com).