RenderWare Graphics

User Guide

Volume II

Contact Us

Criterion Software Ltd.

For general information about RenderWare Graphics e-mail info@csl.com.

Developer Relations

For information regarding Support please email devrels@csl.com.

Sales

For sales information contact: rw-sales@csl.com

Acknowledgements

Contributors	RenderWare Graphics Development and Documentation Teams

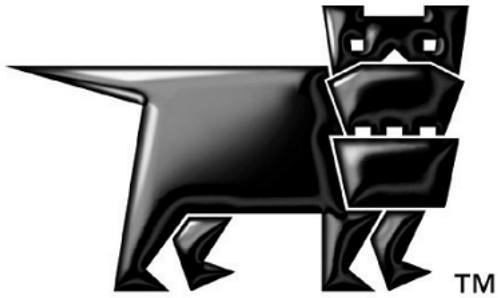
Table of Contents

Part C

Animation libraries

Chapter 15

Skinning



15.1 Introduction

Skinning provides a means to animate a model while reducing the "folding" and "creasing" of polygons as the model animates.

The process begins by defining a bone hierarchy in a model. The bone hierarchy is linked to the model's mesh, so that animating the bone hierarchy animates the mesh's vertices.

Up to four bones affect each vertex in the skin mesh. Each bone has a weighting value to determine how much influence it has in proportion to the others.

Skinning is supported by **RpSkin** and is covered in this chapter. **RpHAnim**, which implements a full-featured hierarchical animation system, also relies on **RpSkin** to provide skinning support for its own animation system. Details of using **RpHAnim** are covered in *The Hierarchical Animation Plugin Chapter*.

The **RpSkin** plugin uses the **RpHAnim** hierarchy to define how the *bones* of the model move. These bones are linked to vertices within an atomic with a matching structure so that animating the bones also animates the atomic.

It is important to understand that the hierarchy is a distinct entity, separate from the atomic. If their respective structures match, multiple hierarchies can be attached to a single skinned model. Similarly, multiple models can also use the same hierarchy.

15.2 Creating Skinning Data

Skinning data involves the generation of two sets of data:

1. The **RpSkin** data resides in an **rwID_CLUMP** chunk in a RenderWare Graphics binary stream, saved using the streaming of the **RpGeometry** that the **RpSkin** is attached to. This defines the relationship between the bone hierarchy and the skin.

2. The **RtAnimAnimation** animation data is stored in an **rwID_HANIMANIMATION** chunk in a binary stream. This defines the animation keyframes that each bone takes during its animation.

Any number of **RtAnimAnimation** animations can be applied to a single **RpHAnimHierarchy** controlling the skin's bones, providing that the bone hierarchy itself remains the same. This is because **RtAnimAnimation** animations are not explicitly linked to model data. So, as long as the model to which the animation is applied has a matching structure, the data will be valid.

It is possible to apply different animations to a hierarchy.

15.2.1 Attaching the RpSkin plugin

Before skinning is supported, the **RpSkin** plugin must be attached by calling the function **RpSkinPluginAttach()**. This registers the **RpSkin** extension to an **RpGeometry** (automatic streaming, constructor extension, and cloning functions with RenderWare Graphics).

15.2.2 Creating the RpSkin data

In most situations, the skinning data will be created at the model export stage. The key function for this purpose is **RpSkinCreate()**.

This function takes the following parameters:

- The number of vertices in the skin.
- The number of bones in the skin.
- An array of vertex weights, one per vertex.
- An array of vertex indices, one per vertex.
- An array of inverse matrices.

These parameters are explained below.

Number of vertices

To reserve memory for the skin, the number of vertices in the skin (normally equal to the number of vertices in the geometry mesh that is going to be skinned) is passed as a parameter.

Number of Bones

This value is extracted from the modeler when using the RenderWare Graphics exporter. This value is used as the array size for the array of inverse bone matrices. (See the **RpSkin** overview in the API Reference details of platform specific limitations.)

For models that requires more bones than could be supported at once by the target platform, it is necessary for them to be split into smaller groups. Each group would require only a subset of the bones in the model that would fit into the target platform. The function, **RtSkinSplitAtomicSplitGeometry()**, can split a skin model into groups where each group would require a given number of bones.

Vertex Weights

An array of **RwMatrixWeights** is passed to **RpSkinCreate()** as a parameter. The array length is determined as the number of vertices for which the skinning data is being created.

Each **RwMatrixWeights** structure is made up of four **RwReal**s. Each represents the weight of the corresponding bone in the array of vertex indices.

A weight may contain a value in the range 0.0 to 1.0. The sum of the four weights affecting the vertex should be 1.0 under normal conditions and, although values not totaling 1.0 give an officially undefined result, experience has shown that others can have interesting results.

Processing of weights for a bone will stop when the first weighting of 0.0 is found, and we assume that there are no more bones affecting the vertex. All the unused weights should be set to 0.0. The vertex weights must therefore be arranged such that all valid weights are first in the **RwMatrixWeights** structure and all zero weights are last. This can be done by sorting the weights into descending order. The vertex indices should reflect this sorted order.

The weights in the vertex weights array must be arranged in the same order as the vertices in the atomic.

Bone Vertex Indices

Each vertex can be affected by up to four bones, and the bone ID of each of the four bones are stored in here. The **RwUInt32** contains four packed **RwUInt8** values, each one of which contains an integer 0 to 255, so the maximum number of bones supported in a skeleton is 256.

The following macro will pack the indices into the bone vertex:

```
#define PACK(b1, b2, b3, b4) ( (b4 << 24) + (b3 << 16) +      \
                               (b2 << 8)  +  b1 )
```

Inverse Bone Matrices

The transformation from skin space to bone space, required at run-time for modifying vertex positions, is achieved by transforming the vertex position by the *Inverse Bone Matrix* (IBM). Since the root of the hierarchy is attached to the mesh in an **RwFrame** hierarchy, the IBM is the inverse of the overall transformation matrix—the *Local Transformation Matrix* (LTM)—from the mesh to the bone.

The **RpSkinCreate()** function requires an array of IBMs, stored in bone order. The length of the array is determined from the number of bones.

Within the hierarchy, **RwFrames** are stored for each bone node, indicating the transformation of that bone with respect to its parents.

The run-time procedure then performs the following operations:

1. Transforms vertex positions from their object space (skin space) into bone space using the IBMs.

2. Performs the key-frame based animation for that frame, which transform the vertex positions back into skin space in its animated pose.

3. The model is then rendered.

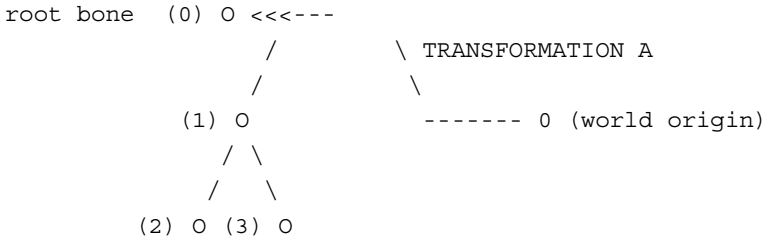
The inverse bone matrices are **RwMatrix**'s. There should be one per bone, i.e. the same number as you are passing to the **numBones** parameter of **RpSkinCreate()**.The inverse bone matrices describe the transformation matrix from "the root" bone in the hierarchy to "each" bone in the hierarchy.

To find the inverse bone matrix, one requires the transformation matrix from the bone to the root bone.

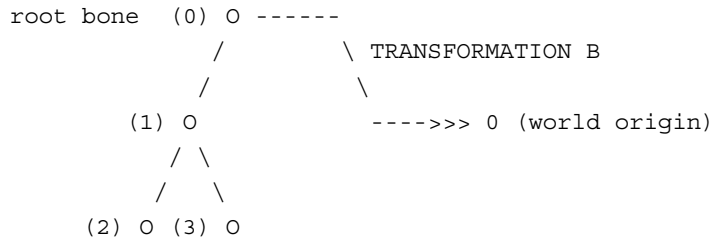
For instance for bone 3 this is obtained by:

(The results from each of the following 4 steps is shown on the diagrams.)

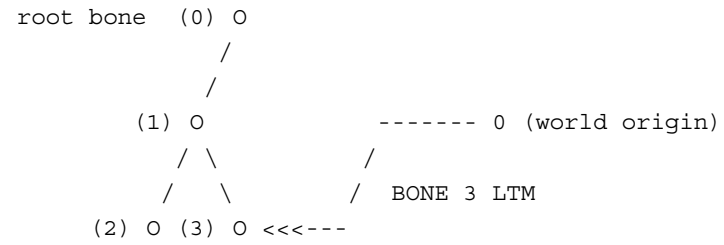
1. finding the transformation matrix from the world origin to the root bone,

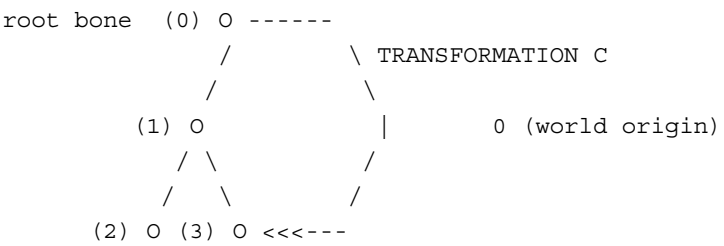


2. inverting this to find the transformation matrix from the root bone to the world origin,

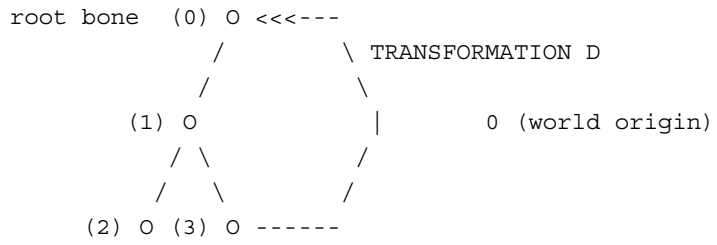


3. multiplying this by the LTM of bone 3 to find the transformation from the root bone to bone 3,





4. inverting this to find the transformation from bone 3 to the root bone.



The following code calculates the inverse bone matrix if you have the world transformation matrix for each bone. Calculate for each bone in turn.

```
{
    /*
     * INPUTS: node is a structure with the current
     *         bone world transformation stored in
     *         a frame root is a structure with the
     *         root bone world transformation stored
     *         in a frame.
     * OUTPUT: ibm is a pointer to the inverse bone
```

```
     *          matrix of the current node.
     */

    RwMatrix *invHierarchyRootMatrix;
    RwMatrix *rootRelativeMatrix;
    RwMatrix *LTM;
    RwMatrix *rootLTM;

    /* Calculate & store the inverse bone matrix *
     * within the hierarchy                      */

    invHierarchyRootMatrix = RwMatrixCreate();
    rootRelativeMatrix = RwMatrixCreate();
    LTM = RwFrameGetLTM(node->frame);
    rootLTM = RwFrameGetLTM(root->frame);

    RwMatrixInvert(invHierarchyRootMatrix, rootLTM);
    RwMatrixMultiply( rootRelativeMatrix,
                      LTM,
                      invHierarchyRootMatrix );
    RwMatrixInvert(ibm, rootRelativeMatrix);

    RwMatrixDestroy(invHierarchyRootMatrix);
    RwMatrixDestroy(rootRelativeMatrix);
}
```

15.2.3 Node IDs

The array of nodes in **RpHAnim** equates to the array of bones in **RpSkin**. The node IDs are usually generated automatically by the RenderWare Graphics exporter for the modeling package. An artist can, however, override the defaults and use their own IDs for a particular node (which relate directly to the bones). This facility makes it easier for programmers to locate any bones that require special treatment, as would be the case for many procedural animation techniques.

The node IDs must be unique within the hierarchy. Normally, a modeling package will supply unique bone IDs, but in the event that they are not tagged, unique IDs should be created dynamically within the exporter by default.

Assigning Node IDs within a Modeling Package

Support for **RpSkin** and **RpHAnim** is available for both the 3ds max and Maya modeling packages. As each modeling package has a different user interface, package-specific details on tagging objects can be found within the appropriate Artist Guide for your preferred package.

Extracting the Node ID at Run-time

The **RpHAnimIDGetIndex()** function returns the index from an **RpHAnimHierarchy** for a particular node ID. This index maps directly into the skin to bone matrix array. The skin to bone matrix array is retrieved with **RpSkinGetSkinToBoneMatrices()**. (Elements of the matrix array are of type **RwMatrix**.)

15.2.4 Destroying the RpSkin data

The RpSkin data is destroyed with **RpSkinDestroy()**, all internal and platform specific data is cleaned up automatically.

15.2.5 Querying the RpSkin data

The following functions give read-only access to the **RpSkin** data.

RpSkinGetNumBones() return the number of bones in the **RpSkin**.

RpSkinGetVertexBoneWeights() return the array of vertex bone weights (**RwMatrixWeights ***) in the **RpSkin**.

RpSkinGetVertexBoneIndices() return the array of packed vertex bone indices (**RwUInt32 ***) in the **RpSkin**.

RpSkinGetSkinToBoneMatrices() return the array of skin to bone matrices (**RwMatrix ***) in the **RpSkin**.

15.3 Using Skinning

When using skinning, there are three main objects to deal with:

- The **RpSkin** object
- The **RpHAnimHierarchy** object
- The **RtAnimAnimation** object

The use of **RpHAnimHierarchy** objects and **RtAnimAnimation** objects is described in *The Hierarchical Animation Plugin Chapter*.

15.3.1 The RpSkin Object

The **RpSkin** object contains the *skinning* data. The vertex weights, vertex indices and IBMs are all used by the skin rendering pipeline.

The **RpSkin** object should be attached to an **RpGeometry** with **RpSkinGeometrySetSkin()**. The **RpGeometry** vertices match up to the skinning vertex information in the **RpSkin**.

The **RpSkin** object attached to an **RpGeometry** can be retrieved with **RpSkinGeometryGetSkin()**.

When an **RpGeometry** is streamed in as part of **rwID_CLUMP** chunk, any attached **RpSkin** will also be streamed in. Likewise, any attached **RpSkin** will automatically be streamed out with an **RpGeometry**.

It is possible that there will be multiple **RpAtomic**s referencing the same **RpGeometry** instance. So the skin's animation data is attached to the **RpAtomic** instead of the **RpGeometry**. The hierarchy is attached to the atomic with **RpSkinAtomicSetHAnimHierarchy()**. An **RpAtomic**'s presently attached hierarchy can be retrieved with **RpSkinAtomicGetHAnimHierarchy()**.

Attaching the **RpSkin** to an **RpGeometry**, and then attaching an **RpHAnimHierarchy** to the **RpAtomic** referencing the **RpGeometry**, has fully setup the data required for *skinning* the mesh. However the rendering pipeline attached to the **RpAtomic** needs to be overloaded with a custom *skinning* pipeline. The default rendering pipeline knows nothing about the skinning data extension.

The **RpSkinType** enumeration lists the different rendering pipeline *types* available within the **RpSkin** plugin. At present these are:

- **rpSKINTYPEGENERIC** – pipeline renders generic skinned geometry.
- **rpSKINTYPEMATFX** – pipeline renders material effected skinned geometry.

- **rpSKINTYPETOON** – pipeline renders toon shaded skinned geometry.

Toon shading is available through the Toon plugin that is part of the FX Pack.

A skinned rendering pipeline is attached to the **RpAtomic** with **RpSkinAtomicSetType()**. The type of the present skinning pipeline can be queried from an **RpAtomic** with **RpSkinAtomicGetType()**.

RpSkin Libraries: **rpskin.lib, rpskinmatfx.lib**, and **rpskintoon.lib**

There are three versions of the **RpSkin** libraries in the RenderWare Graphics SDK. They are both fully featured versions on the **RpSkin** plugin, and they contain identical APIs. However, because the rendering pipelines are large we've taken the step to compile different versions of the plugin so that the user can select precisely the pipelines they will be using.

The **rpskin.lib** library only contains the **rpSKINTYPEGENERIC** pipeline.

Whereas the **rpskinmatfx.lib** library contains *both* the **rpSKINTYPEGENERIC** and **rpSKINTYPEMATFX** pipelines.

Finally the **rpskintoon.lib** library contains *both* the **rpSKINTYPEGENERIC** and **rpSKINTYPETOON** pipelines.

Only one of the skinning libraries should be used in an application at once.

RpSkin & **RpPatch**

The **RpPatch** plugin also supports skinned patch meshes. The **RpPatchMesh**es are used with the skinning plugin in a very similar ways to **RpGeometry**s.

RpPatchMeshSetSkin() should be used instead of **RpSkinGeometrySetSkin()**, and

RpPatchMeshGetSkin() should be used instead of **RpSkinGeometryGetSkin()**.

Once the **RpPatchMesh** has been attached to the **RpAtomic** (in place of an **RpGeometry**), the correct *skinning patch* rendering pipeline must be attached to the **RpAtomic**. There are two *skinning patch* rendering pipelines in the **RpPatch** plugin:

- **rpPATCHTYPESKIN** – pipeline renders skinned patches.
- **rpPATCHTYPESKINMATFX** – pipeline renders skinned material effected patches.

The **RpPatch** pipelines are attched with **RpPatchAtomicSetType()** instead of **RpSkinAtomicSetType()**.

For more details on using the skinning patch pipeline please read *The B-splines and Bézier Patches Chapter*.

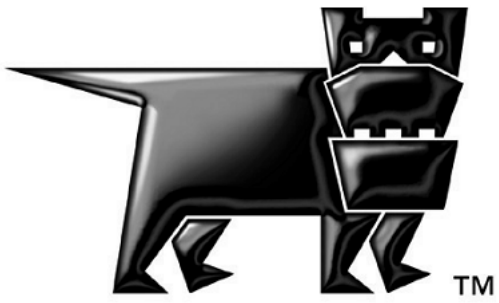
15.4 Examples

The core skinning features exposed by **RpSkin** are used by the **RpHAnim** examples listed below.

- **hanim1**
- **hanim2**
- **hanim3**
- **hanim4**
- **hanimkey**
- **hanimsub**

Chapter 16

Fundamental Types for Animation



16.1 Introduction

The RenderWare Graphics SDK contains support for quaternions (**RtQuat**), which provide functionality for orienting keyframes, and spherical linear interpolators (**RtSlerp**), which provide functionality for interpolating smoothly between these keyframes.

This chapter does not attempt to give an in-depth description of quaternions and Slerps, concentrating instead on detailing the support provided for them.

For further information on these subjects, see the resources listed in the *Recommended Reading* appendix.

16.2 Quaternions

The **RtQuat** toolkit exposes support for quaternions.

A quaternion consists of four elements; a real value and three imaginary values. Together, these can represent both scaling and rotation transforms, with the advantage that operations always produce orthogonal results—i.e., there is no shearing created by rounding errors, such as can occur when using matrices alone.

Quaternions are compact, compared to the nine values needed for a matrix, so they are useful on platforms where memory is at a premium or where the architecture favors smaller data structures.

The **rtquat.h** header file needs to be included and its library linked against.

16.2.1 Usage

Creation

Quaternions are small enough to be declared as automatic variables . They are initialized, rather than being declared as pointers and being allocated memory. For this reason, they don't need constructor or destructor functions and the **RtQuat** structure is transparent and can be addressed directly.

An **RtQuat** structure contains two elements:

- *real* – an **RwReal** value representing the real value
- *imag* – an **RwV3d** vector representing the imaginary values

These can be modified directly if required, although an **RtQuatInit()** function is provided by the API for convenience.

The following code fragment shows a unit quaternion being created and initialized:

```
RtQuat myQuat;

...

RtQuatInit( &myQuat, 0.5f, 0.5f, 0.5f, 0.5f );
```

Rotations

Quaternions are usually used to perform rotations. To this end, the **RtQuat** API includes the **RtQuatRotate()** to initialize a quaternion using a vector which represents the axis of rotation, and an angle representing the rotation itself.

Conversely, the **RtQuatQueryRotate()** function will return the axis and angle from a given quaternion.

Scaling

The length, or *modulus*, of a quaternion represents the scale. Although only rotation is usually required of a quaternion, the **RtQuatScale()** function is provided to allow the application to re-scale a quaternion, into a new quaternion.

Transforming Vectors

A quaternion rotation is often applied to a vector array. For this purpose, the **RtQuat** API provides the **RtQuatTransform()** function, which applies the transformation represented by a quaternion to an array of vectors.

API

The **RtQuat** toolkit provides a full-featured API and includes a number of quaternion operations. The operations and matching functions names are listed in the table below:

FUNCTION	PURPOSE
RtQuatAssign()	Copies one quaternion to another.
RtQuatConjugate()	Negates the imaginary parts of a quaternion.
RtQuatConvertToMatrix()	Takes a quaternion and converts it to the equivalent matrix.
RtQuatConvertFromMatrix()	Takes a matrix and returns the equivalent quaternion.
RtQuatAdd()	Adds two quaternions together, producing a third. (**A=B+C**)
RtQuatSub()	Calculates the difference between two quaternions. (**A=B-C**)
RtQuatIncrement()	Increments a quaternion by another. (Equivalent to **A+=B**)
RtQuatDecrement()	Decrements a quaternion by another. (Equivalent to **A-=B**)
RtQuatIncrementRealPart()	Increments the real part of a quaternion by a specified real value.
RtQuatDecrementRealPart()	Decrements the real part of a quaternion by a specified real value.
RtQuatMultiply()	Calculates the (non-commutative) product of two quaternions.
RtQuatNegate()	Negates a quaternion to the additive inverse.
RtQuatModulus()	Returns the *modulus*—the scaling component—of a quaternion.
RtQuatModulusSquared()	Returns the square of the modulus of a quaternion.
RtQuatExp()	Calculates the exponential of a quaternion.
RtQuatPow()	Calculates the power of a quaternion.
RtQuatLog()	Calculates the logarithm of a quaternion.

RtQuatQueryRotate()	Determines the rotation represented by a quaternion. The rotation is returned as a unit vector along the axis of rotation, and an angle of rotation. The rotation component has two possible descriptions since a rotation about an axis of theta degrees is equivalent to a rotation about an axis pointing in the opposite direction by an angle of 360°-theta in the reverse direction. The rotation with the smaller angle is returned.
RtQuatRotate()	Builds a rotation quaternion from the given axis and angle of rotation.
RtQuatScale()	Scales a quaternion by the specified factor.
RtQuatSquareRoot()	Calculates the square root of the specified quaternion.
RtQuatTransformVectors()	Uses the given quaternion describing a transformation and applies it to the specified array of vectors. The results are then placed in another array (which may be the same array as the source).
RtQuatUnitConvertToMatrix()	Converts from a unit quaternion to a matrix.
RtQuatUnitExp()	Calculates the exponential of a unit quaternion.
RtQuatUnitLog()	Calculates the logarithm of a unit quaternion.
RtQuatUnitPow()	Calculates the power of a unit quaternion.
RtQuatReciprocal()	Reciprocates a quaternion to its multiplicative inverse.

The **RtSlerp** toolkit exposes support for spherical linear interpolations—so-called "Slerps". The **rtslerp.h** header file needs to be included and its library linked against.

This toolkit works closely with quaternions so it also requires **RtQuat** to be linked in to your application.

16.3 Spherical Linear Interpolation

16.3.1 Applications

Slerps are used to spherically interpolate between two quaternions—each typically representing a key-frame orientation. This is a common requirement in animations where linear interpolation is not always appropriate—animating skinned models, for example.

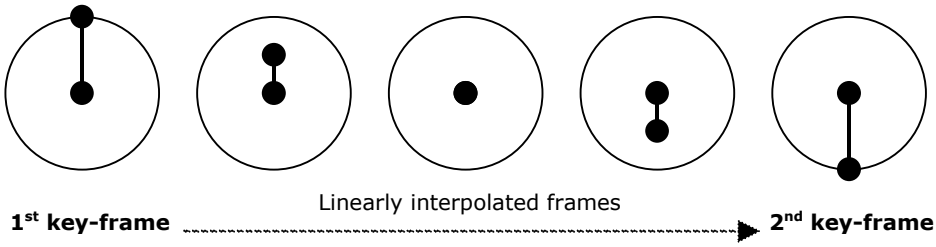
The primary use for both quaternions and Slerps is interpolating rotations in animation. Quaternions are often used together with spherical linear interpolators to interpolate rotations along an arc; the result is generally more "natural" than a linear interpolation.

RenderWare Graphics supports Slerps through the **`RtSlerp`** toolkit, which is covered in Section 1.3 of this chapter.

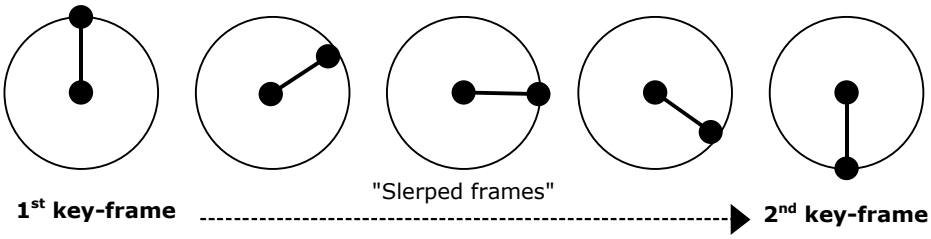
Why not use Morph Targets?

Morph target animation uses linear interpolation exclusively. This means that the interpolated frames created between one key-frame and the next are always positioned in a straight line. This is great for animating, say, a spaceship. But many situations call for interpolating in an arc, rather than a straight line.

A simple example is an analog clock face. The following sequence shows how the hands would move if linear interpolation is used between the two keyframes:



Because Slerps interpolate the animation along an arc rather than a straight line, the clock face above would be animated as shown below. (The frames are said to have been "Slerped", which is an abbreviation for "Spherical-linearly interpolated").



The most common use of Slerps is for skinned animations based around virtual "bones". In such animations, the bones must be animated such that the paths describe curves in space, rather than lines—i.e. the bones maintain their length throughout the interpolation.

16.3.2 Usage

Creation

Slerps are represented by the **RtSlerp** object. This object is transparent and defined as follows:

- *angle* – The angle, in degrees, between source and destination. (**RwReal**).
- *axis* – The axis of rotation for the Slerp. (**RwV3d**).
- *endMat* – The end matrix. (**RwMatrix ***).
- *matRefMask* – Flags specifying which matrices are *not* managed by the Slerp object. (**RwInt32**).
- *startMat* – The start matrix. (**RwMatrix ***).
- *useLerp* – If **TRUE** then linear interpolations will be used. (**RwBool**).

The constants listed below are used with the **matRefMask** element:

- **rtSLERPREFNONE** – Both start and end matrices are copied into the structure rather than accessed by reference.
- **rtSLERPREFSTARTMAT** – The start matrix is referenced and should be destroyed by the application when no longer required. The end matrix is copied.
- **rtSLERPREFENDMAT** – The end matrix is referenced and should be destroyed by the application when no longer required. The start matrix is copied.
- **rtSLERPREFALL** – Both start and end matrices are referenced and should be destroyed by the application when no longer required.

Accessing by reference is usually the preferred option, but it should be noted that it is usually necessary for Slerp objects to persist across rendering cycles, so that they can be used to continue the interpolation. Copying of matrices requires more memory, but does have the advantage that persistent Slerps may be easier to maintain using this method.

Creating a valid Slerp usually requires two calls:

- The first is to **RtSlerpCreate()**. This function takes one argument defining the flags for the **matRefMask** property of the Slerp. One of the constants listed above should be used for this.

- Secondly, the Slerp needs to be initialized with valid data using **`RtSlerpInitialize()`**. This takes a pointer to a Slerp object and pointers to the two key-frame matrices that are to be used. These matrices will be either copied to, or referenced by, the Slerp object according to the flags set in **`RtSlerpCreate()`**.

`RtSlerpCreate()` *must* be called before **`RtSlerpInitialize()`** in order to ensure the flags are set. If this is not done, the result is undefined.

Caches

Two caching structures are exposed by the **`RtSlerp`** toolkit: **`RtQuatSlerpCache`** and **`RtQuatSlerpArgandCache`**. These are used internally by the API to improve performance—either method can be selected.

Full documentation of these structures can be found in the API Reference.

The required cache must be setup before the Slerp can be used for interpolation. The relevant functions are **`RtQuatSetupSlerpCache()`** and **`RtQuatSetupSlerpArgandCache()`**.

Performing Spherical Linear Interpolations

With all the requisite structures initialized, it is now possible to perform the spherical linear interpolation. Two functions are available for this purpose:

- **`RtQuatSlerp()`**, which uses the **`RtQuatSlerpCache`** and
- **`RtQuatSlerpArgand()`**, which makes use of the **`RtQuatSlerpArgandCache()`** instead.

16.4 Summary

The RenderWare Graphics SDK contains support for quaternions (**RtQuat**) and spherical linear interpolators (**RtSlerp**).

16.4.1 Quaternions

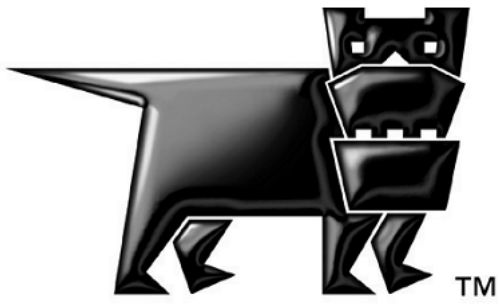
- The **RtQuat** toolkit exposes support for quaternions, which represent both rotation and scaling.
- A quaternion consists of four elements: one real part, and three imaginary parts.
- Morph target animation uses linear interpolation exclusively.
- Quaternions are often used together with spherical linear interpolators—also known as Slerps—to interpolate rotations along an arc. The result generally appears more natural than a linear interpolation. Slerps are implemented by the **RtSlerp** toolkit.
- The **RpHAnim** plugin uses the **RtQuat** toolkit to implement rotational animation.

16.4.2 Spherical Linear Interpolation

- The **RtSlerp** toolkit exposes support for Slerps.
- Slerps are represented by the **RtSlerp** object.

Chapter 17

The Animation Toolkit



17.1 Introduction

The animation toolkit (**RtAnim**) provides support for keyframe based animation systems.

RtAnim works off two main objects: **RtAnimInterpolator** and **RtAnimAnimation**. The interpolator holds the state of the animation during playback, while the animation contains the actual animation data.

As RtAnim does not have any knowledge about the data being animated, it will refer to the animated objects as nodes.

Animation data takes the form of a series of keyframes for each node in an **RtAnimInterpolator**, each describing the state of that node at a specific point in time. Smooth animation results from interpolation between keyframe pairs.

The state held by the interpolator is an array of interpolated keyframes along with references to the current keyframe pairs in the animation that are being interpolated. This last array is exposed for more advanced uses such as allowing procedural animation at the keyframe level.

RtAnim supports overloaded keyframe schemes. This allows a user to register a block of functions for processing a user-defined keyframe structure allowing the system to be used on any type of animation.

17.2 Creating an Interpolation Schemes

In order to play specific types of animation, either higher order or optimized types, you need to define a new interpolation scheme.

Implementing an interpolation scheme comprises the definition of a structure for your keyframe data and a number of functions allowing the animation system to process your keyframes. Once these functions are defined they can be registered with the **RtAnim** toolkit and then all the standard animation functions can be used.

To fully describe an interpolation scheme you need to implement all the functions, that are contained in the **RtAnimInterpolatorInfo** structure, that deal with your keyframe type. The structure is defined as follows:

```
struct RtAnimInterpolatorInfo
{
    RwInt32                              typeID;
    RwInt32                              keyFrameSize;
    RtAnimKeyFrameApplyCallBack          keyFrameApplyCB;
    RtAnimKeyFrameBlendCallBack          keyFrameBlendCB;
    RtAnimKeyFrameInterpolateCallBack    keyFrameInterpolateCB;
    RtAnimKeyFrameAddCallBack            keyFrameAddCB;
    RtAnimKeyFrameMulRecipCallBack       keyFrameMulRecipCB;
    RtAnimKeyFrameStreamReadCallBack     keyFrameStreamReadCB;
    RtAnimKeyFrameStreamWriteCallBack    keyFrameStreamWriteCB;
    RtAnimKeyFrameStreamGetSizeCallBack keyFrameStreamGetSizeCB;
};
```

The types and usage of the structure members are as follows:

RwInt32 typeID: This is an ID that should uniquely identify your interpolation scheme and will be used to link animations to their appropriate interpolation scheme. It is suggested that developers construct unique IDs using **MAKECHUNKID(vendorID, typeID)**.

RwInt32 keyframeSize: This defines the size of the keyframe data in bytes. It is used so that the generic animation functions know how to step through an array of keyframe data. Because the interpolator also caches animation keyframes it is given a maximum keyframe size at creation. This maximum size must be greater than or equal to the keyframe size of the interpolation scheme to use the animation on that interpolator.

void keyFrameApplyCB(void * result, void * voidIFrame): This function should cast the **voidIFrame** pointer into the interpolated keyframe type it supports and convert the interpolated frame into the required result type storing the result in the **result** pointer. As **RtAnim** does not have any knowledge about the resulting data format, it's your responsibility to create an **apply** function that will go through the interpolated keyframe and apply the change to the animation data. This callback is only provided as a way to manage different keyframe types being applied to the same type of destination data.

void keyFrameBlendCB(void * pVoidOut, void * pVoidIn1, void * pVoidIn2, RwReal fAlpha): This function should cast the **pVoidIn1** and **pVoidIn2** pointers to the supported interpolated keyframe type and store a blend from **pVoidIn1** to **pVoidIn2,** based on **fAlpha**, in the interpolated keyframe pointed to by **pVoidOut**. This is for the purpose of blending the states of two **RtAnimInterpolator** objects into a third.

void keyFrameInterpolateCB(void * pVoidOut, void * pVoidIn1, void * pVoidIn2, RwReal time): This function should cast the **pVoidIn1** and **pVoidIn2** pointers to the supported keyframe type and interpolate between the two based on **time**; the result should be stored in the interpolated keyframe pointed to by **pVoidOut**.

void keyFrameAddCB(void * pVoidOut, void * pVoidIn1, void * pVoidIn2): This function should cast the **pVoidIn1** and **pVoidIn2** pointers to the supported interpolated keyframe type and store the sum of the keyframes in the interpolated keyframe pointed to by **pVoidOut**. This is used for delta animations where the states of two **RtAnimInterpolator** objects are added together.

void keyFrameMulRecipCB(void * pVoidFrame, void * pVoidStart): This function should cast the **pVoidFrame** and **pVoidStart** pointers to the supported keyframe type and then multiply **pVoidFrame** by the reciprocal of **pVoidStart**. This is intended to convert the **pVoidFrame** keyframe into a delta from **pVoidStart**.

RtAnimanimation * keyFrameStreamReadCB(RwStream * stream, RtAnimanimation * Animation): This function should read an array of keyframes from the stream. The number of keyframes and the memory to store them is passed in via the **Animation** pointer.

RwBool keyFrameStreamWriteCB(RtAnimation * Animation, RwStream * stream): This function should write the keyframes in the **Animation** out to the supplied stream.

RwInt32 keyFrameStreamGetSizeCB(RtAnimation * Animation) : This function should return the size of the keyframe data in the **Animation** in bytes.

Once these functions have been defined and added to the **RtAnimInterpolatorInfo** structure they can be registered with the **RtAnim** toolkit using **RtAnimRegisterInterpolationScheme** passing in the structure. Once registered **RtAnim** can support creation and usage of keyframes based on the new type ID.

An example of the creation and use of an interpolation schemes is shown in the **Anim** example. This demonstrates a scheme that animates light colors and radius.

17.3 Creating Animation Data

An **RtAnimAnimation** object represents an animation and is streamed out to a separate file, usually with the .ANM extension, that holds the animation data. This defines the animation keyframes that are used to animate nodes.

Any number of **RtAnimAnimation** animations can be applied to a single node, providing that the topology of the nodes itself remains the same. This is because **RtAnimAnimation** animations are not explicitly linked to model data, but simply rely upon a matching structure. So, as long as the model to which the animation is applied has a matching structure, the data will be valid.

Multiple animations are either applied sequentially or using various blending techniques described later.

17.3.1 The API

The function used to create an **RtAnimAnimation** structure is **RtAnimAnimationCreate()**.

This function requires the following parameters:

- An interpolation scheme type ID.
- The number of keyframes.
- Flags (for future expansion/customization).
- The duration of the animation (the time elapsed between the first and last keyframes).

An **RtAnimAnimtion** structure is returned that contains a **void * pFrames** pointer with enough memory allocated for the number of keyframes requested with a keyframe size based on the interpolation scheme type ID.

17.3.2 The Animation Keyframe Structure

Each keyframe must contain the following elements at the start of their structure:

- Previous keyframe pointer.
- Time at which the keyframe occurs in the animation.

These elements are defined by the **RtAnimKeyFrameHeader** structure. This allows standard operations to be applied to keyframes without knowledge of any specific overloaded interpolation scheme.

The previous keyframe pointer is to allow the animation to be played backwards efficiently. This pointer points to the previous keyframe for the node that this keyframe will be applied to. See details in the following section on keyframe ordering.

17.3.3 Keyframe Ordering

The keyframe data does not contain an explicit link to the node it will be applied to. Instead, it is determined implicitly from the ordering of the keyframes in the array. For data cache coherence, the keyframes are sorted in time order.

Each and every node has a keyframe at the beginning and end of the animation. Then there are optional additional keyframes in-between. The first keyframe is stored for each node in turn, then the second for each node in turn. These are used to initialize the first interpolation for each node.

The following tables show an example animation sequence (top), with the order in which the keyframes would be stored (bottom):

NODE NUMBER	(TIME) 0.0	1.2	2.1	2.9	3.2	4.0
0	A		B		C	D
1	E		F		G	H
2	I	J		K		L
3	M					N

KEYFRAME	TIME	NOTES
A	0.0	Initial keyframe (Node 0)
E	0.0	Initial keyframe (Node 1)
I	0.0	Initial keyframe (Node 2)
M	0.0	Initial keyframe (Node 3)
B	2.1	Second keyframe (Node 0)
F	2.1	Second keyframe (Node 1)
J	1.2	Second keyframe (Node 2)
N	4.0	Second keyframe (Node 3)
K	2.9	Interpolation **I** to **J** ends. **J** to **K** begins. (Node 2)
C	3.2	Interpolation **A** to **B** ends. **B** to **C** begins. (Node 0)
G	3.2	Interpolation **E** to **F** ends. **F** to **G** begins. (Node 1)
L	4.0	Interpolation **J** to **K** ends. **K** to **L** begins. (Node 2)
D	4.0	Interpolation **B** to **C** ends. **C** to **D** begins. (Node 0)
H	4.0	Interpolation **F** to **G** ends. **G** to **H** begins. (Node 1)

Keyframe Sorting

An unordered array of keyframes may be sorted in a conventional way using the following as primary and secondary sort keys:

1. The time of the previous keyframe for the node
2. The node index

17.3.4 Streaming Animation Data

Once completed, the **RtAnimAnimation** structure is then written to disk using the **RtAnimAnimationStreamWrite()** or **RtAnimAnimationWrite()** functions. This either writes an animation into a generic **RwStream** or writes an **.ANM** file to a specified filename.

As long as the same node organization is present, **.ANM** files can be applied to one or more data structures.

For example, when using **Hanim,** if two clumps, representing different humanoid characters, have the same node interpolator, then both clumps could share keyframe sequences.

17.3.5 Sub-Animations

Sub-animations can be treated almost exactly like normal animations at runtime and the animation data for them is usually exported in a similar way to the export of a normal animation. In order to process data for a sub-animation simply follow the same process as for a normal animation but only process keyframes for nodes contained within the sub animation.

17.4 Using RtAnim At Runtime

17.4.1 Concepts of Running Animations

Once you have an **RtAnimAnimation** loaded you can create an **RtAnimInterpolator** and run a simple animation. It is important to understand the main stages of updating the animation in order to optimize runtime performance.

The **RtAnimInterpolator** holds an array of interpolated keyframes, one for each node of the animation. As well as storing the interpolated data values, each of these reference the current start and end keyframes from the original animation that are currently being used for the interpolation.

An interpolator is normally updated by 'adding time' to advance it forwards through an animation. This will update the current references to keyframe pairs for each node if any of them have expired, and will also generate interpolated data values according to the current scheme. In some situations, an interpolator may be updated as the result of blending the states of two other interpolators. Either way, when the processing is complete, the interpolated keyframe data may be applied to the animated object.

17.4.2 The Interpolator

The interpolator holds the current state of the animation as well as the array of interpolated keyframes.

```
struct RtAnimInterpolator
{
    RtAnimAnimation                 *pCurrentAnim;
    RwReal                          currentTime;
    void                            *pNextFrame;
    RtAnimCallBack                  pAnimCallBack;
    void                            *pAnimCallBackData;
    RwReal                          animCallBackTime;
    RtAnimCallBack                  pAnimLoopCallBack;
    void                            *pAnimLoopCallBackData;
    RwInt32                         maxKeyFrameSize;
    RwInt32                         currentKeyFrameSize;
    RwInt32                         numNodes;
    RwBool                          isSubInterpolator;
    RwInt32                         offsetInParent;
    RtAnimInterpolator              *parentAnimation;
    RtAnimKeyFrameApplyCallBack     keyFrameApplyCB;
    RtAnimKeyFrameBlendCallBack     keyFrameBlendCB;
    RtAnimKeyFrameInterpolateCallBack keyFrameInterpolateCB;
    RtAnimKeyFrameAddCallBack       keyFrameAddCB;
};
```

The types and usage of the structure members are as follows:

RtAnimAnimation *pCurrentAnim: Holds the pointer to the animation currently played by the interpolator.

RwReal currentTime: Current time of the animation, normally this is between **0.0f** and the animation's duration.

void *pNextFrame: This holds the pointer to the next keyframe, as the keyframes are stored by time order, the interpolator uses this behavior to ensure quick access to the next keyframe.

RtAnimCallBack pAnimCallBack: This callback function can be called at a certain time while the animation is running, enabling you to perform specific actions.

void *pAnimCallBackData: Void pointer to some data that you can pass to the animation callback.

RwReal animCallBackTime: Trigger time for the animation callback.

RtAnimCallBack pAnimLoopCallBack: This callback function is called whenever an interpolator reaches the end of the animation, enabling control over playing of the animation (by default an interpolator will loop indefinitely).

void *pAnimLoopCallBackData: Void pointer to some data that you can pass to the animation loop callback.

RwInt32 maxKeyFrameSize: Maximum size, in bytes, of keyframes usable on this animation (set at creation time). It is used at creation time to ensure that if the animation attached to the interpolator changes to one with a different keyframe size, the interpolator has enough memory to store it.

RwInt32 currentKeyFrameSize: Size of keyframes in the current animation in bytes.

RwInt32 numNodes: Number of nodes driven by the animation.

RwBool isSubInterpolator: States if the interpolator is a normal or a sub-interpolator. (See Sub-Interpolator Animations.)

Creating the Interpolator

The function used to create an **RtAnimInterpolator** structure is **RtAnimInterpolatorCreate()**.

This function requires the following parameters:

- The number of nodes to be supported by that interpolator.
- The maximum size of keyframe to be supported by that interpolator.

An **RtAnimInterpolator** structure is returned.

17.4.3 Applying and Running a Basic Animation

The first step is to call **RtAnimInterpolatorSetCurrentAnim**. This applies the animation to the interpolator and initializes the interpolator state to zero time in the animation. At this stage the array of interpolated keyframes will be initialized to the first keyframe of the animation.

It should be noted that all animations have an interpolation scheme associated with them (based on their type ID), and each scheme knows the size of its keyframe data. If you try to apply an animation to an interpolator where the animation keyframe size is greater than the maximum keyframe size specified when the interpolator was created then the assignment will fail. If the assignment succeeds it will also update the interpolation scheme functions that the interpolator will use to run the animation.

To move forwards and backwards through the animation you call **RtAnimInterpolatorAddAnimTime** or **RtAnimInterpolatorSubAnimTime** respectively. These calls ensure for each node that the pairs of start and end keyframes are set to the correct keyframes for interpolation and will then interpolate to the correct time. After these calls, the interpolator will have a correct set of interpolated keyframes for the current time.

Another function allowing you to move through the animation is **RtAnimInterpolatorSetCurrentTime**. This will involve an extra function call and merely calculates the offset and calls **RtAnimInterpolatorAddAnimTime** or **RtAnimInterpolatorSubAnimTime**. Hence if possible calculate the offset and call the **RtAnimInterpolatorAddAnimTime** or **RtAnimInterpolatorSubAnimTime** functions directly.

These animation functions mentioned are generic and will work regardless of the animation scheme used by the animation you are trying to play.

17.4.4 Animation CallBacks

Two different callbacks are available when running an animation. These are set by calling **RtAnimInterpolatorSetAnimCallBack** and **RtAnimInterpolatorSetAnimLoopCallBack**.

RtAnimInterpolatorSetAnimCallBack will setup a callback to be called at a specific time in an animation. The function takes a callback pointer, a time and a pointer to user data to be passed into the callback. An example of using this callback would be in a walk cycle if you wanted a callback triggered at the times when the characters feet contact the floor.

RtAnimInterpolatorSetAnimLoopCallBack sets up a callback to be called every time the animation loops. This function takes just the callback pointer and user data pointer. This is equivalent to always setting a standard callback at time == duration of the current animation.

Both these callbacks are called during animation update functions such as **RtAnimInterpolatorAddAnimTime**. In each case the animation update will occur before the callback is called, i.e. if an update is taking an animation to time == 2.0 seconds and your callback was at 1.9 seconds, the animation state will be at 2.0 seconds when the callback is executed.

The return state of the **RtAnimCallBack** type is a pointer to the interpolator. If the return from the callback is NULL, the callback will be disabled and will not be called again until it is reset with one of the set callback functions.

17.4.5 Blending Between Animations

The simplest example of blending between animations is a transfer from one animation to another where the end state of animation 1 and start state of animation 2 are not common. In this case you perform a blend to interpolate from a state in animation 1 to a state from animation 2.

A common way to initialize the blend is to use one of the callbacks specified earlier, either a loop callback to blend from the end of the animation or possibly a standard callback if you want to blend before the animation ends.

Blending **RtAnimInterpolator** requires at least two **RtAnimInterpolator** structures and more commonly three. Each interpolator holds state in an animation. Hence we require a state to blend from, a state to blend to and an interpolator to hold the state of the blend result. We'll call these interpolators *In1*, *In2* and *Out*.

Although each interpolator doesn't require an attached animation, the process of attaching an animation also sets up the blending functions used for an interpolator (since they are linked to keyframe type). A newly created interpolator has no blending functions setup but they can be initialized by calling **RtAnimInterpolatorSetKeyFrameCallBacks** passing in the ID of the interpolation scheme you wish to use.

Before starting the blend ensure that *In1* and *In2* hold the state you wish to blend from and to. It's perfectly valid for this state to change further through the blend but no updates will occur because of the blend calls.

To blend between the two interpolators call **RtAnimInterpolatorBlend** passing in the two interpolators and an alpha value where 0.0 results in In1 and 1.0 will result in In2. The result of the blend will be stored in *Out*.

The result of the blend can then be used as the input to another blend. As a result an almost unlimited number of animations can be blended together. This can be used to create complex animations from a small selection of base animations.

As stated before, you can perform blending with two interpolators where the output interpolator is one of the input interpolators. This would however overwrite the input interpolator state potentially requiring regeneration.

There is an example of blending included in the RenderWare Graphics SDK. HAnim1 demonstrates blending from the end of one animation to the start of a second animation using an **RpHAnim** animation.

17.5 Sub-Interpolator Animations

Sub-interpolators appear and act just like standard interpolators. However they can be used to efficiently update sub-groups of nodes in an **RtAnimInterpolator**. Typically this scheme is used where the nodes represent a hierarchy of objects.

To be able to use sub-interpolator animation requires two major steps. One is to create animations that fit the sub-interpolator and the second is to create the sub-interpolator itself.

To create a sub-interpolator you need to know the array index of the root node of the sub-interpolator you want to create. Once you have the index call **RtAnimInterpolatorCreateSubInterpolator** that takes the following parameters:

- **pParentInterpolator** – An **RtAnimInterpolator** containing the branch you want to create the sub-interpolator within.
- **startNode** – This is the array index in the parent interpolator of the node you want to be the root of the sub-interpolator. This is used to calculate offsets into the parent interpolators structures.
- **numNode** –The number of nodes the sub-interpolator needs to hold. The **numNode** added to the **startNode** index should be less or equal than the total number of node in the parent interpolator.
- **MaxKeyframeSize** – This allows you to set a different maximum keyframe size in the sub-interpolator from the parent interpolator. Passing in –1 will use the same size as the parent.

The return value is an **RtAnimInterpolator** pointer, which will appear just like a normal interpolator except that its **isSubInterpolator** will be set to **TRUE**.

To use a sub-interpolator simply use it in the same way you would for a normal interpolator. However note that when applying the interpolators to the result data, the sub-interpolator application usually happens ***after*** the main interpolator. Because of this, the sub-interpolator will overwrite any animation applied by the main interpolator.

Sub-interpolators can be blended together like any standard interpolator using the **RtAnimInterpolatorBlend** assuming the topology of the sub-interpolators matches. However sub-interpolators can also be blended with an interpolator with the same topology as their parent interpolator (i.e. the one they were created from). To perform these blend operations use the function **RtAnimInterpolatorBlendSubInterpolator**. This function allows a sub-interpolator and parent interpolator to be blended together with the output interpolator matching either the parent or sub-interpolator.

In the case where the interpolator matches the parent, all nodes present in the sub-interpolator will be blended into the output interpolator, and all nodes present only in the parent interpolator will be copied to the output interpolator.

Where the output interpolator matches the topology of the sub-interpolator , just the nodes present in the sub-interpolator will be blended into the output interpolator.

This extended support means you do not need to have all the parent interpolator animations duplicated as sub-interpolator animations. One example of this would be a case where you have a cyclic walk animation for the parent interpolator and want to blend it's state for the characters leg with a sub interpolator animation representing just the leg kicking.

The **HAnimSub** example in the RenderWare Graphics SDK demonstrates the use of sub interpolator animations using an **RpHAnim** animation.

17.6 Delta Animations

Delta animations work well when you want to add a number of small effects to a basic animation. It's necessary to take the deltas from a common state in the animation to ensure they can be used together.

In order to make a delta animation you simply need to pass in an animation, the number of nodes in the animation and a time within the animation to calculate a delta from. This will convert all the keyframes to be deltas from the state at the given time. For example

```
RtAnimAnimationMakeDelta(animation, 43, 0.0f);
```

This would change the animation **animation** so that it represents an animation of delta positions based on its state at time 0.0. This process operates on the data in place so the original animation will be destroyed. Because this process can be time consuming it is sensible to convert all the animations offline and stream them out to disk. A delta animation appears identical to a normal animation so must be identified as a delta by the application.

To use a delta animation you then run the delta animation on an **RtAnimInterpolator** to get it to the state you require. At this stage the deltas will be held in the interpolated keyframe array on the delta interpolator. This can then be added to the state of any other interpolator using the function **RtAnimInterpolatorAddTogether**. It is also possible to use all the different blending and sub-interpolator techniques on delta animations and then add the result of more complex operations to other interpolators at the end.

The **HAnim2** example in the RenderWare Graphics SDK demonstrates the use of delta animations using an **RpHAnim** animation.

17.7 Procedural Animation

Procedural animation in **RtAnim** can be done at one of two different stages. In each case there are requirements relating to what stage of the animation update you're up to. The two stages are *source animation data and interpolated keyframe data.*

17.7.1 Procedural Modification of Source Animation Data

Procedurally modifying the source animation data requires knowledge of the ordering of keyframe data in the animations (see the section on creating **RtAnim** data for details). Given an **RtAnimAnimation** object a user can retrieve its **typeID** using **RtAnimAnimationGetTypeID**. This allows them to look up the interpolation scheme information using **RtAnimGetInterpolatorInfo**, which includes the keyframe size of the scheme. Using this you can step through animation keyframes and modify them as required.

Animation source data needs to be modified before performing any animation updates and can be mixed with other procedural updates which happen later in the process.

17.7.2 Procedural Modification of Interpolated Keyframes

Procedural modification of interpolated keyframes needs to happen between calling **RtAnimInterpolatorAddAnimTime** and using the interpolator, either for blending operations or rebuilding the animated data. To modify the interpolated keyframes with the source data, you need to retrieve information about the interpolation scheme to find out how to deal with the keyframes. The interpolated keyframes can be accessed from any interpolator and can be the result of adding time to an interpolator or the result of blending interpolators together.

The macro **rtANIMGETINTERPFRAME()** takes an interpolator and a node index in that interpolator, and returns a void * pointer to the interpolated keyframe for that specific node. This keyframe data can be modified and the results then used in further blending operations or used for rendering following an update of the animated data.

The **HAnim3** example included in the RenderWare Graphics SDK demonstrates applying node translations procedurally in the interpolated keyframes using an **RpHanim** animation.

17.8 Summary

This chapter has dealt with running the **RtAnim** animation system.

It has dealt with the three major steps involved in the process of creating, setting up and using animation data, and also the extended features available for use from the **RtAnim** toolkit to achieve better results.

The basic features were:

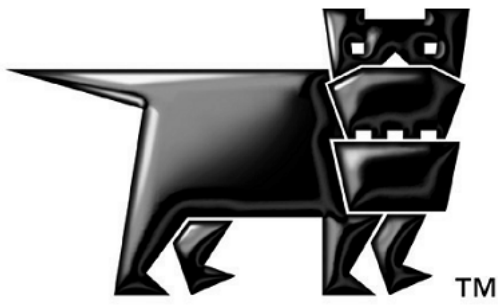
- Creating an Interpolation Schemes through the implementation of customized keyframe types and interpolation schemes.
- Creating animation data, comprising the creation of both the topological interpolator structure and the animation data that fits that structure.
- Using the animation data at runtime including blending together multiple animations to achieve a greater effect.

The extended features described were:

- Using Delta animation to add details and effects to animations.
- Procedurally modifying the animations, both by modifying the source data and modifying the state held with the **RtAnim** structures.

Chapter 18

The Hierarchical Animation Plugin



18.1 Introduction

The Hierarchical Animation (**HAnim**) plugin manages the animation of a hierarchically linked set of nodes related by matrix transformations. These nodes may represent anything, but in the common use of **RpHAnim** they are used to represent an animating hierarchy of objects. These objects may be animated rigidly or may be bones used for skinned animation supported by **RpSkin**.

The way in which **RpHAnim** works is based on **RtAnim**, using three base objects: the **RpHAnimHierarchy, RtAnimInterpolator** and **RtAnimAnimation**. The hierarchy holds a description of the topology, which will be animated, the interpolator holds the state during animation, and the **RtAnimAnimation** holds the actual animation data.

Animation data takes the form of a series of keyframes for each node in a **RpHAnimHierarchy**, each describing the state of that node at a specific point in time. Smooth animation results from interpolation between keyframe pairs.

The state held by the interpolator is in the form of three arrays of keyframes and an optional (present by default) array of matrices. The matrix array holds the result of keyframe to matrix conversions performed during animation updates. This array can be used to drive RenderWare Graphics skinning through **RpSkin**.

The hierarchy can also be attached to sets of **RwFrame**s allowing the animation system to drive standard RenderWare Graphics **RwFrame** hierarchies allowing rigid body animation as well as skinned object animation.

RpHAnim also support overloaded keyframe schemes through **RtAnim**. This allows a user to register a block of functions for processing a user defined keyframe structure allowing the system to be extended beyond quaternion and translation animation.

As **RpHAnim** is based on **RtAnim** which provides the base keyframing services, it is recommended that you read the **RtAnim** chapter before proceeding to **RpHAnim**.

18.2 Creating HAnim Data

Creating **HAnim** data is comprised of two distinct stages:

1. Creating data describing the hierarchical structure of the nodes in a **RpHAnimHierarchy** structure. This data is attached to a **RwFrame** within a **RpClump**, and is streamed out as part of an **rwID_CLUMP** chunk in a RenderWare Graphics binary stream.
2. Creating the **RtAnimAnimation** data that represents the animation and is streamed out to an **rwID_HANIMANIMATION** chunk in a binary stream. This defines the animation keyframes that are used to animate a hierarchy of nodes.

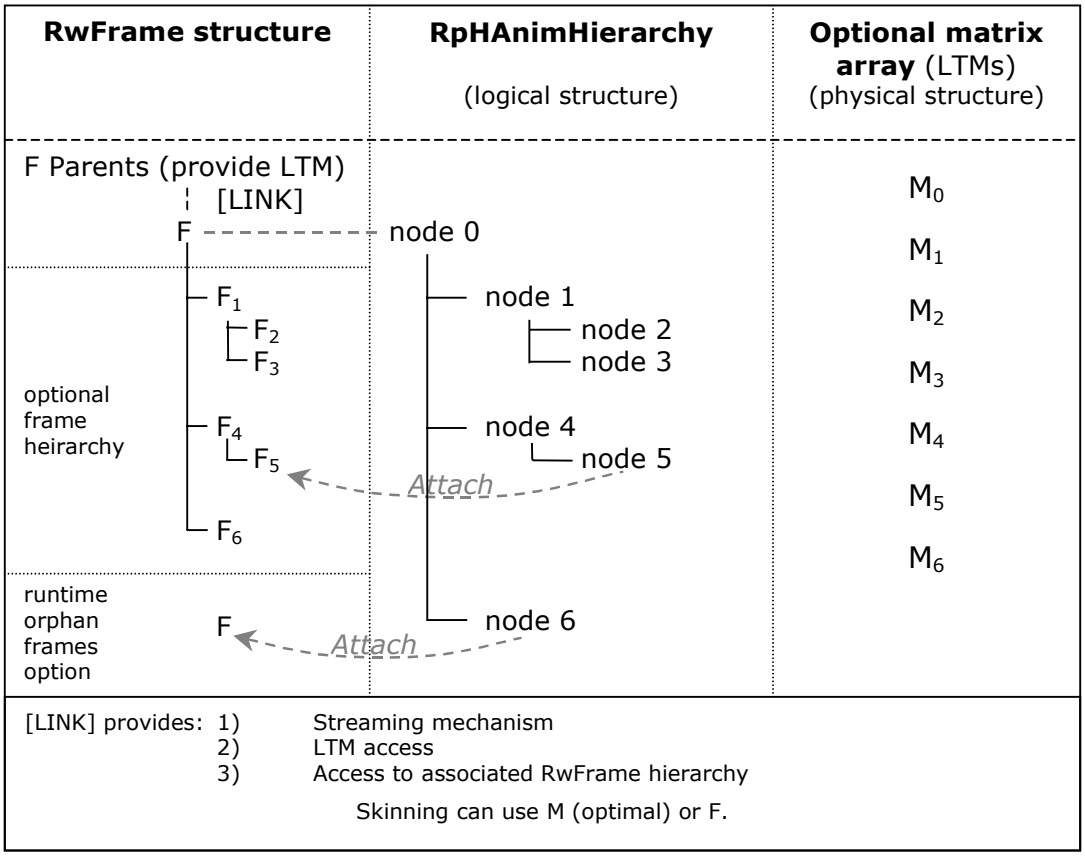
Attaching a hierarchy to an **RwFrame** not only allows it to be streamed out, but allows any parents of the **RwFrame** to effectively provide the LTM for the root of the hierarchy, so that the hierarchy objects can be moved around in a world as a whole.

Any number of **RtAnimAnimation** animations can be applied to a single node hierarchy, providing that the topology of the node hierarchy itself remains the same. This is because **RtAnimAnimation** animations are not explicitly linked to model data, but simply rely upon a matching hierarchical structure. So, as long as the model to which the animation is applied has a matching structure, the data will be valid.

Multiple animations are either applied sequentially or using various blending techniques described later.

18.2.1 Hierarchy Structure Overview

The **RpHAnimHierarchy** structure represents a number of inter-linked components. These include the topology of the hierarchy nodes, the connection to a **RwFrame** for positioning in a world, the state of an animation and the connection to optional **RwFrame**s to allow rigid body animation. These connections are shown in the diagram below.



Hierarchy Overview

18.2.2 Creating A Hierarchy

In most situations, the hierarchical structure will be created at the model export stage. The key function for this purpose is **`RpHAnimHierarchyCreate()`**.

This function takes the following parameters:

- The number of nodes in the hierarchy
- An array of node topology flags
- An array of node IDs
- Hierarchy creation flags
- The maximum keyframe size allowed in the hierarchy

These parameters are explained below.

Number of Nodes

This value is extracted from the modeler when using the RenderWare Graphics exporter. It is used as the array size for the arrays of per-node data such as topology flags and IDs.

Node Topology Flags

The node topology flags define the hierarchical structure of the nodes within a **RpHAnimHierarchy** object. They are traversed depth-first using a stack-based method.

Each node is therefore associated with a pair of flags representing a *push* state and a *pop* state. These flags can be either **TRUE** or **FALSE**, and the combination of flags is used to correctly traverse the hierarchy during animation:

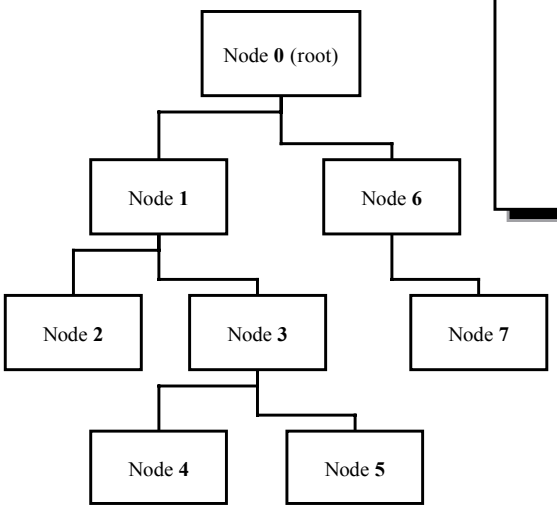
- A *push* flag is set for all nodes ***except*** those, which are considered the last sibling of a common parent. The root node is considered to be the last sibling at its own level and therefore no push flag is present.
- A *pop* flag is set for all leaf nodes only—i.e. those which do not have any child nodes.

The following table shows the possible combinations and their flags.

HAS CHILDREN	IS LAST SIBLING	FLAGS
FALSE	FALSE	PUSH|POP
FALSE	TRUE	POP
TRUE	FALSE	PUSH
TRUE	TRUE	NO FLAGS

These provide enough information at run-time to traverse the tree correctly.

The following diagram shows an example node hierarchy, with the node topology flags listed in the table alongside:



NODE	FLAGS
0	0
1	PUSH
2	PUSH/POP
3	0
4	PUSH/POP
5	POP
6	0
7	POP

Node IDs

Node IDs are usually generated automatically by the RenderWare Graphics exporter for the modeling package. An artist can, however, override the defaults and use their own IDs for particular nodes—a facility that makes it easier for programmers to locate any nodes that require special treatment, as would be the case for many procedural animation techniques.

The Node IDs must be unique within the hierarchy represented by a **RpHAnimHierarchy** object. Normally, a modeling package will supply unique node IDs, but in the event that they are not tagged, unique IDs will be created dynamically within the exporter by default.

Assigning Node IDs within a Modeling Package

Support for **RpHAnim** is available for both the 3ds max and Maya modeling packages. As each modeling package has a different user interface, package-specific details on tagging objects can be found within the appropriate Artist Guide for your preferred package.

Hierarchy Creation Flags

The **RpHAnimHierarchy** flags are split into two categories, creation only flags and general flags. The creation only flags should only be used when creating a hierarchy, these are:

- **rpHANIMHIERARCHYSUBHIERARCHY** (internal use only)
- **rpHANIMHIERARCHYNOMATRICES** causes the hierarchy to store no matrix array; this saves on memory if you only want to animate **RwFrame**s. A hierarchy setup with the **rpHANIMHIERARCHYNOMATRICES** flag can still be used to animated a skinned object. In this case the **RpSkin** plugin will extract the matrices from the **RwFrame**s, this will result in a slight performance hit compared to having a matrix array.

The general flags are:

- **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES**
- **rpHANIMHIERARCHYUPDATELTMS**
- **rpHANIMHIERARCHYLOCALSPACEMATRICES**

The first two flags describe which elements of **RwFrame**s should be updated during animation see section *18.3.2 Setting Up a Hierarchy For Use* for further details on the usage of these flags. The last flag defines whether the matrix array should calculate world space transform matrices or matrices local to the hierarchy root.

Maximum Keyframe Size

This value specifies the maximum size of a keyframe in the animation schemes to be used on this hierarchy. It is required so that efficient one block memory allocation can be made for the hierarchy.

Attaching to an RwFrame for Streaming

An **RpHAnimHierarchy** can be attached to an **RwFrame** to allow it to be streamed out into an **rwID_CLUMP** binary stream chunk. It should be attached to the **RwFrame** representing the first node in the hierarchy, use **RpHAnimFrameGetHierarchy()**. A **RwFrame** can only have one hierarchy.

18.2.3 Tagging RwFrames

RpHAnimFrameSetID, stores an **RwInt32** on each **RwFrame**. These IDs can be set to match up to the node IDs passed into **RpHAnimHierarchyCreate()** allowing the **RpHAnimHierarchy** to drive the update of **RwFrame**s at run time if required.

18.3 Using HAnim At Runtime

18.3.1 Finding a Hierarchy in a Model

Almost all the animation functions drive the state of an **RpHAnimHierachy**. The hierarchies will either come from an **rwID_CLUMP** chunk which has been loaded from a binary stream, procedurally setup as already described or will be created from an existing **RpHAnimHierarchy**.

The most common starting point will be from an **rwID_CLUMP** chunk exported by the RenderWare Graphics exporters into a binary stream. On these models the **RpHAnimHierarchy** will usually be attached to the first child **RwFrame** off the **RpClump**'s **RwFrame**. The safest way to get access to the hierarchy is to use **RwFrameForAllChildren()** calling **RpHAnimFrameGetHierarchy()** to find an attached hierarchy.

18.3.2 Setting Up a Hierarchy For Use

Flags

Using **RpHAnimHierarchySetFlags()** you can modify the way in which a hierarchy behaves during animation updates. These flags are a subset of those passed into **RpHAnimHierarchyCreate()**, and are the ones which can validly be changed at runtime.

rpHANIMHIERARCHYUPDATEMODELLINGMATRICES updates the modeling matrices of any **RwFrame**s attached to the **RpHAnimHierarchy**.

rpHANIMHIERARCHYUPDATELTMS updates the LTM of any **RwFrame**s attached to the **RpHAnimHierarchy.** This update is done in such a way that the standard **RwFrame** resynchronization will not occur unless further user changes are made.

The choice of which matrices in **RwFrame**s to update is determined by the application's use of the results, depending on the presence and the type of any modifications applied to the **RwFrame**s after animation.

When no modifications are made to a hierarchy between animation and usage of **RwFrame**s in rendering, the best performance will be obtained using **rpHANIMHIERARCHYUPDATELTMS** only.

If you are going to modify **RwFrame**s, forcing hierarchies to be resynchronized, it is important to update the modeling matrices. Otherwise the resynchronized hierarchies will contain the wrong LTMs.

If most of the nodes, or any root nodes are modified, only use **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES**, as most of the LTMs will need to be resynchronized anyway. But, if only a few subordinate **RwFrame**s (procedural inverse kinematics (IK) on arms/legs etc.) are updated, use both flags, so that only the updated **RwFrame**s will be resynchronized.

See section ***17.7 Procedural Animation*** for more details.

Attaching an object to the node of a RpHAnimHierarchy

If you want to attach child frames to frames attached to a **RpHAnimHierarchy** then you should either create the hierarchy with the **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES** flag set and the **rpHANIMHIERARCHYUPDATELTMS** flag not set or create the **RpHAnimHierarchy** with the **rpHANIMHIERARCHYUPDATELTMS** flag set and use **RwFrameUpdateObjects()** to force the resynchronization of the child LTMs.

rpHANIMHIERARCHYLOCALSPACEMATRICES makes the hierarchy matrix array updates happen in the local space to hierarchy root.

rpHANIMHIERARCHYLOCALSPACEMATRICES and RwFrame updates

When the local space matrices flag is applied to an **RpHAnimHierarchy** as well as the hierarchy matrix array being calculated in local space updates to **RwFrame**s will also be in local space.

The matrix array is generally used for the purposes of skinning using **RpSkin** and that plugin will deal with local space or non-local space matrices correctly. However **RwFrame**s are often used for rendering rigid bodies.

RwFrame Linking

In order to use the features of **RpHAnim** allowing **RwFrame**s to be updated the hierarchy must be attached to a set of **RwFrame**s. At creation time an array of node IDs should have been created that match up to the IDs stored on the **RwFrame**s using **RpHAnimFrameSetID**. Using these IDs **RpHAnim** can automatically set up pointers from the **RpHAnimHierarchy** to the **RwFrame**s and at update time will update the **RwFrame**s based upon the hierarchy flags.

The simplest functions to use are **RpHAnimHierarchyAttach()** and **RpHAnimHierarchyDetach()**. These functions will link up all matching nodes and **RwFrame**s between the hierarchy and the **RwFrame** it has been linked to (using **RpHAnimFrameSetHierarchy()**).

If you only require specific **RwFrame**s to be updated based on their nodes you can use **RpHAnimHierarchyAttachFrameIndex()** and **RpHAnimHierarchyDetachFrameIndex()**. These take an index into the array of per node data stored in a hierarchy and attach that node to its corresponding **RwFrame**. The **RwFrame** will be found by traversing the hierarchy of **RwFrame**s which the **RpHAnimHierarchy** has been linked to (using **RpHAnimFrameSetHierarchy()**)

The index into the per node arrays can be retrieved using the function **RpHAnimIDGetIndex()** which takes the ID of a node and returns the array index.

Using this index to access the per node data provides the ability to overload the **RwFrame** and hence rigid body animation linking. You can assign a completely individual **RwFrame** to a node by simply setting the per node data **RwFrame** pointer to point to the **RwFrame** you have created. The node information is in the structure member called **pNodeInfo** within the hierarchy. This is an array of the structure **RpHAnimNodeInfo** which contains an **RwFrame** pointer.

If you have used one of the attach functions e.g. **RpHAnimHierarchyAttach()** to attach the nodes to **RwFrame**s contained in the original **RpClump** you can also use this structure access to attach **RpAtomics**, **RpLights** etc. to the **RwFrame** which is being animated.

18.3.3 Concepts of Running Animations

Once you have an **RpHAnimHierarchy** and one or more **RtAnimAnimations** loaded you can run a simple animation by applying it and adding time. It is important to understand the main stages of updating the animation in order to optimize runtime performance. The **RpHAnimHierarchy** holds an array of output matrices and also a **RtAnimInterpolator** that holds 3 arrays of keyframe data.

The process of updating animation involves adding time and blending between different hierarchies, all of which update the start, end and interpolated keyframes. At the end of processing the calling **RpHAnimHierarchyUpdateMatrices()** will update the output matrix array and any attached **RwFrame**s based on the interpolated keyframes now stored in the **RpHAnimHierarchy**.

18.3.4 Applying and Running a Basic Animation

The first step is calling **RpHAnimHierarchySetCurrentAnim()**, this applies the animation to the hierarchy and initializes the hierarchy state and the attached interpolator state to zero time in the animation. At this stage the array of interpolated keyframes will be initialized to the first keyframe of the animation. By completing the update with **RpHAnimHierarchyUpdateMatrices()** and rendering the hierarchy this would result in the starting position of the animation.

It should be noted that all animations have an interpolation scheme associated with them (based on their type ID), and each scheme knows the size of its keyframe data. If you try to apply an animation to a hierarchy where the animation keyframe size is greater than the maximum keyframe size specified when the hierarchy was created then the assignment will fail. If the assignment succeeds it will also update the interpolation scheme functions that the hierarchy will use to run the animation.

To move forwards and backwards through the animation you call **RpHAnimHierarchyAddAnimTime()** or **RpHAnimHierarchySubAnimTime()** respectively. These calls ensure for each node that the pairs of start and end keyframes are set to the correct keyframes for interpolation and will then interpolate to the correct time. After these calls, the hierarchy will have a correct set of interpolated keyframes for the current time. These keyframes are then used to update the matrices with a call to **RpHAnimHierarchyUpdateMatrices()**. At this point rendering the scene will correctly render all skins/rigid objects bound to the hierarchy.

Another function allowing you to move through the animation is **RpHAnimHierarchySetCurrentAnimTime()**. This will involve an extra function call and merely calculates the offset and calls **RpHAnimHierarchyAddAnimTime()** or **RpHAnimHierarchySubAnimTime()**. Hence, if possible, calculate the offset and call the **RpHAnimHierarchyAddAnimTime()** / **RpHAnimHierarchySubAnimTime()** functions directly.

These animation functions mentioned are generic and will work regardless of the animation scheme used by the animation you are trying to play. The **RpHAnimKeyFrame** type which is supplied by default with **RpHAnim** also has optimized version of the add time functions called **RpHAnimHierarchyHAnimKeyFrameAddAnimTime()** respectively. These will only work if the animation type is **rpHANIMSTDKEYFRAMETYPEID**. Similar overloaded add/subtract functions could be written for custom keyframe schemes.

18.4 Features Inherited from RtAnim

As **HAnim** is a based on **RtAnim**, it inherits and expands features provided by **RtAnim**.

The attached **RpInterpolator** can be accessed through the **currentAnim** member of an **RpHAnimHierarchies**.

As the basic principles of the following features are already explained in **RtAnim**, only the **HAnim**'s specific points are included in this section. Please be sure to read **RtAnim**'s corresponding sections.

18.4.1 Blending Between Animations

The simplest example of blending between animations is a transfer from one animation to another where the end pose of animation 1 and start pose of animation 2 are not common. In this case you perform a blend to interpolate from a state in animation 1 to a state from animation 2.

To perform this blending you might need to create additional hierarchies, by calling **RpHAnimHierarchyCreateFromHierarchy()** which, given an input hierarchy, will generate a new hierarchy with a matching structure. You can, however, modify the maximum keyframe size during this creation so that you can create smaller hierarchies than the ones streamed in. This can save memory if you initially run complex interpolation schemes (say for cut-scenes) but then switch to a simpler system for in game animation.

There is an example of blending included in the RenderWare Graphics SDK. HAnim1 demonstrates blending from the end of one animation to the start of a second animation.

18.4.2 Sub Hierarchy Animations

Sub hierarchies appear and act just like standard hierarchies. However their matrix array is shared with that of their parent. Thus by applying animation first to the parent hierarchy and then the sub hierarchy you can run different animations on parts of the hierarchy. After animation the main hierarchy contains a copy of the entire state and can be used for skinning or rigid body rendering without any special behavior.

To create a sub hierarchy you need to know the array index of the root node of the sub hierarchy you want to create. Assuming you know the ID of the root node you can call **RpHAnimIDGetIndex()** to retrieve the index. Once you have the index call **RpHAnimHierarchyCreateSubHierarchy()** which takes the following parameters:

- **pParentHierarchy** – An **RpHAnimHierarchy** containing the branch you want to sub hierarchy.

- **startNode** – This is the array index in the parent hierarchy of the node you want to be the root of the sub hierarchy. This is used to calculate offsets into the parent hierarchy's structures.

- **flags** – These are the same flags as passed into **RpHAnimHierarchyCreate()** and allow you to have modified flags from the parent hierarchy. Thus allowing things like **RwFrame** updates from only sub hierarchies.

- **MaxKeyframeSize** – This allows you to set a different maximum keyframe size in the sub hierarchy from the parent hierarchy. Passing in –1 will use the same size as the parent.

The return value is an **RpHAnimHierarchy** pointer which will appear just like a normal hierarchy except that it will contain the **rpHANIMHIERARCHYSUBHIERARCHY** flag.

To use a sub hierarchy simply use it in the same way you would for a normal hierarchy, however ensure that the final **RpHAnimHierarchyUpdateMatrices()** call on the sub hierarchy happens ***after*** the main hierarchy. The fact that this is called last means that it overwrites any animation applied by the main hierarchy.

Sub hierarchies can be blended together like any standard hierarchy using the **RpHAnimHierarchyBlend()** assuming the topology of the sub hierarchies matches. However sub hierarchies can also be blended with a hierarchy with the same topology as their parent hierarchy (i.e. the one they were created from). To perform these blend operations use the function **RpHAnimHierarchyBlendSubHierarchy()**. This function allows a sub hierarchy and parent hierarchy to be blended together with the output hierarchy matching either the parent or sub-hierarchy. In the case where the hierarchy matches the parent all nodes present in the sub hierarchy will be blended into the output hierarchy, and all nodes present only in the parent hierarchy will be copied to the output hierarchy. Where the output hierarchy matches the topology of the sub hierarchy all nodes present in the sub hierarchy will be blended into the output hierarchy. This extended support means you do not need to have all the parent hierarchy animations duplicated as sub hierarchy animations. One example of this would be a case where you have a walk cycle animation for the parent hierarchy and want to blend it's state for the characters leg with a sub hierarchy animation representing just the leg kicking.

The **HAnimSub** example in the RenderWare Graphics SDK demonstrates the use of sub hierarchy animations.

Sub hierarchy systems will only function correctly if the parent hierarchy either has a matrix array (**rpHANIMHIERARCHYNOMATRICES** flag is not present) or if it is updating **RwFrame** modeling matrices. This is achieved by calling **RpHAnimHierarchyAttach()** and ensuring that the **rpHANIMHIERARCHYUPDATEMODELLINGMATRICES** flag is present.

18.4.3 Delta Animations

Delta animations work well when you want to add a number of small effects to a basic animation. It's necessary to take the deltas from a common pose in the animation to ensure they can be used together.

RpHAnimHierarchyUpdateMatrices() should never be necessary on hierarchies running delta animations unless you want to use a matrix representation of the deltas for some other purpose.

The **HAnim2** example in the RenderWare Graphics SDK demonstrates the use of delta animations with **RpHAnim**.

18.4.4 Overloaded Interpolation Schemes

In order to customize **HAnim** for specific types of animation, either higher order or optimized types, you can define new interpolation schemes.

The default interpolation scheme provided by **HAnim** is based on **RpHAnimKeyFrame** keyframe type, providing support for frame hierarchy animations.

```
struct RpHAnimKeyFrame
{
    RpHAnimKeyFrame  *prevFrame;
    RwReal           time;
    RtQuat           q;
    RwV3d            t;
};
```

prevFrame: Pointer to the previous keyframe for the current node, needed by **RtAnim** to run the animation.

time: Time of the keyframe, needed by **RtAnim** to run the animation.

q: Rotation, stored as a quaternion.

t: Translation, stored as a 3d vector.

When creating a new interpolation scheme for HAnim, ensure that you don't use an ID of 0x1 (defined as **rpHANIMSTDKEYFRAMETYPEID**) as it is the default interpolation scheme ID.

If your scheme overloads the **keyFrameApplyCB** as well, you'll be able to use **RpHAnimHierarchyUpdateMatrices()** to apply the animation to the **RwFrame** hierarchy.

An example of overloaded interpolation schemes is shown in the **HAnimKey** example. This shows a scheme which stores only rotations at each node and retrieves the base node offset from **RpSkin** data. This allows animation of character skeletons (which require no bone translations, assuming rigid bones) to be performed with large data savings. Under this scheme different scaled characters can also share animations.

18.5 Procedural Animation

Procedural animation in **HAnim** can be done at one of 4 different stages. In each case there are requirements relating to what stage of the animation update you're up to. The 4 stages are source animation data, interpolated keyframe data, matrix array data and post animating **RwFrame**s.

The modification of the animation data and the interpolated keyframe is covered in the **RtAnim** user guide.

18.5.1 Procedural Modification of the Matrix Array

To modify a hierarchy in the matrix array you must first call **RpHAnimUpdateHierarchyMatrices()** to ensure the matrices are up to date. The only real exception to this is if you wish to fill in the entire matrix array yourself. It's important to know that the matrices in the array are stored in one of two different spaces based on whether the **rpHANIMHIERARCHYLOCALSPACEMATRICES** flag is set. If the flag is set then the matrices are relative to the root node of the hierarchy whereas if not set then they are world space transforms for the nodes (equivalent to an **RwFrame**s LTM).

To modify the matrices call **RpHAnimHierarchyGetMatrixArray()** to get access to a **RwMatrix** array indexed based on the index returned by **RpHAnimIDGetIndex()**. These matrices should be treated just like normal **RwMatrix** objects and using the same RenderWare Graphics API functions.

The HAnim3 example included in the RenderWare Graphics SDK demonstrates applying node scaling procedurally in the matrix array.

The matrices stored in an **RpHAnimHierarchy** matrix array are only ever used by **RpSkin** to render skinned objects. If you also require rigid bound objects to be affected by procedural animation you must also update any attached **RwFrame** objects.

18.5.2 Procedural Modification of RwFrames

Any attached **RwFrame**s will be updated during a call to **RpHAnimHierarchyUpdateMatrices()**. This call will update either modeling matrices and/or LTMs based on the update flags stored in the hierarchy. If you don't attach all the **RwFrame**s in a hierarchy then updating modeling matrices will not function correctly since at the time when RenderWare Graphics resynchronizes the hierarchy old data from modeling matrices may be used to resynchronize the LTMs. If only LTMs are being updated then any procedural modifications need to avoid flagging **RwFrame**s as dirty to prevent resynchronization. This process is not generally advised unless you are certain it is required.

To update the frames as with other update methods get access to the **RwFrame**s through the **RpHAnimHierarchy** structures **pNodeInfo** array first using **RpHAnimIDGetIndex()** to get an array index into the structures.

These updates will affect all rigid bound atomics and also skinned atomics that are bound to an **RpHAnimHierarchy** with the **rpHANIMHIERARCHYNOMATRICES** flag set.

18.6 Compressed Keyframes

The toolkit **RtCmpKey** provides an alternative scheme for storing the keyframe data. **RtCmpKey** uses the structure, **RtCompressedKeyFrame**. This structure is similar to **RpHAnimKeyFrame** except the rotation and translation are stored in 16-bit fixed point format rather than in 32-bit floating point.

```
struct RtCompressedKeyFrame
{
    RtCompressedKeyFrame *prevFrame;
    RwReal               time;
    RwUInt16             qx;
    RwUInt16             qy;
    RwUInt16             qz;
    RwUInt16             qw;
    RwUInt16             tx;
    RwUInt16             ty;
    RwUInt16             tz;
};
```

prevFrame: Pointer to the previous keyframe for the current node, needed by **RtAnim** to run the animation.

time: Time of the keyframe, needed by **RtAnim** to run the animation.

qx, qy, qz, qw: Rotation, stored as a quaternion using 16-bit integers.

tx, ty, tz: Translation, stored using 16-bit integers.

The function, **RtCompressedKeyFrameCompressFloat(),** is provided to compress the 32-bit floating points values to 16-bit fixed points.

RtCmpKey's own animation callbacks must be used if **RtCompressedKeyFrame** is used in place **RpHAnimKeyFrame** during animation.

RtCmpKey provides a trade off between memory usage and performance. Using compressed keyframes will reduce memory usage but the penalty is the extra compression and decompression time.

18.7 Summary

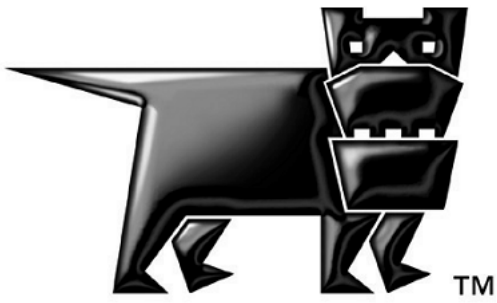
This chapter has dealt with running the **HAnim** animation system for animating both rigid bodies and skinned models.

It has dealt with the process of setting up and using animation data, and using the extended features of the **HAnim** plugin to achieve better results:

- How to create a hierarchy, by using the node flag to represent the topology, and the hierarchies flags to change their type and behavior.
- How to tag an ID to an **RwFrame** to ease the link between frames and hierarchy.
- How to load and play a hierarchical animation.
- The extended features described were:
- Blending multiple animations at runtime to add details and new effects.
- Creating and using sub-hierarchies to apply different animation to different part of a hierarchy.
- Creating and using delta animations to add effects and details to animations.
- Overloading the interpolation schemes to add higher order or optimized interpolation scheme.
- Procedurally modifying the animations, both by modifying the source data and modifying the state held with the **HAnim** hierarchical structures.
- Compressed keyframes can be used to reduce memory usage but will require compression and decompression time.

Chapter 19

The UV Animation Plugin



19.1 Introduction

UV coordinates may be used to describe the way a texture maps over the geometry of a 3D object.

When rendering, it's sometimes desirable to change the UV coordinates that were assigned by the artist when they textured the object in an art package. Effects, such as flowing textures over an object, could therefore be achieved.

Modifying individual UV coordinates is a costly operation; modifying entire groups of coordinates one by one would be particularly expensive.

RenderWare Graphics' **RpMatFX** plugin provides an efficient method of applying a change simultaneously to all UVs used by a texture rendering operation. This is done on a per-material basis.

The **RpUVAnim** plugin provides a convenient method for storing animations of these changes, and attaching those animations to materials.

The animations may be stored within dictionaries, which are streamed independently of the materials. The dictionaries are managed using RenderWare Graphics' **RtDict** toolkit.

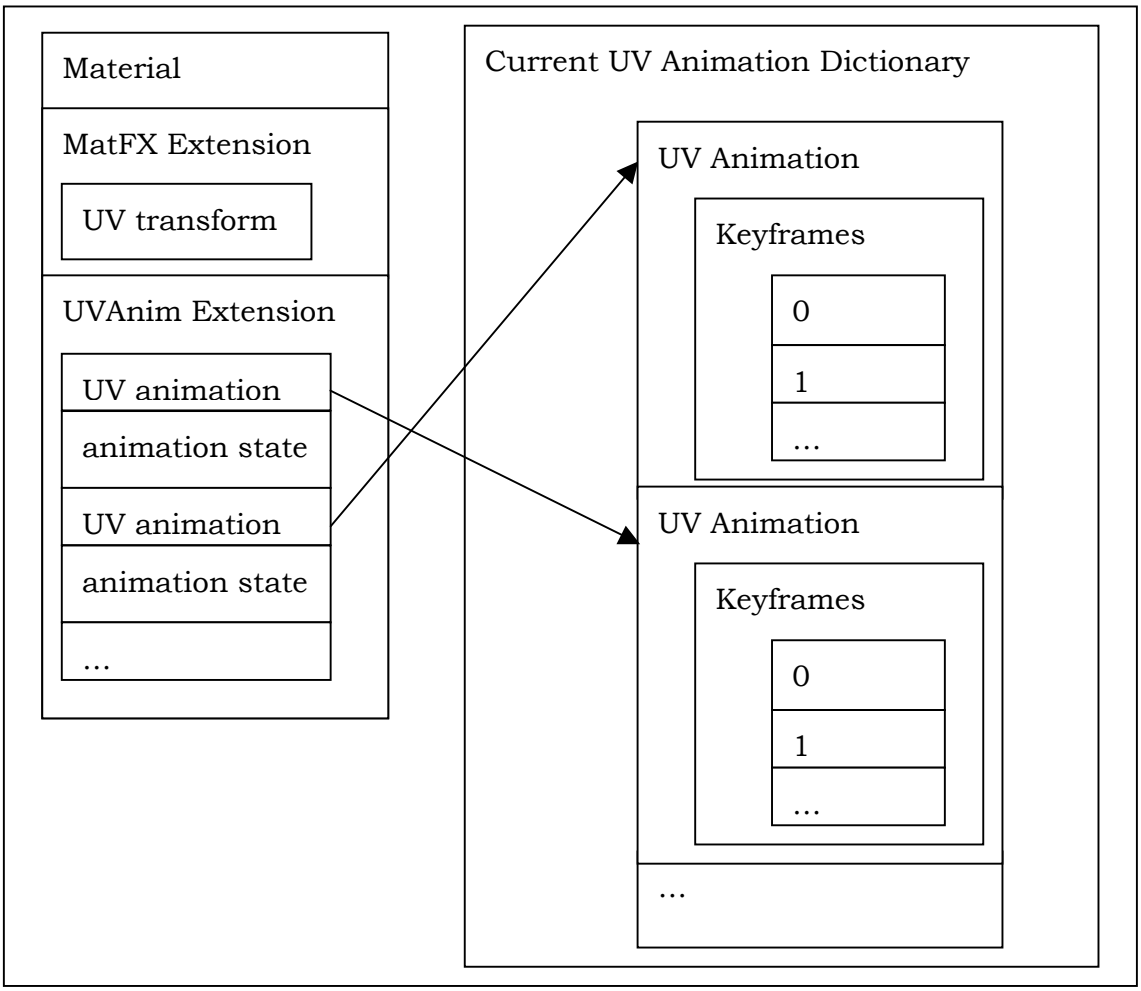
Streaming a material does not stream the attached UV animation; only a reference is read or written.

When a material with a UV animation reference is read in, **RpUVAnim** tries to match the reference against an animation in the current UV animation dictionary. It also stores the current state of the animation with each material.

Multiple animations may be attached to any one material and independently played.

RpUVAnim uses the generic dictionary library, **RtDict**, to store dictionaries of UV animations.

A simplified view of the situation is shown below.



UV Animation & Modeling Packages

Artists usually specify UV animations for a material during modeling. The RenderWare Graphics modeling package exporters support this process—detailed documentation can be found in the particular Artists Guide relevant to the modeling package.

UV animations created in the modeling packages can be previewed in the Visualizer.

19.1.1 This Document

This document describes basic and advanced usage of the UV animation plugin.

Section 19.2 details how to set up the plugin, and play back animations that were saved from the exporters.

Section 19.3 goes into more detail about the internals of the animations, describing how to set them up in code and directly apply them to materials.

19.1.2 Other Resources

API Reference:

- **RpUVAnim** plugin
- **RtAnim** toolkit
- **RtDict** toolkit
- **RpMatFX** plugin

- **uvanim** example
- The Animation Toolkit chapter of the User Guide
- The Dictionaries Toolkit chapter of the User Guide
- The Material Effects Plugin chapter of the User Guide, specifically the sections on applying single and dual pass UV transforms
- RenderWare Graphics' artists guides

19.2 Basic UV Animation Usage

The UV animation plugin extends **RpMaterial**s. It can store references to up to eight separate UV animations on each material.

The simplest way to create and use UV animations in RenderWare Graphics is via the 3dsmax and Maya exporters. The exporters support the creation and attachment of UV animations to materials.

The animations themselves are saved out to a separate dictionary. This dictionary must be loaded before the materials, that attach to the animations inside that dictionary, are loaded.

This section explains how to load and display UV-animated objects or scenes that were created with the exporters.

19.2.1 Attaching the Plugins

Before using any features of **RpUVAnim**, the world, material effects and UV animation plugins must be attached:

```
if (!RpWorldPluginAttach())
{
    return FALSE;
}
if (!RpMatFXPluginAttach())
{
    return FALSE;
}
if (!RpUVAnimPluginAttach())
{
    return FALSE;
}
```

19.2.2 Loading the UV Animation Dictionary

The exporters can save out the UV animation dictionary inside an **.rws** file with the objects that use it, or in a separate **.uva** file containing the dictionary on its own.

After opening a stream containing the dictionary, find and load the dictionary:

```
if (!RwStreamFindChunk(stream, rwID_UVANIMDICT, 0, 0))
{
    return FALSE;
}
RtDict *uvdict = RtDictSchemaStreamReadDict(
                    RpUVAnimGetDictSchema(),
                    stream);
```

Note the usage of **RpUVAnimGetDictSchema()** to access the schema for UV animation dictionaries.

After loading the dictionary, set it as the current UV animation dictionary. This enables **RpUVAnim** to link animation references on materials to the animations in the dictionary you have just loaded:

```
RtDictSchemaSetCurrentDict(RpUVAnimGetDictSchema(), uvdict);
```

Remember that on shutdown you will have to destroy the dictionary that you loaded earlier:

```
RtDictDestroy(uvDict);
```

Consult the **RtDict** documentation for more details on how to use dictionaries.

19.2.3 Loading the 3D Object

Any 3D object that uses materials (worlds, clumps, atomics) may have UV animation references set on it. These will automatically be loaded and linked to the current UV animation dictionary.

For example, a clump with UV animations on some materials is loaded exactly as you would any other clump:

```
RpClump *clump=RpClumpStreamRead(stream);
```

19.2.4 Obtaining a List of Materials to Animate

You will need to obtain a list of materials that have UV animations in order to animate them.

Functions that may be of use here are:

- **RpWorldForAllMaterials**
- **RpWorldForAllClumps**
- **RpClumpForAllAtomics**
- **RpAtomicGetGeometry**
- **RpGeometryForAllMaterials**
- **RpMaterialUVAnimExists**

The **uvanim** example demonstrates how to obtain a list of materials from clumps, atomics and worlds. This is fairly easy; but be careful not to include the same material twice in the list. You will need to call **RpMaterialUVAnimExists** to determine if the material has an animation on it, as the example assumes all materials should be animated.

19.2.5 Animating the Material

At this point, you should be able to access individual materials that have UV animations set on them. Making the material animate is then trivial.

Use **RpMaterialUVAnimAddAnimTime** to move the animation forward in time:

```
RpMaterialUVAnimAddAnimTime(material, deltaTime);
```

This doesn't modify the transform applied to the material. That is done with the **RpMaterialUVAnimApplyUpdate** function.

```
RpMaterialUVAnimApplyUpdate(material);
```

RpMaterialUVAnimApplyUpdate makes assumptions on how to combine multiple animations together, and how to animate both single and dual pass UV transform effects.

You could write and use your own function if your situation is more complex, for example when using multitexturing effects.

19.3 Creating and Applying UV Animations In Code

This section describes how a UV animation can be created and set on a material directly in code.

You might want to do this if you want more direct control of the animation creation and application.

The same setup and animation guidelines detailed in section 19.2 apply.

19.3.1 Creating a UV Animation

UV animations may be created with **RpUVAnimCreate**, which returns a pointer to an **RpUVAnim**:

```
RwUInt32 nodeIndexToUVChannelMap = {0, 1};
RpUVAnim *myAnim = RpUVAnimCreate(
                   "MyAnim", /* name */
                   2,        /* numNodes */
                   20,       /* numFrames */
                   10.0f,    /* duration */
                   nodeIndexToUVChannelMap,
                   rpUVANIMLINEARKEYFRAMES,
                             /* keyframeType */);
```

RpUVAnim is a typedef of **RtAnimAnimation**. It shares the same structure, but stores some custom data.

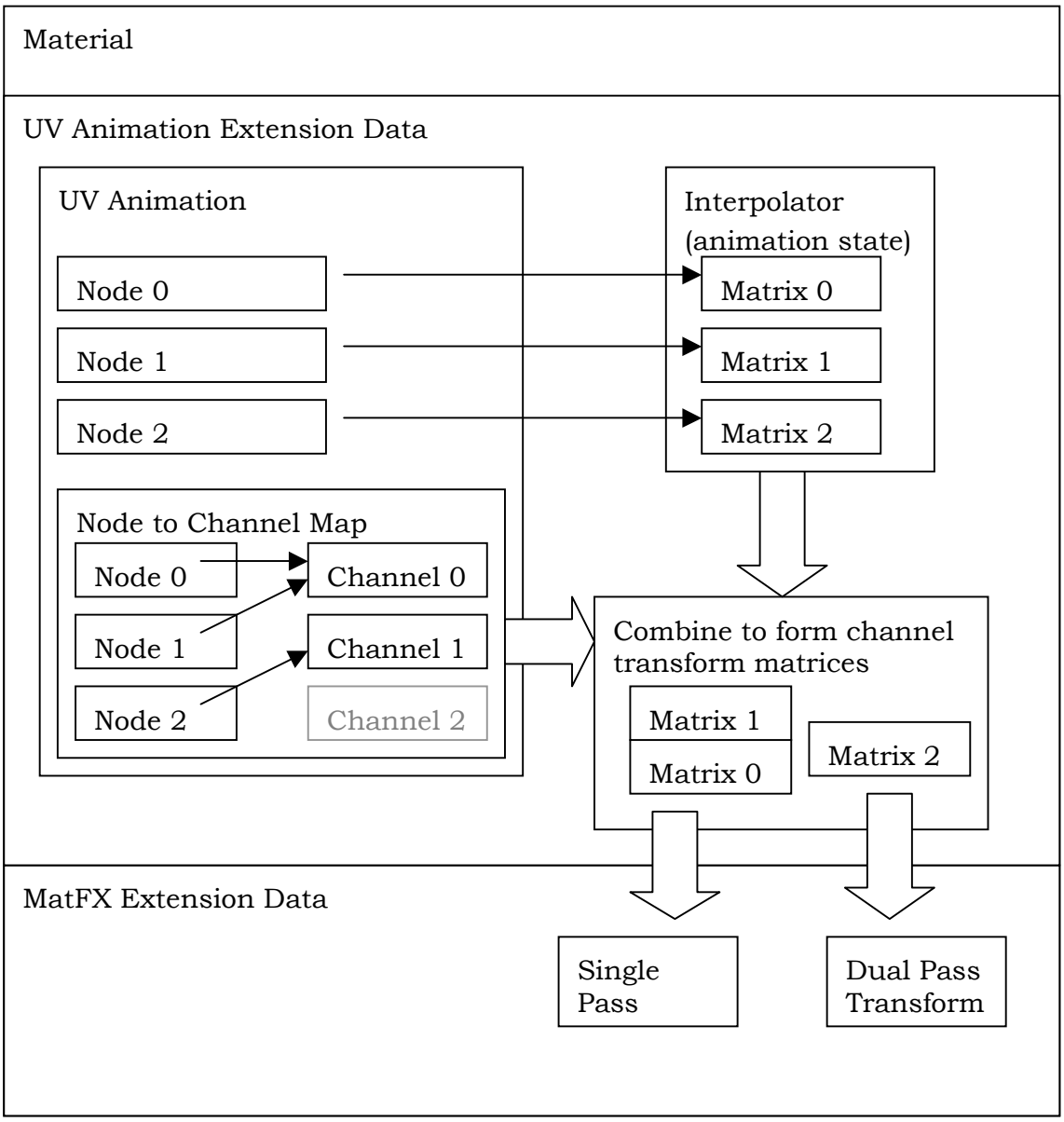
One of the custom pieces of data is a reference count. **RpUVAnim** is a reference counted type.

The parameters **numNodes**, **numFrames** and **duration** are the same as those specified for **RtAnimAnimationCreate**. Consult the **RtAnim** documentation for more details on initializing animations, animation nodes and keyframe management.

Each node of a UV animation controls a single output matrix. In the default implementation of **RpMaterialUVAnimApplyUpdate**, these matrices are combined, as determined by the **nodeIndexToUVChannelMap** stored in the animation. The combination is done by premultiplication of the interpolated matrices.

Regardless of how many output channels are specified in **nodeIndexToUVChannelMap**, only two output matrices are constructed by **RpMaterialUVAnimApplyUpdate**. These matrices are then copied to the single and dual pass UV transforms of the material.

The figure below details this process.



19.3.2 Setting up the Animation

After you've created the animation as, as described in section 19.3.1, you'll need to initialize the individual keyframes in the correct order. The **RtAnim** toolkit user guide chapter describes how to do this, and the **uvanim** example has sample code.

RpUVAnim provides a **RpUVAnimKeyFrameInit** utility function to assist you in setting up keyframes.

19.3.3 Managing the Lifetime of the Animation

Every animation you create needs to be destroyed, using **RpUVAnimDestroy**, when you are finished with it:

```
RpUVAnimDestroy(myAnim);
```

You could also assign your animation to a UV animation dictionary, and then destroy it. This transfers ownership of the animation to the dictionary; the animation will be finally destroyed when the dictionary is destroyed:

```
RtDict *dict = RtDictSchemaGetCurrentDict(RpUVAnimGetDictSchema());
RtDictAddEntry(dict, myAnim);
RpUVAnimDestroy(myAnim);
```

Since **RpUVAnim** is reference counted, a 'copy' of **myAnim** is now owned by the dictionary.

19.3.4 Using the Appropriate Effect on the Material

In order for the UV animation to actually affect the material it is placed on, that material must first have been set with the appropriate UV transformation effect. Either single or dual pass UV transformation effects may be applied:

```
RpMatFXMaterialSetEffects(material, rpMATFXEFFECTUVTRANSFORM);
```

or

```
RpMatFXMaterialSetEffects(material,
                          rpMATFXEFFECTDUALUVTRANSFORM);
```

The atomic or world sector that uses the material must also have effects enabled:

```
RpMatFXAtomicEnableEffects(atomic);
```

or

```
RpMatFXWorldSectorEnableEffects(sector);
```

19.3.5 Setting the UV Animation on the Material

The UV animation material extension has slots for up to eight UV animations per material.

An animation may be placed in a slot with the **RpMaterialSetUVAnim** function:

```
if (!RpMaterialSetUVAnim(material, anim, 0 /* slot */)
{
    return FALSE;
}
```

To update the single and dual pass channels, **RpMaterialUVAnimApplyUpdate** goes over the animation in each slot. It accumulates matrices for each channel as determined by the **nodeToChannelMap**, supplied on creation of the animation.

19.3.6 Accessing the Interpolators

RpUVAnim stores an interpolator for each animation slot in its extension to materials. This is used to store the interpolated state of the animation during playback.

If you require full control over the interpolators used to apply the animations, you can access them with the **RpMaterialUVAnimGetInterpolator** function. The **RpMaterialUVAnimSetInterpolator** function can be used to set the interpolators.

This could be useful if you want to share the same interpolator between multiple materials, possibly for efficiency reasons.

You don't need to manage the lifetime of interpolators you place in animation slots. **RpUVAnim** will destroy them for you when the material is destructed.

But, if you've applied the same interpolator to multiple materials, **RtAnimInterpolatorDestroy** will be called multiple times, as will **RpUVAnimDestroy**. This would be erroneous. So, in this case, you will need to destroy the interpolator yourself and reset the interpolator in the UV animation slots to NULL.

19.4 Summary

The UV animation plugin, **RpUVAnim**, provides a way of applying animations to the UV coordinates of a material.

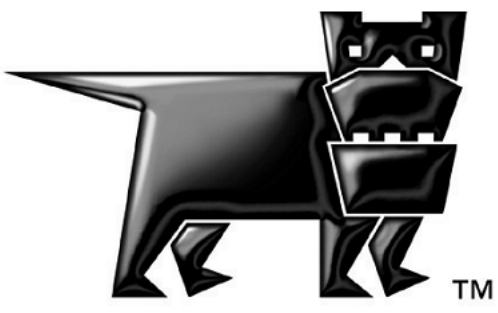
RpMaterial objects are extended with necessary data for supporting UV animations.

UV animations may be stored in dictionaries. Dictionaries of UV animations can be loaded. Materials can then be streamed in and locate UV animations in a preloaded dictionary. The UV animations can then be played back in an **RtAnimAnimation**–based manner.

UV animations can be also be created directly, and applied to materials.

Chapter 20

Morphing



20.1 Introduction

The chapter covers the data structures that describe morph sequences. The **RpMorph** plugin executes this data when it drives the morphing process. The chapter describes the morphing data held in the **RpAtomic** and **RpGeometry**, and the structures **RpMorphTarget**, **RpInterpolator**, and **RpMorphInterpolator**. It describes how to implement morphing, from preliminaries, to destruction, with some variations on morphing, and it discusses the "morph" example.

20.1.1 What Morphing Is

RenderWare Graphics supports animation in morphing, delta morphing, rigid body animation and skinning. Morphing is the simplest of these, and is implemented primarily in the **RpMorph** plugin. Morphing changes an object from one pre-defined shape to another pre-defined shape by simple linear interpolation. This is shown in the example, **RW\Graphics\examples\morph**, which shows the world stretching from a globe into one of three egg-shapes and back, in six consecutive morph interpolations.

Morphing works well for animating the objects being destroyed like racing cars crashing, objects being crushed or stretched like cushions or bean bags, objects being deformed like a ball when it bounces, or a simple cartoon character in movement.

The start and end states of morph interpolations are called "morph targets" in RenderWare Graphics, but they are sometimes known as "keyframes" so morphing is also referred to as "keyframe interpolation animation" or "keyframe animation".

20.1.2 What Morphing is Not

RenderWare Graphics supports other forms of animation distinct from morphing.

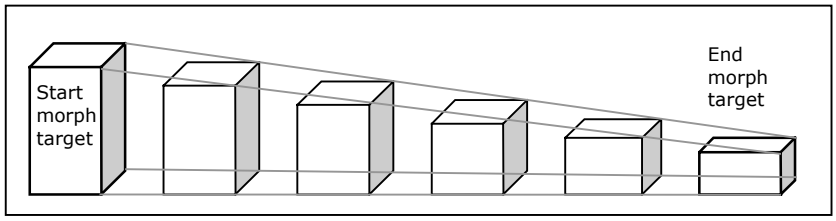
Rigid body animation, in **RpHAnim**, is different from morphing because it changes the relative positions of objects. Rigid body animation works efficiently when it animates a foot that moves relative to a shin, when the shin moves relative to a thigh which moves relative to a body. Morphing works well for objects that change their shape.

Skinning, in **RpSkin**, combines elements of rigid body animation and elements of morphing to represent the way that skin stretches over solid forms.

Delta morphing, in **RpDMorph**, supports changes between multiple shapes at the same time. It is used to change the expression of a sad face into a surprised face that is also sad. It uses one shape for the sad face, and changes it into the a blend of the shape for a surprised face and the shape for a sad face. Morphing is different from delta morphing in that morphing allows changes between no more than two states.

20.1.3 Basic Concepts

Two of the structures that support morphing are defined outside the **RpMorph** plugin. The **RpMorphTarget**s are part of **RpGeometry**, and **RpInterpolators** are part of the **RpAtomic**. Both belong to the **RpWorld** plugin. The array of **RpMorphInterpolator**s is added to the **RpGeometry** by the plugin.



Morphing is a linear interpolation between two shapes

In RenderWare Graphics, morphing requires a single starting state and a single end state at any given time. These are called "morph targets", as in the illustration above. Intermediate states are interpolated between them. The interpolation is always linear.

20.1.4 Strengths and Weaknesses

This means that morphing is not well suited for imitating the movement of the hands of a clock because they follow curved paths, and because the image of each hand rotates. Morphing can not represent either effect. The clock face animation could only be approximated by **RpMorph** in a number of small consecutive morphs between several positions and this would use up memory to store every position. Rigid body animation would be more appropriate.

Morphing can animate between single shapes; a face into a smiling face, or a face into a frown. The **RpDMorph** plugin was written to interpolate multiple shapes.

Morphing is not limited to simple transformations. Once the vertices of an object and its triangles are defined in a geometry, the form can be morphed into any other form defined by the same vertices and triangles. And a geometry can be morphed successively between a sequence of its morph targets to make up more complicated animation effects.

It should be noted that RenderWare Graphics Bézier patches are not compatible with morphing.

20.1.5 Other Documents

- See the API reference for details of the code for **RpMorph** and **RpWorld**.

- This chapter assumes you are familiar with the concepts of geometries, atomics, clumps and streaming from their descriptions in this User Guide. They can be found in the Chapters on *World and Static Models*, *Dynamic Models* and *Serialization*.

- Rigid body animation, delta morphing and skinning can be found in the Chapters on *The Hierarchical Animation Plugin*, on *Skinning*, and on *Delta Morphing* respectively.

20.2 Morphing Structures

Some of the basic data used for morphing is built into the **RpGeometry** and **RpAtomic** objects of the World plugin. The geometry may contain an array of morph targets, the static states, between which the morphs are interpolated. An atomic using such a geometry contains values that specify the current state of morph interpolation for a particular instance of the object.

The **RpMorph** plugin provides a morph animation system by extending the **RpGeometry** and **RpAtomic** objects, to specify a sequence of interpolations between specific morph targets, and the current position within such a sequence.

20.2.1 Geometry

Each **RpGeometry** stores the number of its morph targets. It may hold one or more. There must be at least two morph targets if we are to morph *between* them. So when a geometry specifies that it has only one morph target this indicates that it cannot support morphing.

The **RpMorph** plugin adds animation extension animation data to geometries, and the extension data holds a pointer to the geometry's morph targets. The morph targets are held in an array, so they are addressed by their index numbers. The morph targets can be used in any combination so that morphs can be defined between any two of them, and any morph target can be used in many morphs.

RpGeometry also has an array of **RpMorphInterpolator**s in its animation extension data (the appended data managed by the **RpMorph** plugin). As it is an array, the interpolators are addressed by their index numbers. Each **RpMorphInterpolator** defines an animation from one morph target to another in a specified number of seconds. The interpolators are linked in one or more sequences that describe an animation through a series of morph targets. Each interpolator points to the next one in the sequence.

20.2.2 Atomic

An **RpAtomic** holds three relevant data items:

- a pointer to its geometry, so it refers to its geometry's morph targets and the morph animation sequence.
- an **RpInterpolator** driven by **RpMorph** that is used to store the current interpolation state. The **RpMorph** plugin updates this as an animation progresses.

- the current position within the animation. (The animation is defined by a sequence of **RpMorphInterpolator**s in the geometry). Different atomics sharing a geometry may be at a different points in the animation.

20.2.3 Morph Targets

Morph target structures are part of a geometry. An **RpMorphTarget** is an opaque structure that contains:

- an array of vertices that determine the shape of the morph target. The order of the vertices corresponds to the order of the geometry's own vertex array. The colors, textures, triangle array and other data are held in the geometry and map to the same vertex order. The geometry can hold more shapes for its vertices by adding more morph targets to its array.

- a morph target may also have an array of normal vectors at each vertex, but this is optional.

- a morph target has a bounding sphere that encloses all the vertices in their current positions. This can be used for calculating the effects of light, for collision detection, for culling and for clip tests.

20.2.4 Interpolators

There are two types of interpolator used by the **RpMorph** plugin: **RpMorphInterpolator**, used by the geometry, and **RpInterpolator** used by the atomic.

RpInterpolator

The **RpInterpolator** is part of the atomic. It is opaque and holds the data for the atomic's current morph. Most importantly it stores the 'time' field, that represents the current point in the morph's duration between the two current morph targets. The **RpMorph** plugin updates the time field.

If the developer chooses to use the morph target data without **RpMorph**'s animation facilities then the **RpInterpolator** API Set and Get functions can be used to update an interpolator directly.

RpMorphInterpolator

The **RpMorphInterpolator** does not hold any current state information (as the **RpInterpolator** does) but instead it defines parameters for a single morph interpolation. The parameters will be copied across to the **RpInterpolator** at the appropriate time. So **RpMorphInterpolator** contains

- a pointer to its starting morph target

- a pointer to its end morph target
- and its duration in seconds (also called its "scale" or "time")
- a pointer to the next **RpMorphInterpolator** in sequence, or to itself if there is none.

An array of **RpMorphInterpolator**s is added to the geometry when the morph plugin is present. The array stores definitions of single morphs. It allows each interpolator to link on to another. The morph plugin reads this data automatically rendering a sequence of images from one morph to another.

20.3 How to Morph a Geometry

This section deals with preliminaries to implementing a morph interpolation. It describes how morphing data is normally imported from a modeling package, and how the timing is added using API functions. It describes how the interpolation data is made to move, and how movements can be varied and finally, how the extra data is destroyed.

20.3.1 Before Adding a Morph Animation

RpMorph relies on the **RpWorld** plugin being attached.

Morphing is one of the simplest forms of animation. It is not compatible with the more complicated delta morphing or with skinning, and does not support Bézier patches.

The header file **rpmorph.h** must be appended to the **#include** list.

RpMorphPluginAttach() must be called to attach the **RpMorph** plugin.

20.3.2 How To Set Up Morph Data

Setting up Morph Data with a Modeling Package

RenderWare Graphics supports the *3ds max* or *Maya* modeling packages, and either of them can be used to create and preview a morph. The Artists Guides for the packages will explain how to export "**.rws**" files, that contain morphing clumps, using RenderWare Graphics' exporters. The exporters translate the data from the package for streaming into RenderWare Graphics.

This process replaces some of the developer's tasks. The function **RpClumpStreamRead()** loads the data and unpacks it into

- a clump
- its atomic
- the atomic's geometry
- the geometry's morph targets
- and the geometry's morph interpolators.

This is how it is done in the "Morph" example:

```
if( stream )
{
    RpClump *clump = NULL;

    if( RwStreamFindChunk(stream, rwID_CLUMP, NULL, NULL) )
        clump = RpClumpStreamRead(stream);

    RwStreamClose(stream, NULL);
```

This loads all the data from the clump down to the morph targets. Note that the example streams from the legacy **.dff** file type that contains a single clump.

The meaning of "legacy" file types is that in the future, RenderWare Graphics exporters *may* not export to these file types. However, the binary format of these files continue to be supported, and RenderWare Graphics 3.5 and 3.6 will continue to read them. There is no need to re-export existing DFF/BSP/etc. artwork as RWS files.

Modifying the Data with the API

Once the exporter has translated the data from the graphics package into internal data structures, the objects are ready to be animated and rendered. The following functions are needed only if the developer wants to alter the behavior of the data from the graphics package.

Sometimes the application will need to replace the exported interpolations, to modify them or to create an animation programmatically. For instance, it might want to animate waves onto a sea using its own algorithm. These functions are exposed for this purpose.

RpMorphGeometryCreateInterpolators() reserves space for the number of interpolators required, destroying any interpolators already in place. The number of morph interpolators is passed as a parameter.

RpMorphGeometrySetInterpolator() is called once for each morph interpolator and fills or alters all the interpolator's data fields. It sets the interpolator's "next" pointer to the next interpolator in the sequence or to the first, if this is the last morph interpolator in the array.

RpMorphGeometrySetNextInterpolator() is provided to override the default behavior of **RpMorphGeometrySetInterpolator**(). It sets or alters the "next" pointer. The "next" pointer determines which morph interpolator in the array of morph interpolators will be executed after this one.

Intercepting the Interpolator Sequence

The callback will be used only in exceptional cases, but if it is needed, this is the point to set the callback function, **RpMorphGeometrySetCallBack()**. It is used to set the callback function that will be executed at the end of each interpolator to trigger some effect or variation.

20.3.3 Animating the Morph

RpMorphAtomicSetCurrentInterpolator() sets the animation to the start at the morph interpolator whose index is passed as a parameter. This will usually be zero. Then **RpMorphAtomicSetTime()** is called to set the time, usually zero, within the initial morph interpolator. Now the morph atomic is ready to be rendered.

When updating before rendering a new frame, the function **RpMorphAtomicAddTime()** is called to update the time field in the interpolator and advance to the next interpolator in the sequence when appropriate.

20.3.4 Effects and Variations

The **RpMorphInterpolator** structure is small, and the most efficient way to vary morph animations is to define them as new sequences of morph interpolators. For instance, it is generally more efficient to add an interpolator to reverse a simple morph than it is to swap the start and end morph targets and re-run it.

It is sometimes desirable for a morph to begin slowly, continue at speed and then slow down to its end. This can be done by representing the start and end of the sequence by slower morph interpolators that operate between intermediate morph targets.

In the section above, the geometry's callback function could be set. The callback function is called automatically when each interpolation is completed, and the default callback function simply moves on to the next morph interpolator. The developer can write an alternative callback function to detect a moment in a particular action and trigger an extra effect, and the **RpGeometrySetCallback()** function will select it.

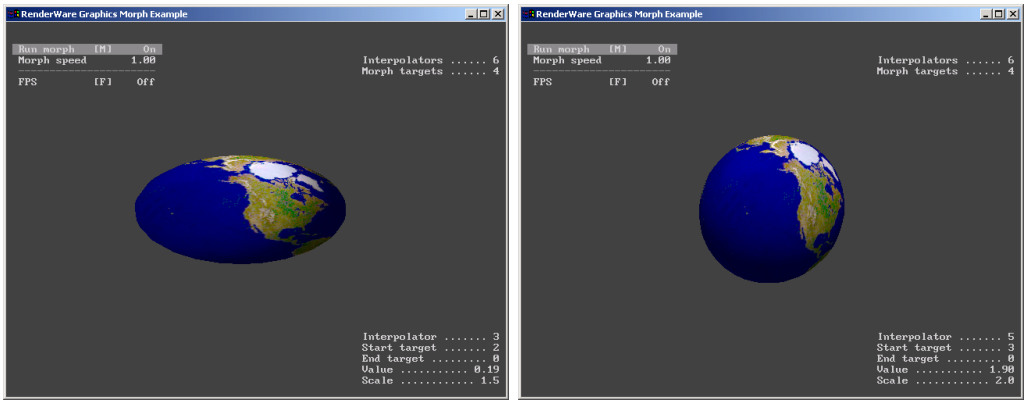
20.3.5 Destruction

If all the structures are set in place correctly, from the clump to the atomic, the geometry and their animation extensions holding morph targets and morph interpolators, then the **RpClumpDestroy** function will destroy all of them in the right order.

20.4 The Morph Example

The RenderWare Graphics example **RW\Graphics\examples\morph** shows a globe constantly changing shape between its default shape and one of three transformations of itself. The transformations are used because they are easy to produce in example code, not because **RpMorph** is limited to morphing between transformations.

The Morph example reads in the file "**world.dff**" in the function **RpClumpStreamRead()** as described above. The data is read in from the file stream. It may also be helpful to note in the code, that the function **RpClumpForAllAtomics()** calls the callback function that updates the geometry with the state for the present frame.



Two screen shots of the Morph example

The up and down cursor keys on a PC target can be used to select the "Morph speed" item. In this state, dragging the mouse or analogue controller rotates the globe demonstrating that the morph is interpolating in 3D. This operation calls the function **ClumpRotate()**.

The "Morph targets" value at the top right of the window shows that there are four morph targets. The "Start target" and "End target" values at the bottom right during the animation will show that the morph targets are numbered 0 to 3. The morph target "0" is the most frequent. It is the default, spherical shape of the earth. All the interpolations morph to or from morph target 0.

The six morph "Interpolators" are numbered 0 to 5 in the field labeled "Interpolator". The globe stretches from its spherical shape (morph target 0) in interpolators 0, 2 and 4, when the "Start Target" is always "0". Conversely when the earth is contracting back to its spherical shape, the "End target" is "0" and the "Interpolator" is 1,3 or 5.

The duration value of each interpolator is shown as the "Scale". Each "Interpolator", 0-5, adopts its own duration each time it is executed.

The "Value" represents the point in time at which each successive interpolation has reached. It increments up to the current value of "Scale". Many of these values, especially this one, will be easier to read if the cursor keys are used to select "Morph speed" to slow down the time scale below "1.0".

20.5 Summary

The **RpMorph** plugin supports animations built up from linear interpolation between start and end states known as morph targets. Morph targets are variations on the shape of a geometry, and are defined as arrays of vertices that correspond directly with the vertex arrays of their respective geometries.

In an animation sequence, the transition between states is defined by an **RpMorphInterpolator** over a specified duration, and the vertices are recalculated internally. Each interpolator is stored in an array in a geometry and the interpolation is linear.

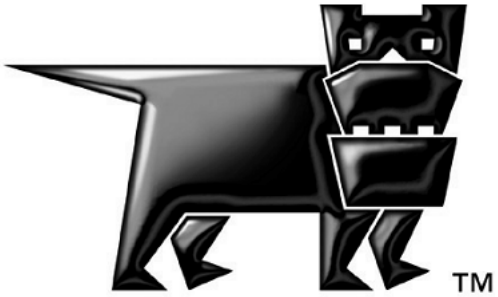
The geometry's **RpMorphInterpolator** array holds the data for one or more interpolation sequences. Each sequence can consist of a single interpolation or a series of consecutive interpolations in which the same morph target that ends one interpolation also begins the next one.

The morphing process implemented by **RpMorph** is integral to the **RpGeometry** and to RenderWare Graphics' rendering process.

Morphing is appropriate for linear changes. Rotation and curved paths would have to be approximated by joining sequences of morphs together and other forms of animation in RenderWare Graphics may be more appropriate.

Chapter 21

Delta Morphing



21.1 Introduction

This chapter describes Delta Morphing or "DMorphing" which is supported by the **RpDMorph** plugin and describes the facilities it provides. (Much of the chapter refers to dynamic models, **RpClump**s and **RpGeometry** objects, which are covered in the *Dynamic Models* chapter.)

In this section, DMorphing is introduced. In *Section 21.2* basic DMorph usage is covered and an example loading a pre-built model is presented. In *Section 21.3*, geometry (**RpGeometry**) and DMorph targets (**RpDMorphTarget**) are introduced. Animation (**RpDMorphAnimation**) is covered in *Section 21.4*, and the chapter is summarized in *Section 21.5*.

21.1.1 Morphing & Delta Morphing

Morphing is used to generate the intermediate frames needed to seamlessly morph one geometry to match another. For example, changing a facial expression from a frown to a smile. In use, the developer specifies starting and ending target objects and then functions are used to generate interpolated new geometry data from these two targets over time.

DMorphing differs in that there are a *number* of targets, which may be applied to a base geometry. In RenderWare Graphics, this can be used to generate combinations of **RpGeometry**. For example, changing a facial expression from a frown to a smile, with a hint of a grin. In the process, the base **RpGeometry** has one or more "delta morph targets" (**RpDMorphTarget**) (or "deltas" for short) applied to it. The **RpDMorphTarget**s can overlap and morph any combination of the base **RpGeometry**'s vertex components: positions, normals, prelight colors and texture coordinates.

21.1.2 DMorphing

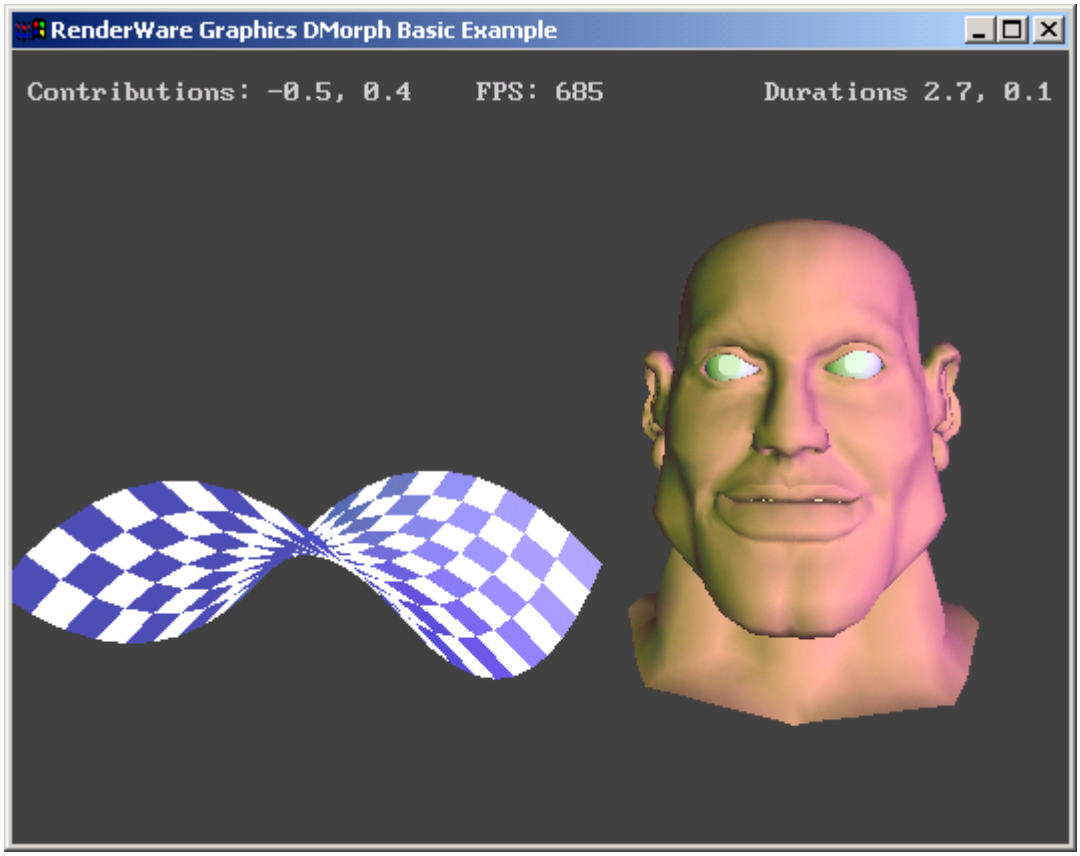
In RenderWare Graphics, DMorphing also differs from normal morphing in the way the targets are stored internally. The base **RpGeometry** and **RpDMorphTarget**s are created in the exporter or procedurally – where each is, initially, an absolute and complete model. The **RpDMorphTarget**s are stored as a plugin extension of the base **RpGeometry** data, and are *compressed* by excluding sequences of vertices where the delta is zero. As such it makes it much more efficient in terms of memory usage.

21.1.3 Animation

Although it is possible to directly manipulate the amounts that each **RpDMorphTarget** is applied to the base **RpGeometry**, a standard animation system is provided where a series of keyframes can be applied to each **RpDMorphTarget** and used to animate the system over time.

21.1.4 Examples

The example, found in **examples/dmorph**,, will be used in this chapter to illustrate the features provided by the DMorph plugin's API. The example uses two different **RpGeometry** objects to illustrate the two ways in which **RpDMorphTarget**s can be used – a human face which is animated, and a curved surface which can be manipulated – both have a base **RpGeometry** and two **RpDMorphTarget**s.



DMorph Example

21.2 Basic DMorph Usage

Before any of the plugin's features can be used, the plugin should be attached using **RpDMorphPluginAttach()**. Note, DMorphing is fully compatible with the skinning and matfx plugins, but it is *not* compatible with the (normal) morphing plugin.

In this section, we shall assume that we have a pre-built DMorph example, complete with **RpGeometry**, **RpDMorphTarget**s and an **RpDMorphAnimation** – such as the face in the example.

21.2.1 Loading a pre-built example

It is useful to realize that **RpGeometry** and its **RpDMorphTarget**s are *separate* from any animation (in fact, DMorphing can be achieved without any explicit animation).

New exports of DMorph models will be contained within an **.rws** file by default. This encapsulates the clump and DMorph animation legacy file types described below, each of which may be streamed using the RenderWare Graphics binary stream API.

The meaning of "legacy" file types is that in the future, the RenderWare Graphics exporters *may* not export to these file types. However, the binary format of these files continues to be supported, and this version of RenderWare Graphics will read them. There is no need to re-export existing DFF/BSP/etc. artwork as RWS files.

Geometry

A clump contains the base **RpGeometry** and all of its associated **RpDMorphTarget**s. **RpClumpStreamRead()** can be used to load this, after finding the **rwID_CLUMP** chunk header. The legacy **.dff** file type is used to store this clump.

Animation

The details of the frame are stored in the legacy file type **.dma** and can be loaded using **RpDMorphAnimationRead()** – internally, this opens an **RwStream** on the file, finds the DMorph animation chunk header (**rwID_DMORPHANIMATION**) and then calls **RpDMorphAnimationStreamRead()** before closing the **RwStream**. The animation is stored as an **RpDMorphAnimation**.

Before any DMorphing can take place, the **RpDMorphAnimation** has to be set onto the **RpAtomic** using **RpDMorphAtomicSetAnimation()** (note, this is only applicable where the **RpAtomic** has **RpGeometry** with **RpDMorphTarget**s attached). (**RpDMorphAtomicGetAnimation()** can be used to get the animation.)

In the DMorph example, the face is a pre-built example:

```
/*
 * Load DMorphing animation file...
 */

<BaseAnimation> =
    RpDMorphAnimationRead(RWSTRING("./models/face.dma"));
```

21.2.2 Animating

The **RpAtomic** has an animation interpolator which is set to the initial frame in the **RpDMorphAnimation** for each **RpDMorphTarget**, and the interpolation times are set to zero. The DMorph values are set to those at the start of the animation.

The animation can subsequently be advanced during run-time with **RpDMorphAtomicAddTime()**.

In the example, the face is animated over time. In addition to the basic DMorph usage described here, the animation settings of the face can be controlled by the user. (This is discussed further in *21.4 Animation*.)

```
/*
 * Update the animation of the atomic...
 */

RpDMorphAtomicAddTime(<atomic>, <deltaTime>);
```

21.3 RpGeometry and RpDMorphTargets

21.3.1 RpGeometry

An **RpGeometry** can be created in the exporter for involved or complex geometries, or procedurally for relatively simple or regular geometries such as cubes, spheres and surfaces, etc. The base **RpGeometry** is created as any normal **RpGeometry**, and is given one morph target using **RpGeometryGetMorphTarget()** (note the difference between “morph target” and “delta morph target” – remember DMorphing is not compatible with normal morphing). Each of the "to-be" deltas are also generated in this way – exactly as the base **RpGeometry**. However, only the base **RpGeometry** should be put in an **RpClump**.

For convenience, **RpDMorphTargetGetBoundSphere()** can be used to get the bounding sphere of the **RpDMorphTarget**. The bounding sphere is returned as if the **RpDMorphTarget** had been fully applied to the base **RpGeometry**.

In the example, the curved surface is generated procedurally. Three similar geometry objects are created all of which, necessarily, have the same number of vertices. The base **RpGeometry** is a flat surface, and the two **RpDMorphTarget**s are curved along one of the x and z axes.

21.3.2 Adding RpDMorphTargets

To attach the deltas, **RpDMorphGeometryCreateDMorphTargets()** is called with the base **RpGeometry** and creates space for a number of **RpDMorphTarget**s. Each target can then be added to the base **RpGeometry** using **RpDMorphGeometryAddDMorphTarget()**. With this function, the delta is assigned an **RpDMorphTarget** index and the vertex data which is to be morphed: vertices, normals, prelight colors and texture coordinates. These also have to be passed in the flag field, which can be logically or'd together; the flags are: **rpGEOMETRYPOSITIONS, rpGEOMETRYNORMALS, rpGEOMETRYPRELIT, rpGEOMETRYTEXTURED.**

To pass the vertex positions of an **RpGeometry** object, **RpMorphTargetGetVertices()** can be used on **RpGeometryGetMorphTarget()** for the **RpGeometry** object in question. (Likewise, similar functions exist to obtain the normals, prelight colors and texture coordinates.)

In the example, the surface has vertices and normals:

```
/*
 * Add DMorph targets to the base surface geometry...
 */
```

```
RwV3d *verts;
RwV3d *norms;

/* create space for 2 delta morph target */
RpDMorphGeometryCreateDMorphTargets(<BaseGeom>, 2);

verts = RpMorphTargetGetVertices
           (RpGeometryGetMorphTarget(<DeltaGeom>, 0));

norms = RpMorphTargetGetVertexNormals
           (RpGeometryGetMorphTarget(<DeltaGeom>, 0));

if (!RpDMorphGeometryAddDMorphTarget  (<BaseGeom>, 0,
        verts, norms, NULL, NULL,
        rpGEOMETRYPOSITIONS | rpGEOMETRYNORMALS))
{
    <Error>
}

. . .
```

The flags define which elements should be DMorphed and are queried using **RpDMorphTargetGetFlags()** on an **RpDMorphTarget**, which can be obtained from the base **RpGeometry** with **RpDMorphGeometryGetDMorphTarget().**

For convenience, **RpDMorphTarget**s can be named using **RpDMorphTargetSetName()** (or they can be named in the exporter). With the face example, labels of “rage” and “smile” could be used. An **RpDMorphTarget** name can then be retrieved later using **RpDMorphTargetGetName()**.

21.3.3 Saving DMorph RpGeometry

Once everything has been generated and attached, we can save the **RpGeometry** with its **RpDMorphTarget**s using **RpClumpStreamWrite()**.

21.3.4 Direct control of DMorph Values

Whether generated procedurally or pre-built, the number of **RpDMorphTarget**s of an **RpGeometry** can be obtained using **RpDMorphGeometryGetNumDMorphTargets()**. This is useful when used in conjunction with **RpDMorphAtomicGetDMorphValues()** for directly controlling the DMorph values. The latter function returns a pointer to an array of **RwReal**s that are the contributions that each **RpDMorphTarget** has on the base **RpGeometry** – and thus the values can be overwritten directly to alter the DMorphed **RpGeometry**. (It may also be done in combination with a standard DMorph animation, see *21.4 Animation*.)

The contributions that each **RpDMorphTarget** applies is normally in the range [0, 1], where a value of zero means that no contribution is being applied, while a value of one means that the entire contribution is being applied. However, the value may be set out of this range, including negative values.

Note, you need to call **RpDMorphAtomicInitialize()** to overload the **RpAtomic** so that it can be DMorphed – otherwise the object will remain rigid. However, this function is called automatically when DMorph enabled **RpGeometry** is loaded.

In the example, the DMorph values can be controlled directly by the user to alter the shape of the curved surface:

```
/*
 * Alters the surface contributions...
 */

RwInt32 max = RpDMorphGeometryGetNumDMorphTargets(
                  RpAtomicGetGeometry(<atomic>));
RwReal *dlist;
RwInt32 i;

dlist = RpDMorphAtomicGetDMorphValues(<atomic>);

for (i=0; i<max; i++)
{
    dlist[i] = <contrib_array>[i];
}
```

21.3.5 Transforming RpGeometry with RpDMorphTargets Attached

Transforming the **RwFrame** of the base **RpGeometry** will successfully transform the **RpGeometry** and its associated **RpDMorphTarget**s.

However, if **RpGeometryTransform()** is called (to transforms the **RpGeometry**'s **RpMorphTarget** vertices, see the *Dynamic Models* chapter), then **RpDMorphGeometryTransformDMorphTargets()** can be used to apply the specified transformation matrix equally to all **RpDMorphTarget**s defined on the base **RpGeometry** transforming both the vertex position deltas and vertex normal deltas.

21.3.6 Destroying RpDMorphTargets

An **RpDMorphTarget** can be removed from an **RpGeometry** object using **RpDMorphGeometryRemoveDMorphTarget()** (freeing up the data created with **RpDMorphCreateDMorphTargets()** and creating room for a new **RpDMorphTarget** to be added). Note, if the target was applying a contribution to the base geometry when it was removed, that influence will remain intact. To remedy this, the DMorph value should be directly set to zero before removal.

All **RpDMorphTarget**s can be destroyed with **RpDMorphGeometryDestroyDMorphTargets()**.

21.4 Animation

21.4.1 Creating Frames

An animation takes "delta morph animation" in the form of a series of frames that are associated with each **RpDMorphTarget**. An **RpDMorphTarget** can have an **RpDMorphAnimation** created with **RpDMorphAnimationCreate()**. For each animation one or more keyframe sequences can be created using **RpDMorphAnimationCreateFrames()**. The function must be called for every **RpDMorphTarget** that is to be controlled by the **RpDMorphAnimation**. (Some sequences may be left absent for **RpDMorphTarget**s which are unused or which are to be procedurally controlled externally.)

Each frame can then be set using **RpDMorphAnimationFrameSet()** whose arguments are:

Anim	A pointer to the **RpDMorphAnimation** object
DMorphTargetIndex	The index to identify the **RpDMorphTarget**
FrameIndex	The index to identify the frame
StartValue	The contribution of the delta to the base **RpGeometry** at the start of the frame
EndValue	The contribution of the delta to the base **RpGeometry** at the end of the frame
Duration	The duration of the frame in seconds
NextFrame	The index of the next frame

Alternatively, the last four arguments can be set individually with: **RpDMorphAnimationFrameSetStartValue()**, **RpDMorphAnimationFrameSetEndValue()**, **RpDMorphAnimationFrameSetDuration()** and **RpDMorphAnimationFrameSetNext()**, respectively. The values can be obtained using the "get" version of these functions.

Finally, the animation is set onto the **RpAtomic** using **RpDMorphAtomicSetAnimation()**.

In the example, the face has a number of frames per **RpDMorphTarget**. The user can increase or decrease the duration of the frames as a group, and can control each **RpDMorphTarget** independently:

```
/*
 * Change durations of the face animation...
 */

RwUInt32 frames;
```

```
RwUInt32 i;

frames = RpDMorphAnimationGetNumFrames(<BaseAnimation>, 0);
for (i=0; i<frames; i++)
{
    RpDMorphAnimationFrameSetDuration(
        FaceBaseAnimation, 0, i, <duration>);
}
```

21.4.2 Saving Animations

Once everything has been setup, an **RpDMorphAnimation** can be saved using **RpDMorphAnimationWrite()**. (This internally calls **RpDMorphAnimationStreamGetSize()** and **RpDMorphAnimationStreamWrite()**.)

21.4.3 Editing and Querying Frame Sequences

RpDMorphAtomicSetAnimFrame() will set the specified **RpDMorphTarget** to the start of a particular frame in an **RpDMorphAnimation**. (A value of **rpDMORPHNULLFRAME** may be specified for the frame index, which effectively disconnects a particular **DMorphTarget** from the **RpDMorphAnimation**.)

RpDMorphAtomicGetAnimFrameTime() can be used to get the interpolated time within the current animation frame of the specified **RpDMorphTarget** – zero at the start of the frame and equal to the frame duration at the end. Similarly, **RpDMorphAtomicSetAnimFrameTime()** is used to set the interpolation time within the current animation frame for a particular **RpDMorphTarget**.

RpDMorphAtomicGetAnimTime() is used to obtain the *total* amount of time added to the animation of a delta morph atomic. (It is impossible to set the absolute animation time directly – but this can be achieved using **RpDMorphAtomicSetAnimation()** and then adding the appropriate time with **RpDMorphAtomicAddTime()**.)

21.4.4 Loop CallBacks

RpDMorphAtomicSetAnimLoopCallBack() is used to set an **RpAtomic** callback that will be called whenever an **RpDMorphAnimation** loops during **RpDMorphAtomicAddTime()**. The function can be retrieved using **RpDMorphAtomicGetAnimLoopCallBack()**. (There is no default callback.)

21.4.5 Running an Animation

RpDMorphAtomicAddTime() is used to advance the animation of a DMorph **RpAtomic** by the given amount of time (an animation must have already been attached with **RpDMorphAtomicSetAnimation()**).

It is not possible to play an animation backwards, and adding negative time will produce invalid results. Note that if the animation loops, the time returned by this function does not reset to zero. It is the total time added to the animation including loops.

21.4.6 Destroying Frames

RpDMorphAnimationDestroyFrames() destroys the keyframe sequence in an **RpDMorphAnimation** corresponding to a particular **RpDMorphTarget**.

RpDMorphAnimationDestroy() destroys an **RpDMorphAnimation** and any keyframe sequences it contains.

21.5 Summary

21.5.1 Delta Morphing

- Delta morphing has a number of targets (**RpDMorphTarget**) that can be applied to a base geometry (**RpGeometry**).
- Each target contributes to the overall **RpGeometry**.
- Targets can overlap.
- It is possible to DMorph vertex positions, normals, prelight colors, and texture coordinates.

The example, **DMorph**, demonstrates basic usage, **RpGeometry** & **RpDMorphTarget** and **RpDMorphAnimation**.

21.5.2 Basic Usage

Delta morphing can be considered two-part. The **RpGeometry** and **RpDMorphTarget**s, and the **RpDMorphAnimation**.

With a pre-built model:

- It can be loaded with **RpClumpStreamRead()**.
- Its animation is loaded using **RpDMorphAnimationRead()**.

In general:

- **RpDMorphAtomicSetAnimation()** is used to set the **RpDMorphAnimation** onto the atomic.
- The **RpDMorphAnimation** can be run using **RpDMorphAtomicAddTime()** at run-time.

21.5.3 RpGeometry and RpDMorphTargets

For the **RpDMorphTarget**:

- **RpDMorphGeometryCreateDMorphTargets()** creates space for a number of **RpDMorphTargets**.
- **RpDMorphTargets** are added using **RpDMorphGeometryAddDMorphTarget()**.

DMorph values can be controlled directly using **RpDMorphAtomicGetDMorphValues()** which returns the array of contributions from each **RpDMorphTarget**.

21.5.4 RpDMorphAnimation

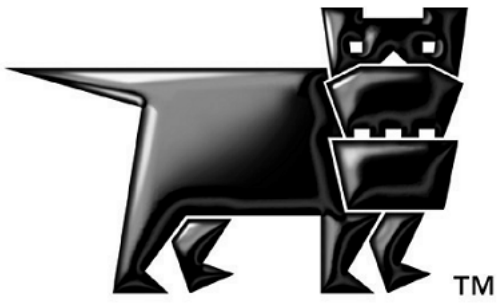
- An **RpDMorphAnimation** is created using **RpDMorphAnimationCreate()**.
- Frames can be added to each **RpDMorphAnimation** with **RpDMorphAnimationCreateFrames()**.
- Frames are set using **RpDMorphAnimationFrameSet()**.
- Finally, **RpDMorphAtomicSetAnimation()** is used to set-up the whole system.

Part D

Special Effects Libraries

Chapter 22

The Material Effects Plugin



22.1 Introduction

The Material Effects plugin, **RpMatFX**, provides a set of off-the-shelf material effects which can be applied to materials used by atomics and world sectors.

This chapter explains the features provided by this plugin and how to use them.

22.1.1 How RpMatFX Works

The **RpMatFX** plugin exposes a high-level API to a set of off-the-shelf effects pipelines. The high-level API hides the complexities of setting up the effects, which differs widely from platform to platform.

22.1.2 RpMatFX Features

The **RpMatFX** plugin supports four effects:

- Environment mapping
- Bump mapping
- Combined environment & bump mapping
- Dual-pass texture mapping
- Single pass with texture coordinate transformation
- Dual pass with texture coordinate transformation in each pass

These effects can be applied to a material used by either an atomic or world sector object. Different materials can have different effects enabled, so an atomic might, for example, be rendered with materials supporting both environment mapping and dual-pass texture mapping.

Material Effects & Modeling Packages

Artists usually specify the effect(s) required for a specific model during the export stage. The RenderWare Graphics modeling package exporters support this process—detailed documentation can be found in the particular Artists Guide relevant to the modeling package.

22.2 Using Material Effects

The Material Effects plugin works on **RpMaterial** objects rendered within atomics or world sectors.

The procedure for using **RpMatFX** is a four-step process:

1. Select the desired effect and apply it to the material
2. Initialize any additional data for the effect
3. Enable the **RpMatFX** renderer for the atomic or world sector which uses the material
4. Render the scene

The following sections will cover each step in more detail.

22.2.1 Selecting The Effect

The **RpMatFX** API exposes the **RpMatFXMaterialSetEffects()** function to select the required effect. This must be the first function called when setting up an effect on a material.

The **RpMatFXMaterialSetEffects()** function requires a pointer to the **RpMaterial** object that is to have the effect applied to it, as well as a *flag* which defines the effect to apply. The flag settings are listed in the table below:

FLAG	EFFECT
rpMATFXEFFECTBUMPMAP	Enables bump mapping on the chosen material.
rpMATFXEFFECTENVMAP	Enables environment mapping on the chosen material.
rpMATFXEFFECTBUMPENVMAP	Enables both bump and environment mapping on the chosen material.
rpMATFXEFFECTDUAL	Enables dual pass texturing
rpMATFXEFFECTUVTRANSFORM	Enables uv transformation
rpMATFXEFFECTDUALUVTRANSFORM	Enables dual uv transformation for two passes

22.2.2 Initializing Effect Data

Once an effect has been chosen, any additional data needed to implement that effect—additional texture maps, bump maps, etc.—need to be set up so the effect can be rendered.

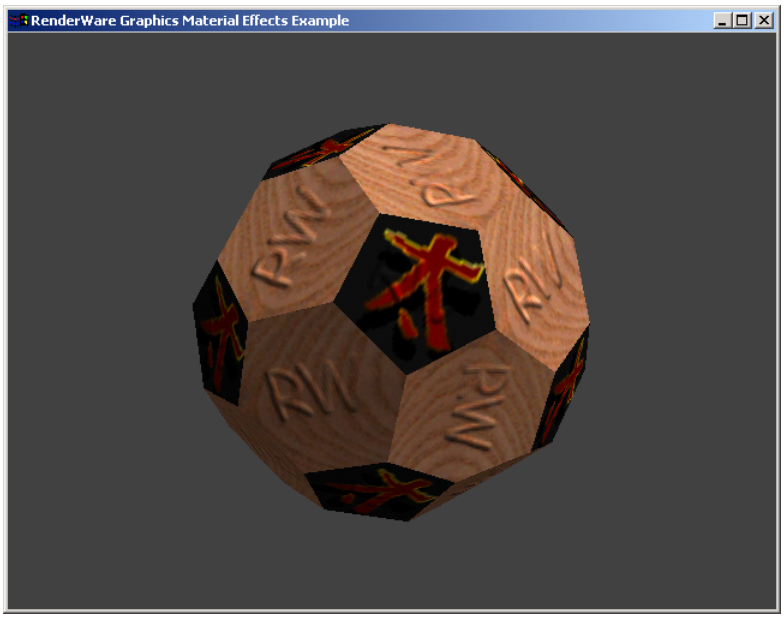
The initialization steps vary depending on the effect selected, so this section looks at each of the effects in turn.

The effects supported are:

- Bump mapping
- Environment mapping
- Bump & environment mapping
- Dual-pass texture mapping
- UV transformation
- Dual-pass UV transformations

Bump Mapping

The bump mapping effect renders a second bump texture over a base texture such that the illusion of a bumpy surface is created. An example is shown in the screenshot below.



A bump mapped model

Each bump mapped material requires the following properties to be initialized:

- A bump map texture defining the 'bumpiness'
- The definition of the light direction for the bump map
- The bump map coefficient, which defines the intensity of the bump mapping

The **RpMatFXMaterialSetupBumpMap()** function is used to set up the bump mapping effect's properties for each material in one call. In addition to this setup function, the properties also have individual access functions, which are useful if you need to modify a material effect property after it has already been initialized:

Setting Bump Mapping Properties

The functions described below are used to set the bump map properties for a material.

- **RpMatFXMaterialSetBumpMapTexture()** – sets the bump map texture;
- **RpMatFXMaterialSetBumpMapFrame()** – sets the bump map lighting direction, (represented by an **RwFrame** object). If this property is left undefined, the frame is derived from the current camera's *at* vector;
- **RpMatFXMaterialSetBumpMapCoefficient()** – sets the bump map's coefficient.

Retrieving Bump Mapping Properties

The functions described below are used to retrieve the bump map properties for a material.

- **RpMatFXMaterialGetBumpMapTexture()** – returns the material's current bump map texture;
- **RpMatFXMaterialGetBumpMapFrame()** – returns the material's current bump map lighting direction as an **RwFrame** object;
- **RpMatFXMaterialGetBumpMapCoefficient()** – returns the material's bump map coefficient.

Bumped Textures

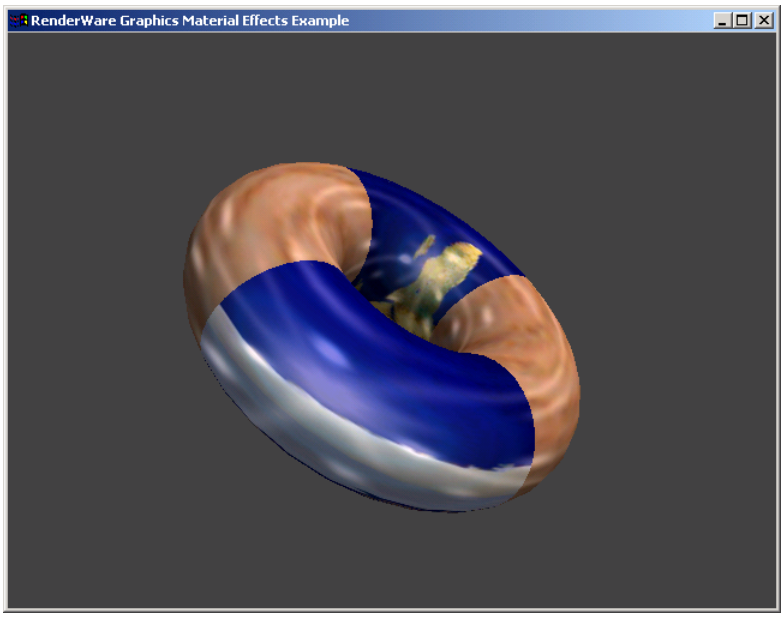
When a bump map texture is set, the **RpMatFX** plugin integrates it into the alpha channel of the base texture. The resulting, combined, texture is called a *bumped* texture and a pointer to it is obtained with a call to **RpMatFXMaterialGetBumpedTexture()**.

The two textures used for bump mapping are converted into one internal texture. To supply RenderWare with combined textures, you should store the intensity of the bump-texture in the alpha channel of the base texture, and save the texture with a name that is constructed from the interleaved characters of the base and the bump texture. For example, if you had a base texture called **base.png** and a bump-map called **bump.png** the combined textures would be called **bbausmep.png**. Put this texture in the texture path and it will be loaded and used. You will no longer need the two original textures.

Environment Mapping

The environment mapping effect creates the illusion of a reflective surface by mapping an environment texture—a texture containing an image of the material's surroundings—onto the material.

Variations on the effect can be created by using different images for the environment texture. For example, a texture that contains only highlights can be used to create the effect of a glossy surface.



An environment mapped object

Each environment mapped material requires the following properties to be initialized:

- The environment map texture
- An **RwFrame** which defines the environment map's projection. If this is not defined, a default frame object is generated which always faces the current camera
- A flag defining whether the frame buffer's alpha channel should be used when applying the environment map
- An environment map coefficient, which defines how reflective the material is, i.e. the intensity of the environment map

As with the bump mapping effect, each property also supports individual access functions.

Setting Environment Mapping Properties

The functions described below are used to set the environment map properties for a material.

- **RpMatFXMaterialSetEnvMapTexture()** – sets the environment map texture

- **RpMatFXMaterialSetEnvMapFrame()** – sets the environment mapping projection (an **RwFrame** object)

- **RpMatFXMaterialSetEnvMapFrameBufferAlpha()** – a boolean value, which should be set to **TRUE** if the frame buffer's alpha channel is to be used

- **RpMatFXMaterialSetEnvMapCoefficient()** – sets the environment map coefficient

Retrieving Environment Mapping Properties

The functions described below are used to retrieve the environment map properties for a material.

- **RpMatFXMaterialGetEnvMapTexture()** – returns the environment map texture

- **RpMatFXMaterialGetEnvMapFrame()** – returns the environment mapping projection

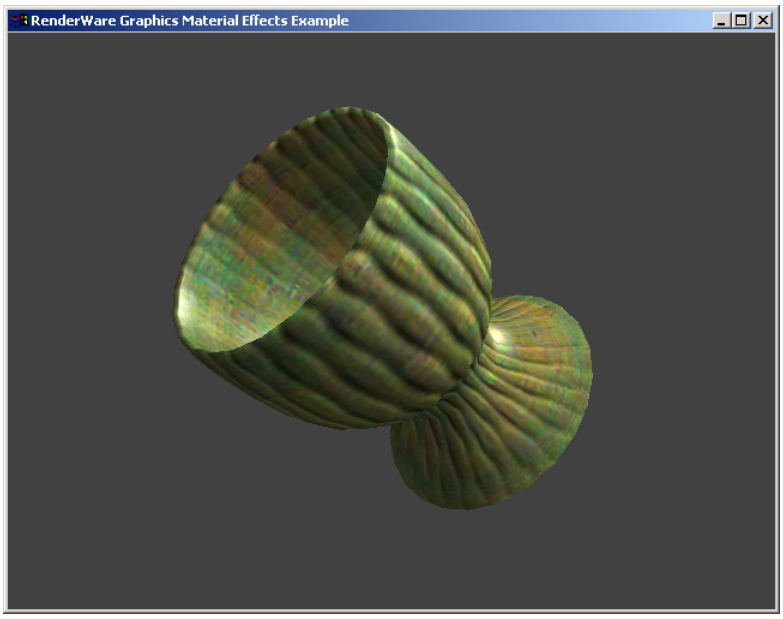
- **RpMatFXMaterialGetEnvMapFrameBufferAlpha()** – returns **TRUE** if the frame buffer's alpha channel will be used

- **RpMatFXMaterialGetEnvMapCoefficient()** – returns the environment map coefficient

Bump & Environment Mapping

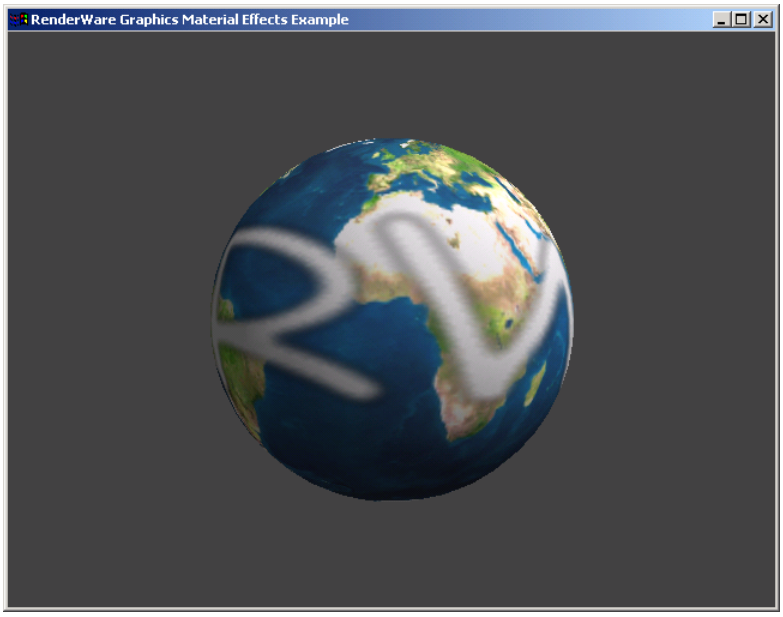
This effect combines both bump mapping and environment mapping effects, covered in detail above. This effect is achieved by combining both bump and environment mapping effects. The property setup process is performed by calling both the **`RpMatFXMaterialSetupBumpMap()`** and **`RpMatFXMaterialSetupEnvMap()`** functions.

The property access functions are also identical to those described earlier in the bump mapping and environment mapping sections.



An object with both environment and bump mapping

Dual-Pass Texture Mapping



An object with dual-pass texturing.
(The overlaid 'RW' text is the second texture.)

This effect works by combining the material's own texture with a second texture according to specified combination flags.

The setup function is **`RpMatFXMaterialSetupDualTexture()`** and sets the following properties:

- The second texture (the first is defined by the **`RpMaterial`** object)
- The blend function for the source data
- The blend function for the target data

Blend Functions

The two blend functions determine how the two textures are blended with the data in the frame buffer. They are defined by the **`RwBlendFunction`** enumeration. A brief description of each flag follows—see the *Blending* section in the *Immediate Modes* chapter for a complete description of the blending system.

- **`rwBLENDNABLEND`** – "Not A Blend" – no blending is performed
- **`rwBLENDZERO`** – RGBA channels set to zero
- **`rwBLENDONE`** – RGBA channels are set to 1
- **`rwBLENDSRCCOLOR`** – Source RGBA only
- **`rwBLENDINVSRCCOLOR`** – Inverse of source RGBA only

- **rwBLENDSRCALPHA** – Source alpha only on all channels
- **rwBLENDINVSRCALPHA** – Inverse source alpha only on all channels
- **rwBLENDDESTALPHA** – Destination alpha only on all channels
- **rwBLENDINVDESTALPHA** – Inverse destination alpha only on all channels
- **rwBLENDDESTCOLOR** – Destination RGBA values only
- **rwBLENDINVDESTCOLOR** – Inverse destination RGBA only
- **rwBLENDSRCALPHASAT** – Source alpha (saturated)

Each dual-pass effect property has its individual access functions.

Setting dual-pass effect properties

The functions described below are used to set the dual-pass effect properties for a material.

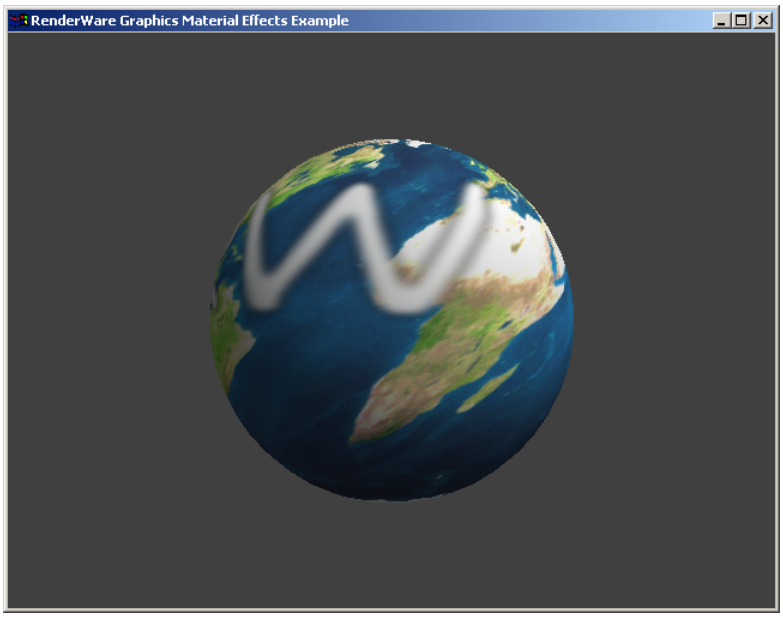
- **RpMatFXMaterialSetDualTexture()** – sets the second texture. The first texture is the one already contained within the **RpMaterial** object.
- **RpMatFXMaterialSetDualBlendModes()** – sets both blending functions.

Retrieving dual-pass effect properties

The functions described below are used to set the dual-pass effect properties for a material.

- **RpMatFXMaterialGetDualTexture()** – returns a pointer to the second texture.
- **RpMatFXMaterialGetDualBlendModes()** – returns the settings of the two blending functions.

Single and dual-pass with UV Transformation



An object with dual pass texturing and UV transformations.
(This is the same object as in the previous image with a shear transform applied to the first pass UVs, and a scale applied to the second pass UVs.)

These effects provide an efficient way to apply a transformation to the texture coordinates during rendering so that textures may be translated, rotated, stretched, scaled, or sheared. The application may animate the transformation by updating the transformation matrix each frame.

Once either of the effects **rpMATFXEFFECTUVTRANSFORM**, or **rpMATFXEFFECTDUALUVTRANSFORM** has been enabled, the UV transformation matrices may be set with the function **RpMatFXMaterialSetUVTransformMatrices()**. This takes two matrix pointers:

- The transformation matrix to be applied to the first pass UVs.
- The transformation matrix to be applied to the second pass UVs. (Only applicable when using the dual pass effect)

If either of the matrices are set to NULL, then an identity transform will be assumed. Otherwise, the matrices must be created by the application.

For the dual-pass uv transformations, the second pass texture and blending mode can be set as with the standard dual-pass effect, using **RpMatFXMaterialSetDualTexture()**, and **RpMatFXMaterialSetDualBlendModes()**.

Transform Matrices

Although the matrices are of type **RwMatrix**, the transformation is two dimensional, so only the subset of the matrix elements that affect X and Y coordinates are relevant:

$$\begin{pmatrix} R_x & R_y & 0 \\ U_x & U_y & 0 \\ 0 & 0 & 0 \\ P_x & P_y & 0 \end{pmatrix} \begin{matrix} \text{(right)} \\ \text{(up)} \\ \text{(at)} \\ \text{(pos)} \end{matrix}$$

The transformed texture coordinates are:

$$u' = R_x u + U_x v + P_x$$
$$v' = R_y u + U_y v + P_y$$

The matrix values may be initialized directly (remembering to apply the **`RwMatrixUpdate()`** function), or may be constructed using the **`RwMatrix`** functions for scaling, translating, and rotating matrices. Rotations should be about the z-axis.

Retrieving UV-transform effect properties

- **`RpMatFXMaterialGetUVTransformMatrices()`** – returns pointers to the UV transform matrices.

22.2.3 Enabling the Effects Renderer

Once the data for the effect has been initialized for a material, the **`RpMatFX`** renderer needs to be *enabled* for the atomic or world sector that contains the material.

If this operation is not performed, the additional effects data will be ignored and the material rendered using only the basic **`RpMaterial`** data.

Enabling the effects renderer on an atomic or world sector only needs to be performed once, regardless of how many **`RpMatFX`**-extended materials it contains.

Enabling Effects on an Atomic

To enable material effects for an atomic which contains an **`RpMatFX`**-extended material, call: **`RpMatFXAtomicEnableEffects()`**, passing the relevant atomic as a parameter.

Enabling Effects on a World Sector

To enable material effects for a world sector use: **`RpMatFXWorldSectorEnableEffects()`**, passing the relevant atomic as a parameter.

Rendering

Once the effects have been enabled on the relevant atomics and world sectors, rendering of the affected materials is performed automatically.

The Material Effects plugin's own renderer detects materials with **RpMatFX** data as they pass through RenderWare Graphics' rendering pipeline. When one is found, the plugin temporarily substitutes its own rendering pipeline to render these materials, reverting to the normal rendering pipeline for normal materials.

22.3 Examples

The following example shows how to set up an atomic containing a bump mapped material, then render it. Only relevant code fragments are shown.

A complete **RpMatFX** example is provided with the SDK's **examples** folder, named **matfx1**.

22.3.1 A Bump Mapping Example

Preparation

As was discussed in section 1.2.2, each bump mapped material requires the following properties to be set:

- A bump map texture defining the 'bumpiness';
- A definition of the light direction for the bump map;
- A bump map coefficient.

The **RpMatFXMaterialSetupBumpMap()** function is used to set the required data for each material.

A number of objects are involved in this example. The first is the atomic object which contains the model with the material which will contain our bump map.

```
RpAtomic *myAtomic;
```

Next, there's the material itself and the bump map texture which will be applied to it.

```
RpMaterial *myMaterial;
RwTexture *bumpTexture;
```

In addition, the bump map texture will also need a frame. This frame determines the direction of the bump map's lighting. This lighting gives us the effect of a bumpy surface, even though the surface is really flat.

```
RwFrame *bumpLighting; /* bump map lighting direction */
```

It should be noted that this light, which behaves similarly to directional lighting, is not a real light: it *only* affects the bump map on this particular material.

Other lights that impinge on this object will light the model as usual, but you will need to modify the bump map lighting to match in order to maintain the illusion.

Finally, we need a bump coefficient, which is defined as an **RwReal**, (defined here as 0.77, but it could be any value). This determines how bumpy the surface will appear to be. A low value gives only a slightly bumpy surface whereas a larger value produces a bumpier effect.

```
RwReal bumpCoefficient = 0.77; /* bump coefficient */
```

For the purpose of this example, these objects are assumed to have been already initialized with valid data.

Initializing the Material Effect

With the objects defined and initialized, the first step in the initialization process is to set the desired effect for the material.

```
RpMatFXMaterialSetEffects( myMaterial, rpMATFXEFFECTBUMPMAP );
```

This tags the material as a bump mapped material, but we still need to initialize the necessary data the bump map.

```
RpMatFXMaterialSetupBumpMap(    myMaterial,
                                bumpTexture,
                                bumpLighting,
                                bumpCoefficient );
```

Enabling Effects on the Atomic

For the purposes of this example, the **RpMaterial** object represented by **myMaterial** is assumed to be already contained within the atomic named **myAtomic**.

In order for the bump mapped material to render with the **RpMatFX** renderer, the atomic it is contained within *must* be enabled for material effects rendering:

```
RpMatFXAtomicEnableEffects( myAtomic );
```

Rendering the Effect

The Material Effects plugin hooks into the RenderWare Graphics rendering engine, so rendering of the atomic containing our bump mapped material can be performed using either:

```
RpAtomicRender( myAtomic );
```

or, if the atomic has been added to an **RpWorld** object, (named **myWorld**, in the example below) by a call to:

```
RpWorldRender( myWorld );
```

This completes the example.

22.4 Summary

The Material Effects plugin, **RpMatFX**, provides a set of off-the-shelf material effects which can be applied to materials used by atomics and world sectors.

22.4.1 Supported Effects

The **RpMatFX** plugin supports six effects:

- Environment mapping
- Bump mapping
- Environment & bump mapping
- Dual-pass texture mapping
- Single pass with texture coordinate transformation
- Dual pass with texture coordinate transformation

22.4.2 Extended Objects

Materials

The Material Effects plugin works by extending **RpMaterial** objects with necessary data for its supported effects.

The data must be set up prior to rendering, using the appropriate **RpMatFXEffectSetup...()** function.

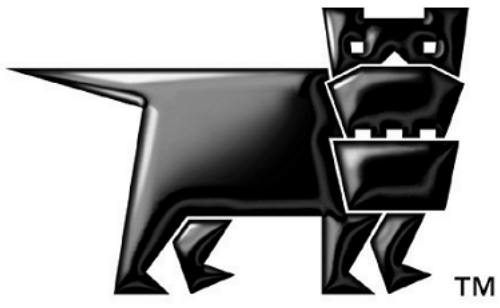
Atomics & World Sectors

RpMatFX hooks into **RpAtomic** and **RpWorldSector** object rendering, so a material *must* be used in either—or both—objects in order for it to be rendered.

The **RpMatFX** plugin will only hook into the renderer for an atomic or world sector by a call to either **RpMatFXAtomicEnableEffects()** or **RpMatFXWorldSectorEnableEffects()**, respectively. These functions *enable* the effects on the objects passed.

Chapter 23

Lightmaps



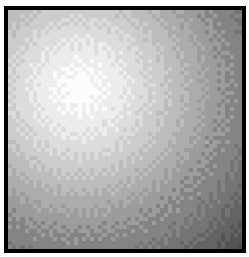
23.1 Introduction

This chapter describes the creation and use of lightmaps. These are **RwTexture**s, which are used to store pre-calculated lighting information - the brightness of static light incident on the static surfaces in a scene. This chapter covers their pros and cons, describing the process of creating lightmaps, importing, and of using them at run-time. The chapter provides concepts and examples of use, rather than detailed API specifications, which may be found in the API Reference. It provides a step-by-step guide to adding lightmaps to an application, describing the data objects involved. Much of this is done with reference to the Lightmaps example.

23.1.1 What are lightmaps?

Lightmaps are applied to static geometry (usually encoded as **RpWorldSector**s, though sometimes **RpGeometry**s) as a second texturing pass. While the base texture specifies the color-dependent reflectivity of geometry, the lightmap instead specifies the intensity of static light incident to the surface. The final, displayed color of a surface at a point is thus determined as the base texture color multiplied by the sum of the lightmap color and the interpolated dynamic vertex lighting color. This combination allows detailed, high-quality static lighting to be combined cheaply with lower-quality dynamic lighting.

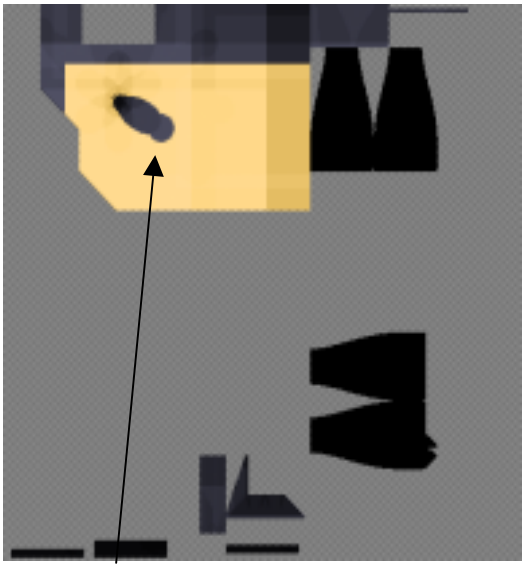
The base texture specifies diffuse (direction-independent) reflectivity and the lightmap similarly specifies a direction-independent sum of the light incident on a surface.



Part of a lightmap showing the distribution of light on a single wall, lit by one light source

For lightmapped geometry, given that it is rendered with two texture passes, two UV values are required per vertex. The second set of UV values is used to map each polygon in the scene to a unique area in one of the scene's lightmaps (a scene's static lighting may be stored in one or many lightmaps, depending upon a developer's wishes). Hence, any given texel in a lightmap is used only once in the scene (at only one sample point). They are not tiled, as base textures often are – in terms of memory usage. This is offset by the fact that, whereas base textures need to be of a fairly high resolution, light levels usually change gradually and hence lightmaps are usually of a significantly lower resolution.

A 'lumel' is one element of a lightmap. This is analogous to a 'texel' as an element of a texture, or a 'pixel' as an element of a monitor or TV screen. A lumel records, in one RGBA value, the color and intensity of the light incident at one sample point on a surface (note that the 'A' component of the RGBA value is ignored or used internally by the plugin).



A lightmap as generated by the Lightmaps example. (Notice the shadow of the vase - this area of the lightmap must map to the floor polygons beneath the vase.)

23.1.2 Why use lightmaps?

Lightmaps are used to reduce the processing required to render a scene at run-time. The basic lighting equations used to calculate lightmaps are no different from those used in the standard dynamic vertex lighting algorithms in RenderWare Graphics. Lightmaps do, however, provide some improvements to the quality of static lighting, beyond that feasible for dynamic vertex lighting. For instance, they sample lighting more uniformly and at a higher frequency across surfaces. Shadowing is also detected and anti-aliasing may be performed (as well as additional processes, if the lightmap illumination process is overloaded, such as the filtering of light through translucent objects).

Whilst lightmaps provide only static lighting information, they allow computationally expensive lighting calculations to be performed offline (at some point in the content-creation toolchain) such that only a few, dynamic light sources need be taken into account at run-time (with other techniques potentially providing dynamic shadows). The present speed of graphics processors and the quality of graphics that players now expect mean that it is for the most part not feasible to dynamically calculate high-quality lighting quickly enough for real-time games. This point is clearly illustrated by the Lightmaps example; it takes potentially several minutes for the lighting in a particular view to be calculated, yet once these calculations are complete, the user is able to navigate the same environment interactively.

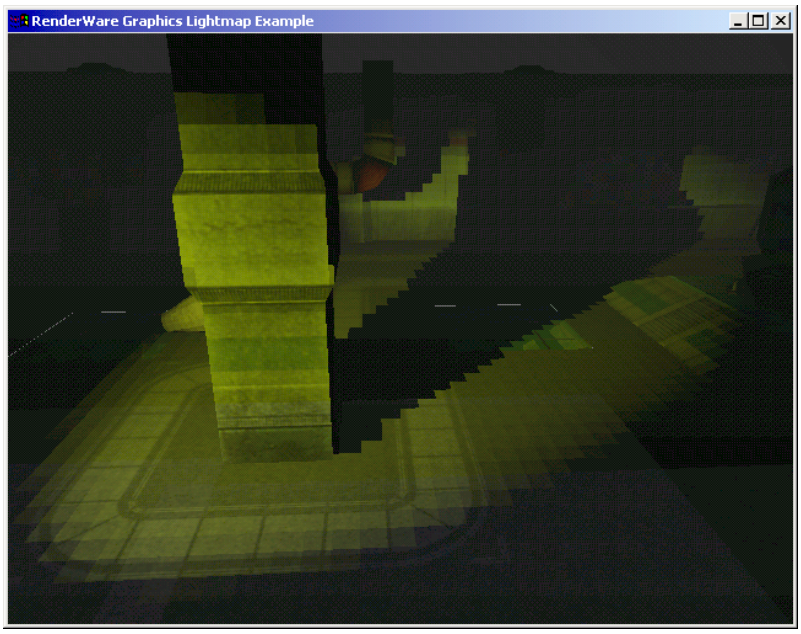
23.1.3 What are the costs of lightmaps?

Rendering geometry with lightmaps will naturally incur a fill-rate penalty, owing to the second pass of texturing (though this will generally be a small penalty on multitexture-enabled hardware).

Lightmaps can also occupy a significant amount of memory. On some systems, extra time may be required to transfer the lightmap to texture memory. The size of lightmaps, however, is controllable by the developer and lightmaps will in general be of a significantly lower resolution and bit-depth than base textures. The **`RtLtMap`** toolkit attempts to map static surfaces into lightmaps as efficiently as possible, to avoid wasting lightmap space.

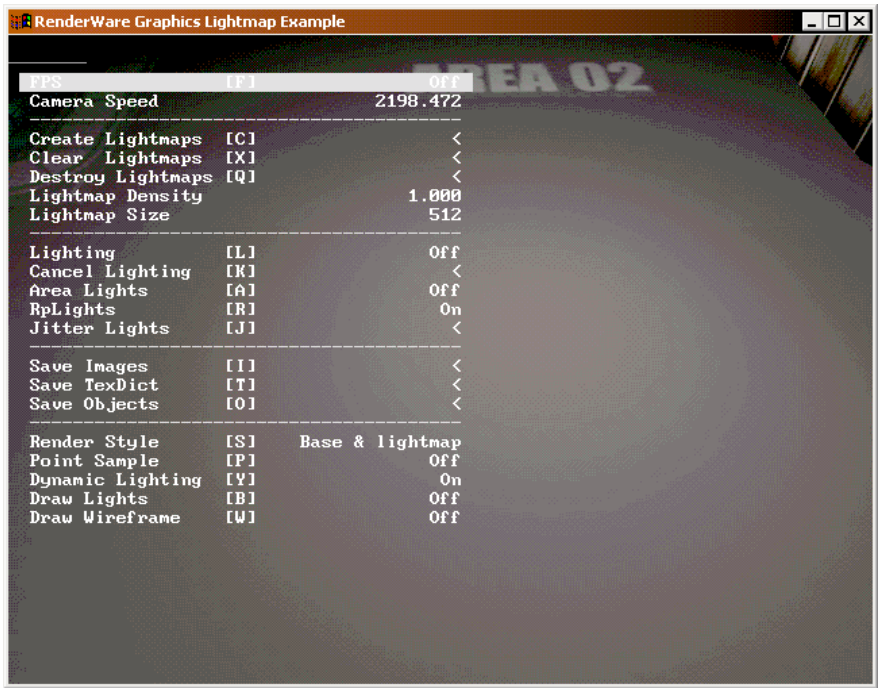
23.1.4 When not to use lightmaps?

Lightmaps encode static lighting information only, hence they are not of much use when a scene is lit entirely by dynamically varying light-sources. Lightmaps cannot represent the illumination from moving lights, the effects of light on moving objects, nor illuminations or shadows from moving objects. This does not, however preclude the combination of static lighting, encoded in lightmaps, and dynamic vertex lighting. The Lightmaps example illustrates the use of a moving light source: it applies the moving light over the pre-calculated light-mapped surfaces.



Lightmaps inevitably suffer from aliasing, being a discrete sampling of a continuous function

Lightmaps may produce ugly 'banding' artifacts when light values vary slowly across a surface that has a very smooth (or absent) base texture. For example, if a lamp hangs below a plain white ceiling, circular banding around the light will be quite obvious. At the opposite end of the spectrum, individual texels may be particularly clear in lightmaps with sharp color gradients (this is simply aliasing, the 'jaggies' familiar to all 3D graphics developers). Both of these artifacts may be countered by careful tweaking of base textures, lighting or lightmap resolution.



Lightmaps with low color gradients will generally cause visible banding artifacts when applied to plain surfaces.

23.1.5 Compatibility

The **RpLtMap** plugin renders lightmapped objects, and to do so it uses two texture passes during rasterization. It is not compatible with the **RpMatFX** plugin (so lightmapped objects may not use 'material effects' and material effects objects may not be lightmapped).

23.1.6 Other documents

- The API Reference provides technical details for the functions and data structures of the **RpLtMap** plugin, the **RtLtMap** toolkit and the **RtLtMapCnv** toolkit, as well as providing platform-specific information.

- The book "3D Games: Real-Time Rendering and Software Technology, Volume 1", by A. Watt and F. Policarpo, provides useful background reading on lightmapping and real-time 3D graphics programming in general.

- The pages listed below contain lightmap tutorials. A web search may yield further information.
  www.flipcode.com/tutorials/tut_lightmaps.shtml
  http://polygone.flipcode.com/tut_lightmap.htm
  www.delphi3d.net/articles/viewarticle.php?article=lightmapping.htm
  http://members.net-tech.com.au/alaneb/lightmapping_tutorial.html

23.2 Lightmap functionality overview

The RenderWare Graphics API divides the use of lightmaps into two stages; the creation of lightmaps and the use of lightmaps. Both stages use the **RpLtMap** plugin. The first stage uses the additional functions of the **RtLtMap** and **RtLtMapConv** toolkit.

The toolkit **RtLtMap** is used during lightmap creation, when:

- Lightmaps are allocated
- The static surfaces in a scene are mapped to areas in lightmaps
- The color and intensity of incident light for each texel is calculated and stored in the lightmaps

Alternatively, **RtLtMapCnv** can be used to import external lightmaps by,

- External lightmaps are generated by other packages and exported with the lightmapped objects.
- Internal lightmaps are allocated.
- The static surfaces in a scene are mapped to areas in internal lightmaps.
- The internal lightmaps are generated by converting the external lightmaps.

At run-time, the **RpLtMap** plugin provides functionality to:

- Load lightmaps from disk and associate them with the appropriate scene objects
- Apply the appropriate parts of the lightmaps as second-pass textures to their respective surfaces in the scene
- Extends world sectors, atomics and materials.
- The following sections in this chapter will cover the above three processes, splitting the descriptions into three phases:
- A description of the data objects involved in generating and using lightmaps
- A step-by-step guide to adding lightmaps to an application, covering generation, importing and use of lightmaps
- A review of the Lightmaps example (which illustrates all of the points covered in this chapter) and an introduction to the various options and possibilities for developers using lightmaps

23.3 Lightmap-related data objects

This section introduces the various data objects involved in the creation and use of lightmaps. It will also cover many API functions related to these objects (though some functions may instead be covered in the following section). The objects and functions that are described here are either pre-existing RenderWare Graphics objects or are defined by the **RtLtMap** toolkit. The **RpLtMap** plugin only uses pre-existing RenderWare Graphics objects.

Here is a summary list of the objects covered in this section:

- The **RtLtMapLightingSession** holds the data about a lightmap, and this data is necessarily used when creating a lightmap.
- The existing **RwTexture** object is used to encode lightmaps (so there is no actual **RtLtMapLightMap** object).
- **RpWorldSector**s are extended with plugin data to contain lightmaps and define lighting properties of each sector.
- **RpAtomic**s are extended with plugin data to contain lightmaps and define lighting properties of each atomic.
- **RpMaterial**s are extended with plugin data to define lighting properties for surfaces tagged with specific materials.
- The **RtLtMapAreaLightGroup** describes one or more area lights. This may be used, optionally, during lightmap illumination.

These are covered in order below, giving a summary of related functions.

23.3.1 Lighting Sessions

The **RtLtMapLightingSession** structure holds information used to manage the lighting of a scene. It specifies the objects to be lit, the lights used to light them and the methods used during lighting calculations. A lighting 'session' may be time-sliced such that lighting may be performed incrementally, and this structure will track lighting progress through a session.

The **RtLtMapLightingSession** structure is used widely in the **RtLtMap** toolkit. Here is a list of the functions that make use of it:

- **RtLtMapLightMapsCreate()**
- **RtLtMapLightMapsDestroy()**
- **RtLtMapIlluminate()**

- **RtLtMapImagesPurge()**
- **RtLtMapLightMapsClear()**
- **RtLtMapAreaLightGroupCreate()**
- **RtLtMapTexDictionaryCreate()**

The values in a **RtLtMapLightingSession** should be initialized by the function **RtLtMapLightingSessionInitialize()**. Only the scene's **RpWorld** must be specified before the **RtLtMapLightingSession** can be used. The members of the structure will now be listed, in three groups.

Scene specification

The following members are used to specify the scene (the objects that are to be lit and the lights that will illuminate them):

- **world**: a pointer to the world
- **camera**: a pointer to a camera (or **NULL**), the frustum of which determines which surfaces are to be illuminated
- **sectorList**: a pointer to an array of world sectors to be illuminated (or **NULL**)
- **numSectors**: the number of sectors in the array
- **atomicList**: a pointer to an array of atomics to be illuminated (or **NULL**)
- **numAtomics**: the number of atomics in the array

Progress tracking

The following members are used to track the progress of scene illumination, if it is performed incrementally, in 'slices' (performed via calls to **RpLtMapIlluminate()**):

- **totalObj**: the number of all the objects in the current scene (note that this value is automatically calculated when **RpLtMapIlluminate()** is called). An object being either a **RpWorldSector** or a **RpAtomic**
- **startObj**: the starting object which to begin the next illumination slice.
- **numObj**: the number of objects to illuminate during the next illumination slice.

Callback functions

The **RtLtMapLightingSession** holds addresses for three callback functions which, when non-**NULL**, may be used to override the default functionality invoked within **RtLtMapIlluminate()**:

- **sampleCallBack:** an **RtLtMapIlluminateSampleCallBack**. If **NULL**, **RtLtMapDefaultSampleCallBack** will be used. This callback performs lighting for groups of samples in objects being lit – see the API reference documentation for further details.

- **visCallBack**: an **RtLtMapIlluminateVisCallBack**. If **NULL**, **RtLtMapDefaultVisCallBack** will be used. This callback determines the visibility (zero, partial or full) between every light source and every sample in the scene - see the API reference documentation for further details.

- **progressCallBack**: an **RtLtMapIlluminateProgressCallBack**. If **NULL**, it will be ignored. This is called at five points during the illumination process, to provide feedback on progress to the user – see the API reference documentation for further details.

The Lightmaps example does *not* use custom callback functions.

23.3.2 Lightmaps

A 'lightmap' is just a **RwTexture**, where the value of each texel specifies the intensity and color of light incident at a sample point on the surface of an object in the scene. The function **RtLtMapLightMapsCreate()** is used to create lightmaps, for the objects specified by an **RtLtMapLightingSession** structure. Depending on the global or per-material (see the section on materials, below) lightmap density settings for these objects, a single lightmap may cover the surface of one or many objects. The function **RtLtMapLightMapsDestroy()** destroys the lightmaps attached to objects specified in an **RtLtMapLightingSession** structure.

For PlayStation2, lightmaps are specified slightly differently (they are in fact inverted 'darkmaps'). For further details, see the API reference documentation for **RtLtMapSkyLightMapMakeDarkMap()** and **RtLtMapSkyLightingSessionProcessBaseTextures()**.

The process that calculates the light that falls on each sample point in the scene is the **RtLtMapIlluminate()** function. This will be dealt with in further detail later on in this document.

Lightmap management functions

Lightmap textures are square and have a default side length given by the value **rpLTMAPDEFAULTLIGHTMAPSIZE**. This value may be overridden using the function **RtLtMapLightMapSetDefaultSize()**, the value passed to which will be used in the next call to **RpLtMapLightMapsCreate()**.

The names (and filenames, if saved to disk individually) of lightmap textures consist of a prefix and a counter, in the form "ltmp0000", "ltmp0001", "ltmp0002" and so on. The prefix and count may be accessed by the functions **RtLtMapSetDefaultPrefixString()**, **RtLtMapGetDefaultPrefixString()**, **RtLtMapSetLightMapCounter()** and **RtLtMapGetLightMapCounter()**. The name of a lightmap may be altered after it has been created, using the function **RwTextureSetName()**.

To clear lightmaps after they have been calculated, the function **RtLtMapLightMapsClear()** is provided. If its second parameter is **NULL**, it clears the lightmaps back to a black and white checkerboard pattern, whereas if it contains the address of an RGBA value, the lightmaps will be cleared to this color.

23.3.3 World Sectors

The **RpWorldSector** object is extended by the **RpLtMap** plugin, to contain lightmap-related data. Flags within this data, of type **RtLtMapObjectFlags**, may be accessed by through the functions **RtLtMapWorldSectorGetFlags()** and **RtLtMapWorldSectorSetFlags()**.

Here is a summary of the **RtLtMapObjectFlags**:

- **rtLTMAPOBJECTLIGHTMAP**: this object is to be lightmapped
- **rtLTMAPOBJECTVERTEXLIGHT**: this object's vertex prelight colors should be lit within **RtLtMapIlluminate()**
- **rtLTMAPOBJECTNOSHADOW**: this object does not cast shadows (probably so that dynamic shadows may be used later)

The default size of the lightmap created for an **RpWorldSector** is given by the value **rpLTMAPDEFAULTLIGHTMAPSIZE**. This default may be changed by the function **RtLtMapLightMapSetDefaultSize()**. The function **RtLtMapWorldSectorSetLightMapSize()** may be used to set lightmap size for an individual **RpWorldSector** (this should be used before **RtLtMapLightMapsCreate()** is called).

The function **RtLtMapWorldSectorGetNumSamples()** returns the number of sample points, corresponding to lightmap texels and vertex prelight colors (the world sector's flags determine which may be present), in the specified world sector.

RtLtMapWorldSectorLightMapClear() may be used to clear the lightmap for an **RpWorldSector** to the default black and white pattern, or to a specific color (note that this affects all the objects which use this lightmap). The lightmap may be destroyed with the function **RtLtMapWorldSectorLightMapDestroy()**, though its memory will only be released if it is not still in use by other objects.

23.3.4 Atomics

The **RpAtomic** object is extended by the **RpLtMap** plugin, to contain lightmap-related data. It contains the same extension data as does the **RpWorldSector** object and equivalent API functions are available for accessing this data in atomics (for example, use **RtLtMapAtomicGetFlags()**, instead of **RtLtMapWorldSectorGetFlags()**, to access an atomic's flags).

23.3.5 Materials

The **RpMaterial** object is extended by the **RpLtMap** plugin, to contain lightmap-related data. Flags within this data, of type **RtLtMapMaterialFlags**, may be accessed by through the functions **RtLtMapMaterialGetFlags()** and **RtLtMapMaterialSetFlags()**. These flags define how materials will interact with light in a scene.

Here is a summary of the **RtLtMapMaterialFlags**:

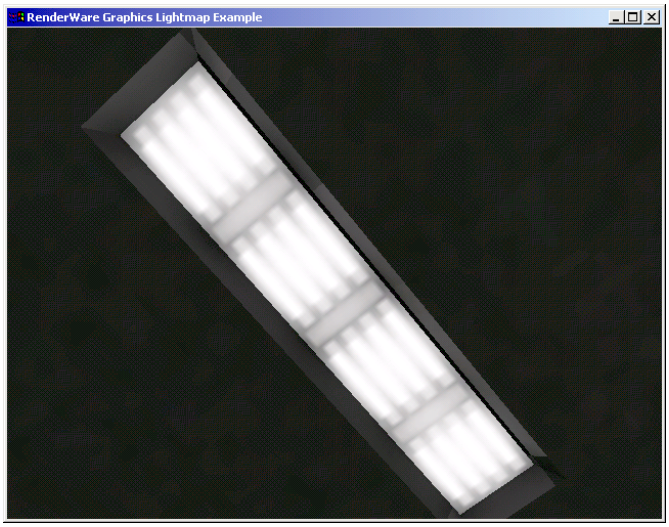
- **rtLTMAPMATERIALLIGHTMAP**: surfaces using this material should be lightmapped
- **rtLTMAPMATERIALAREALIGHT**: surfaces using this material emit light
- **rtLTMAPMATERIALNOSHADOW**: surfaces using this material do not block light (will not cast shadows)
- **rtLTMAPMATERIALSKY**: surfaces using this material block everything except directional light (such that light from the sun and sky may be represented as directional lights, even in the presence of 'sky polygons' enclosing the world)
- **rtLTMAPMATERIALFLATSHADE**: surfaces using this material will be flat shaded, using polygon normals rather than vertex normals
- **rtLTMAPMATERIALVERTEXLIGHT**: surfaces using this material will be lit at the vertices.

For further details, see the API reference documentation.

API functions used to modify material extension data will be described in the following section on area lights, to which all of these functions pertain.

23.3.6 Area Lights

An "area light" emits light from a two-dimensional area, as opposed to a "point light", which emits light from a zero-dimensional point. Area lights include fluorescent panels, the sky and lamp bulbs in diffusing shades. Point lights include candles, the sun and lamp bulbs without diffusing shades (assuming that each is viewed from far enough away, relative to their size!).



An Area Light *emits light from an area*

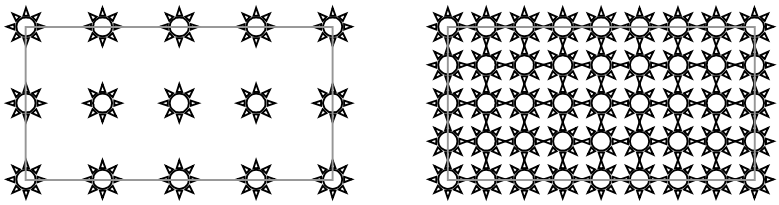
The function **RtLtMapAreaLightGroupCreate()** allocates memory for the **RtLtMapAreaLightGroup** structure, and fills it with data defining one or more area lights - these being identified by the flags (of type **RtLtMapMaterialFlags**) of materials used by the objects specified by the **RtLtMapLightingSession** structure passed to the function. Internally, area lights are represented as sets of point lights, at uniformly spaced sample points - these are very similar to standard **RpLight**s, though they only emit light from the surface's *front* side. The function **RtLtMapAreaLightGroupDestroy()** destroys the area light structure, releasing its memory.

The illumination from each sub-light within an area light decreases by the inverse square law over distance. Given this, there will be, for any area light, a distance, or radius, over which it is strong enough to *noticeably* illuminate other surfaces – this is the Region Of Influence (ROI) of the light. In the interests of efficiency, the **RtLtMap** toolkit will attempt to avoid applying the effects of an area light source to any surfaces outside its ROI.

The ROI of all area lights may be adjusted by a global modifier, which effectively brightens or dims all area lights. The functions **RtLtMapGetAreaLightRadiusModifier()** and **RtLtMapSetAreaLightRadiusModifier()** are used to read and write this global modifier value. Area light ROI may also be adjusted on a per-material basis (area lights are defined by the surfaces whose materials are flagged to emit light), with the functions **RtLtMapMaterialGetAreaLightRadiusModifier()** and **RtLtMapMaterialSetAreaLightRadiusModifier()**.

The RGBA value (alpha being ignored) of the light being emitted from area lights of a given material can be adjusted using the functions **RtLtMapMaterialGetAreaLightColor()** and **RtLtMapMaterialSetAreaLightColor()**. The magnitude of this color vector will affect an area light's ROI, though in order to obtain maximum precision in the specification of the light's hue, it is best to keep the magnitude large and to scale the light's brightness using **RtLtMapMaterialSetAreaLightRadiusModifier()**.

The ROI of an area light is calculated based upon the estimated visual error caused by ignoring the light's influence outside of this region. The 'cut-off' error value may be adjusted using the functions **RtLtMapGetAreaLightErrorCutoff()** and **RtLtMapSetAreaLightErrorCutoff()**.



Different densities of sub-lights within an area light

The world-space density of the point lights that represent area lights is passed as a parameter to **RtLtMapAreaLightGroupCreate()**. This may be multiplied by global or per-material modifiers, which may be adjusted using the functions **RtLtMapGetAreaLightDensityModifier()**, **RtLtMapSetAreaLightDensityModifier()**, **RtLtMapMaterialGetAreaLightDensityModifier()** and **RtLtMapMaterialSetAreaLightDensityModifier()**. The higher the density of sample points within an area light, the more accurate the resultant lighting will be (the smoother the soft shadows you will get), but the longer it will take to calculate the lighting solution.

The number of samples (sub-lights) within an area light affects *only* the quality of the resultant lighting solution, it does *not* affect the brightness of the area light source – so whilst there are more sub-lights constituting the area light, each one is commensurately dimmer.

The larger an area light is, the more sample points it will have on its surface; the increase is a function of the area of the light (so doubling its length will quadruple the number of sample points). The value of an increase in the number of samples, however, diminishes as the number of existing samples increases. The function **RtLtMapSetMaxAreaLightSamplesPerMesh()** sets a sensible limit to this increase, capping the number of samples which may be created by a given area light mesh. **RtLtMapGetMaxAreaLightSamplesPerMesh()** retrieves that limit.

23.4 Creating and using lightmaps

This section provides a step-by-step guide to creating and then using lightmaps, covering the export of data from a modeling package, calculation of lightmap UV coordinates, illumination of lightmaps, rendering with lightmaps and saving and reloading lightmap data.

23.4.1 Lightmap creation

Lightmap creation is performed in two stages: the export of lightmap-specific data from a modeling package and the creation of lightmaps (and calculation of per-vertex lightmap UV coordinates) in a RenderWare Graphics application (such as the lightmap example).

Exporting lightmaps-compliant data

In order for the **RtLtMap** toolkit to calculate per-vertex lightmap UV coordinates for world sectors and atomics, the exporter must export a second set of per-vertex UV coordinates, initialized to specific values (which are calculated on the basis of information that only the modeling package has access to). To enable this, ensure the "Generate RtLtMap UVs" option is enabled before exporting an atomic or world.

This option enables the export of the necessary data for **RtLtMap** to generate the lightmaps itself. It should not be confused with the option to export lightmaps generated by the application.

Setting up the plugin and toolkit

For an application to load or save lightmap data and to render geometry with lightmaps, the **RpLtMap** plugin must be attached - **RpLtMapPluginAttach()** should be called, after **RwEngineInit()** and **RpWorldPluginAttach()** and before **RwEngineOpen()**. The header **rpltmap.h** should be included in application code and the **RpLtMap** library linked into the application.

For an application to create and/or illuminate lightmaps, the **RtLtMap** and **RtBary** toolkits must be linked into the application, in addition to the **RpLtMap** plugin. The header **rtltmap.h** should be included.

Creating lightmaps

Once the **RpWorld** and/or **RpAtomic**s (correctly exported, as above) constituting a scene have been loaded from disk, lightmaps must be created for them and their per-vertex lightmap UV coordinates must be set up.

In order to create lightmaps for a scene, an **RtLtMapLightingSession** structure should be set up to specify the lightmapped objects therein. This structure should be allocated and then initialized with the function **RtLtMapLightingSessionInitialize()**, which sets up a pointer to the scene's **RpWorld** (the only value which *must* be specified before the **RtLtMapLightingSession** can be used to create lightmaps for the scene).

The function **RtLtMapLightMapsCreate()** is used to create the lightmaps for a scene and to set up per-vertex lightmap UV coordinates. This takes an **RtLtMapLightingSession**, density value and color value as parameters. The density may be worked out by trial and error (it depends upon the desired world-space resolution of lightmaps in the scene), but the lightmap example uses a simple calculation (involving the world's bounding box) to automatically determine a sensible value.

The color value merely specifies the color to which to clear the newly created lightmaps (if this parameter is NULL, the default black and white checkerboard pattern will be used).

Here is some example code, illustrating the above points:

```
{
    RtLtMapLightingSession     lightingSession;
    RpWorld                   *world;

    /* ...Stream RpWorldStreamRead into variable 'world'... */

    RtLtMapLightingSessionInitialize(&lightingSession, world);
    RtLtMapLightMapsCreate(&lightingSession, 100.0f, NULL);
}
```

At this stage the scene may be rendered (see the following section on rendering with lightmaps for details). Assuming the lightmaps have not been cleared to a user-specified color, the lightmaps should be visible as a uniform checkerboard pattern modulating the world's base textures. The higher the density value passed to **RtLtMapLightMapsCreate()**, the smaller the cells of this pattern will be. This will be more clearly visible if point sampling is used for the lightmap textures (see the API reference documentation for **RpLtMapSetRenderStyle()** for details).

23.4.2 Lightmap illumination

Once lightmaps have been created and mapped to world geometry, their texels must be set to values representing the intensity of static light incident at sample points on the surfaces of the lightmapped objects in the scene. The function **RtLtMapIlluminate()** is used to achieve this.

This function determines visibility between every light and every sample point in the scene (as specified by a **RtLtMapLightingSession**), in addition to evaluating a light falloff equation for each such pair. Depending upon the number of lights, the number of sample points and the complexity of the occluding geometry in the scene, it may take a very long time for this process to complete. Hence, it may be useful to perform lighting in slices (as described in the *23.3.1 Lighting Sessions* section introducing lighting sessions) and/or to use a **RtLtMapIlluminateProgressCallBack** (which may be specified in the **RtLtMapLightingSession**) to keep track of lighting progress.

Super Sampling

Lightmaps can be generated at a higher resolution and down sampled to a lower resolution for display. This can produced better quality lightmaps due to the higher sampling resolution without needing similar higher resolution lightmaps for display.

Supersampling is selected during illumination. **RtLtMapIlluminate()** contains a parameter, *SuperSample*, which is used to select the supersample value. This sets the sampling resolution to be a scale factor of the lightmap's resolution.

Area Lights

RtLtMapIlluminate() takes a pointer to a **RtLtMapAreaLightsGroup**, specifying area lights to be used during illumination. Several such structures may be chained together, so that (for example) if several worlds are connected together by portals, the area lights from all of the worlds may be taken into account during illumination.

Here is some example code demonstrating the creation and use of area lights:

```
{
    RtLtMapAreaLightGroup    *areaLights;

    RtLtMapSetAreaLightDensityModifier(0.5f);
    RtLtMapSetAreaLightRadiusModifier(2.0f);
    RtLtMapSetAreaLightErrorCutoff(4);
    areaLights = RtLtMapAreaLightGroupCreate(&lightingSession, 0);
    RtLtMapIlluminate(&lightingSession, areaLights, 1);
    RtLtMapAreaLightGroupDestroy(areaLights);
}
```

23.4.3 Rendering with lightmaps

Once lightmaps have been created for objects in a scene (whether this was performed during the current application execution or whether the objects and lightmaps have been loaded from disk), they may be rendered using the usual object rendering functions without modification. The lightmap rendering pipeline is automatically assigned to objects when **RtLtMapLightMapsCreate()** is called, or when objects are loaded from disk and are found to contain lightmapping extension data.

23.4.4 Saving and reloading lightmap data

Once lightmaps have been created for a scene, the objects in that scene should be saved to disk so that their lightmap extension data and the second set of per-vertex UV coordinates are stored.

Additionally, the lightmaps themselves must be stored. The function **RtLtMapTexDictionaryCreate()** may be used to create a platform-dependent texture dictionary containing all of the lightmaps used by the objects specified by an **RtLtMapLightingSession**. This may be saved directly, as one file, or the lightmaps may be converted into platform-independent **RwImage**s and saved individually. If the latter approach is chosen, the names of the image files must be the same as the names of the lightmap textures (retrieved using **RwTextureGetName()**), because these names are used when atomics and world sectors are loaded from disk, to determine which lightmap is used by which object.

The grouping of lightmaps is up to the user, lightmaps could be stored in the same texture dictionary as the scene's base textures if desired. All that is important is that the lightmaps are available when the scene's objects are loaded from file, either in the current texture dictionary (already loaded from disk) or as image files on the current image search path.

Fully functional file loading code is provided in the lightmap example, in the function **_loadWorld()**, in **lightmaps.c**.

23.4.5 Postprocessing lightmaps

Lightmaps on the PlayStation 2 uses a proprietary, two-pass algorithm, to render full-color lightmapped objects. This method gives better performance than a four-pass algorithm but requires post processing of the lightmaps and the objects' base texture.

Firstly, the texels of the lightmaps needs to be inverted. This is performed by the functions **RtLtMapLightingSessionLightMapProcess()** and **RtLtMapSkyLightMapMakeDarkMap()**.

Secondly, a 'luminance' value needs to be computed and stored in the alpha component of the object's base texture's texel. This would be produce an inverted display on a PC but would appear correct on the PlayStation 2. Likewise, non-processed lightmaps will appear inverted on PlayStation 2.

The function, **RtLtMapSkyLightingSessionBaseTexureProcess()** is used to compute and store the 'luminance' value. Two methods are available to compute this value.

RtLtMapSkyLumCalcSigmaCallBack() computes the luminance value using the three components of the RGB texels and is more suitable for fairly evenly lit scenes.

RtLtMapSkyLumCalcMaxCallBack() uses the maximum of the RGB components to compute the value. This function is better suited for scenes with sharp changes from well lit to very dark regions.

23.4.6 Host Generation

Lightmaps can be generated on a host platform, normally a PC, for use on a different target platform. In such situations, the exported texture dictionary must either be in the target platform's format or a platform independent format. Care must be taken when exporting the lightmap texture dictionary to ensure the lightmap's raster format is in the platform's optimal format.

The raster format on a PC may not necessary be suitable for the target platform. This can lead to incorrect lighting or pixelated images.

A second effect of using a different host to generate the lightmaps is the image may appear darker. This is normally due to an incorrect gamma setting in the objects' base texture.

23.5 The lightmaps example

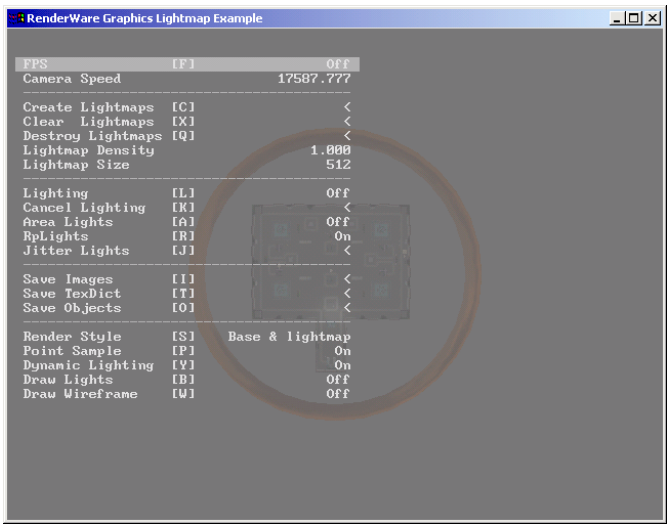
The Lightmaps example demonstrates most of the functionality of the **RpLtMap** plugin and **RtLtMap** toolkit. It can load a freshly exported world and/or atomics and generate, illuminate and store lightmaps for them. Its menu also provides access to many tweakable settings, such that, by experimentation, the user may come to better understand the way lightmaps work and the various quality/performance tradeoffs involved.

The artist or developer may wish to use this example to generate the lightmaps for a particular game scene. It can be amended fairly easily, to tailor the lightmaps for a more specific use, but it is provided primarily to demonstrate how to write code to create and use lightmaps in a RenderWare Graphics application.

Example: A view of the example scene, showing freshly created lightmaps

23.5.1 Starting the example

To start, launch the example in the usual way. The camera in its default state will show a view of a circular world as seen from above.



Example: Screen at launch

The application window may at first appear to be largely dark gray, since the scene is rendered with fogging enabled. To get a better view of the example scene, move carefully in towards it. To do this, remove the menu and readme text; on the PC press the spacebar, twice; on other platforms refer to the platform specific **txt** file for commands. Then press the "up" key, a few times to move the camera down to the floor of the world you are looking at –press the "down" key to back up if you go too far. Once you are close to the floor, drag the mouse up the screen to tilt the camera from looking straight down to looking horizontally.

At this point, lit items in the world are visible. Notice the dynamic light that passes over the central vase every five seconds. Note that this vase is vertex-lit rather than being lightmapped – it will still be lit during the lightmap illumination process. A couple of appropriate-looking materials in the scene are set up to be area light sources.

The default scene can be overridden by drag'n'dropping a **BSP** or **DFF** onto the viewer on a PC (or by passing a filename as a command-line parameter on a console). Note that the textures for a scene should be stored in a directory of the same name as the scene's **BSP** (excluding the "**.BSP**" extension). Currently the example does not support loading **.RWS** files.

23.5.2 The menu options

At this point it is worth displaying the menu again and looking at the options, and reading the details of the help file. They can be displayed by pressing the spacebar. This section will provide a brief description for each of the menu options.

FPS is used in other examples and merely toggles the display of the framerate in the upper right corner of the display window.

Camera Speed is used to modify the movement speed of the camera. Pressing the "left" key, or its equivalent, while this menu item is selected, will halve the camera's movement speed, whilst pressing the "right" key, or its equivalent, will double the camera's movement speed. This affords rapid movement around large scenes as well as precise positioning with respect to detailed geometry.

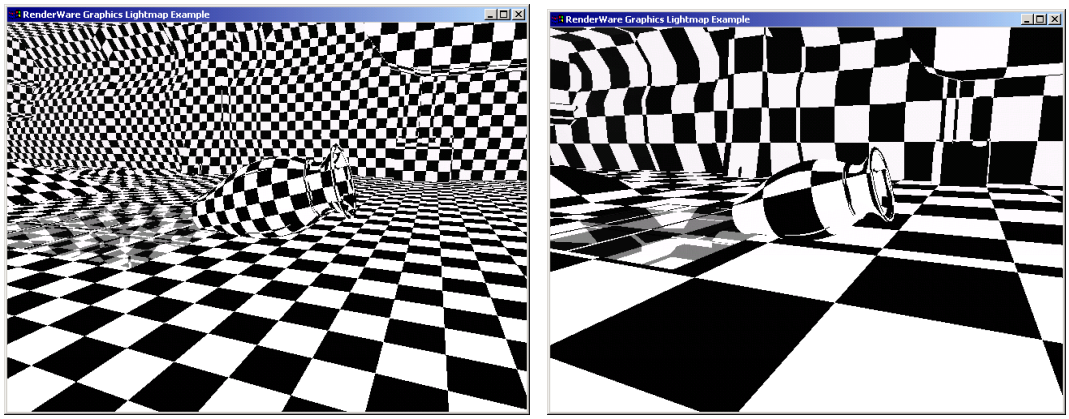
Lightmap creation

Create Lightmaps allocates lightmaps for the geometry currently in view, calculating per-vertex lightmap UV coordinates and initializing lightmaps to a black and white checkerboard pattern – these being performed through a call to **`RtLtMapLightMapsCreate()`**.

Clear Lightmaps calls the function **`RtLtMapLightMapsClear()`**, which clears the scene's lightmaps back to a checkerboard pattern, if they have been partially or fully illuminated.

Destroy Lightmaps calls the function **`RtLtMapLightMapsDestroy()`**, which releases the memory of the scene's lightmaps. The scene's geometry no longer references lightmaps and will no longer be rendered using the lightmap pipelines, so the checkerboard pattern (or static lighting) will disappear.

Lightmap Density alters the density of lightmap samples (texels) in world-space. This density value is passed to **`RtLtMapLightMapsCreate()`** when lightmaps are next created. Higher densities produce higher-quality lighting, but will also require longer processing times.



Higher and lower density lightmaps

Lightmap Size calls **`RtLtMapAtomicSetLightMapSize()`**, which sets the size (resolution) of lightmap textures. This value is passed to **`RtLtMapLightMapsCreate()`** when lightmaps are next created. This function is provided to allow the developer to choose between few, large lightmaps or many, small lightmaps.

Lightmap Supersample specifies the supersampling value during lightmap illumination. This value sets the sampling resolution by scaling the lightmaps texture resolution.

Lightmap illumination

Lighting initiates the illumination process for the areas of lightmapped geometry that are visible in the current view. The progress of this lighting 'session' is displayed, as a percentage completion value in the center of the view window. The initial view is cached, so that the camera may be moved during illumination. The lighting calculations may also be paused and resumed any number of times by re-executing this menu option.

Cancel Lighting cancels the current illumination calculations, if any are in progress. The next time the *Lighting* menu option is activated, a new lighting 'session' will be initiated.

Area Lights toggles the use of area lights during lightmap illumination calculations. If area lights are not to be used, the second parameter of **`RtLtMapIlluminate()`** is set to NULL when it is called; otherwise an **`RtLtMapAreaLightGroup`** pointer is passed, describing the area lights in the scene. The **`RtLtMapAreaLightGroup`** is created, by a call to **`RtLtMapAreaLightGroupCreate()`**, the first time that this menu option is toggled to **`TRUE`**.

RpLights toggles the use of the **`RpLight`**s in the scene during lightmap illumination calculations. The use of these lights is activated and deactivated by toggling their **`RpLightFlag`**s between lighting and not lighting atomics and world sectors.

Jitter Lights calls **`LightJitterCB()`**, to "jitter" each of the **`RpLight`**s in the scene. In this context, to jitter means to process a single light as if it occupied a small range of positions or angles (in practice, this means replacing the light with multiple, dimmer lights). The purpose of this is to soften shadows cast by these lights, the softness being proportional to the distance from the occluding object casting the shadow. To see the jittered lights directly, select the *Draw Lights* option, described below. Once lights have been jittered, they cannot be unjittered. Jitter is not part of the Lightmap API, but the function in the example shows the developer how the effect may be coded.

File handling

Save Images saves the lightmaps as individual, platform-independent image files.

Save TexDict saves the lightmaps and the normal textures into a single texture dictionary file. The dictionary can be in platform independent or platform dependent form.

Save Objects saves the scene's objects (world and/or atomics) to disk, with any new lightmap UV coordinates or lightmap plugin data that may have been created during the execution of the example. (This option assumes that your files are write-enabled.)

Display options

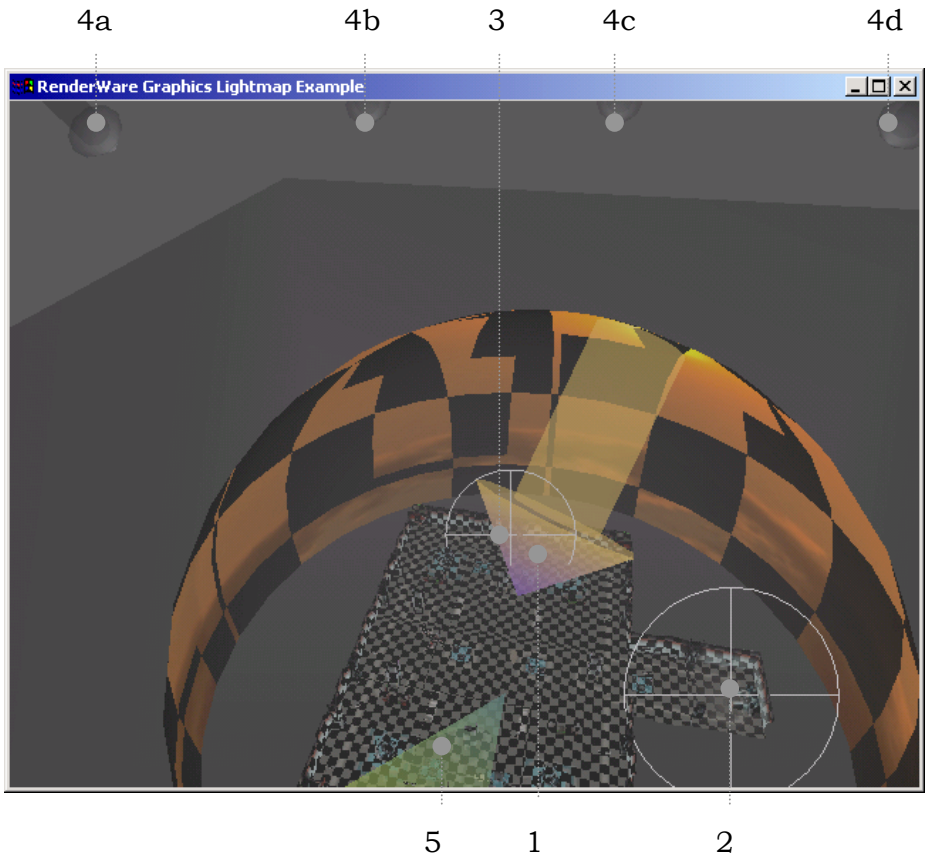
Render Style cycles between displaying lightmaps, base textures and the combination of both. If *Destroy Lightmaps* has removed the lightmaps (or if lightmaps have not yet been created), this option can show only the base textures.

Point Sample toggles the filtering mode of the scene's lightmap textures between point sampling and bilinear interpolated. Bilinear interpolation will always be used in a product, but point sampling shows lightmap texels as rectangles so that the layout of the lightmaps in the scene is more clearly visible.

Dynamic Lighting toggles the use of one ambient and one moving point light source. Both of these lights are dynamic – they are not used during lightmap illumination, but rather are dynamically combined with the static lighting represented by the lightmaps. The dynamic lights can be seen through the use of the *Draw Lights* option, described below. The effect of the point light is seen on the floor and walls in its ROI and it affects the central vase every five seconds or so.

Draw Lights toggles the rendering of visible 3D representations for the scene's **`RpLight`**s. If you back out of the world, you can see the directional lights outside the world that simulate sky light and the single (much brighter) directional light that simulates light from the sun. They are suspended high above the floor of the world, as illustrated in the screen shot below.

- The two white 'target' symbols (⊕) represent the static point light (1) and the moving dynamic point light (2).
- The large 3D arrow (3) represents the sun's directional light, and emanates from the image of the sun on the hemispherical sky.
- The four dark, barely visible, 3D arrows (4a-4d) show where four of the multiple directional light sources for the sky area light are placed.
- The cone (5) represents a spotlight.
- The dark gray box is the world's bounding box, drawn in the color of the scene's single ambient light source.



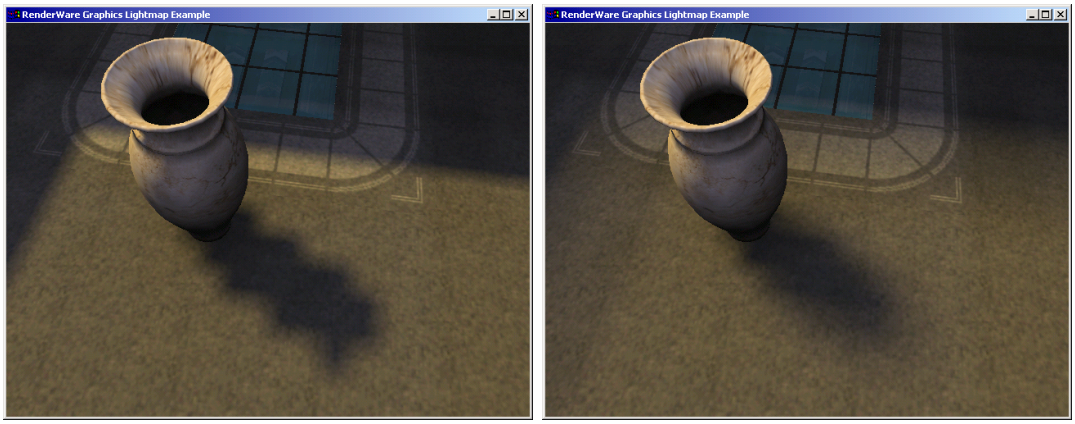
Draw Wireframe cycles between displaying no wireframe geometry, displaying wireframe bounding boxes for the scene's world sectors and displaying the world's triangles in wireframe in addition to the bounding boxes.

23.5.3 Options and issues

This section provides some further illustration of the issues involved with some of the options available when creating and using lightmaps.

Jittering

There is no function in the toolkit or plugin to jitter lights automatically. The Lightmaps example includes a function that demonstrates how to achieve this effect. It replaces a single, bright light source with several dull light sources, randomly displaced in angle or position (depending upon light type). As shown in the images below, the effect is to soften shadow edges, in proportion to the distance from the occluding object casting the shadow.



The shadow in front of the vase on the left reveals the texels of the light map. The jittered version to the right tends to hide them.

Varying lightmap resolution

It may be beneficial to vary lightmap resolution in different parts of a scene (for example, to provide detailed shadows in small areas, or to cover large areas without consuming huge amounts of texture memory). However, where lightmapped surfaces of different resolutions share a border, there is often an ugly visual discontinuity so this should be avoided if at all possible.

The image on the left depicts an area of varying lightmap resolution. The image on the right shows the resultant visual discontinuity.

Moving lights and pre-lights

The shadows of moving lights and/or objects cannot be represented using lightmaps. These shadows must be displayed using different means (such as projected textures, projected polygons or stencil-buffer shadow silhouette planes).

At run-time, pre-light values and the light from dynamic lights are added to lightmap values. In the *Lightmaps* example, the central vase uses pre-light vertex lighting rather than lightmaps to achieve a very similar effect (due to the high resolution of the vase geometry).

Point Sampling

The images below demonstrate the visual difference between bilinearly-filtered and point-sampled lightmap textures.

The image on the left shows a point-sampled lightmap, with individual texels clearly visible. The lightmap on the right uses bilinear interpolation

Switchable lightmaps

Under certain circumstances, it may be useful to generate two or more versions of the lightmap covering a sub-section of a scene. Such alternative versions may be swapped simply at run-time (using **`RpLtMapWorldSectorSetLightMap()`**), to give the effect of, for example, a light being switched on and off.

Overloaded illumination callbacks

As is described in greater detail in the API Reference documentation, two callbacks may be overloaded during the lightmap illumination process – the sample callback and visibility callback. The latter, for instance, may be modified to implement light filtering, such that geometry (or even volumetric fog) can act as a color-sensitive filter to light, rather than blocking it entirely. Effects such as light coloration through a stained-glass window, attenuation and diffusion due to light scattering within fog, or reflections from specular surfaces, can all be implemented by overloading the visibility callback.

23.5.4 Troubleshooting

The use of lightmaps to encode static lighting has its limitations, but it is well worth the extra work of designing game environments to conceal awkward shadows and orienting objects and lightmaps into alignment. Occasionally the edges of texels will be very obvious. A small area of high-resolution lightmap may help conceal them, or bilinear interpolation may be effective enough.

Lightmap density may be automatically scaled during lightmap creation, if it is found that all of the polygons of a single atomic or world sector cannot be packed into one lightmap. When this happens, the lightmap's world-space sample density is halved and the packing is retried. The result is that, on the surfaces of this object, lightmap texels will be twice as wide as on other surfaces. This can be rectified by increasing lightmap size or by decreasing the initial world-space lightmap sample density.

It is recommended to use grainy rather than smooth base textures with lightmaps. Smooth base textures make the transitions between lighting values in lightmaps much more pronounced (usually appearing as ugly banding).

Try to align texels with shadows and with objects that mask light. In the screen shot above, the shadow of the skylight is much more acceptable because it is aligned with the texels in the lightmap, in comparison with the shadow of the vase, which is diagonal and its jagged edges are obtrusive. This improvement is not due to any actual anti-aliasing process, though result often suffices anyway.

Light "Jittering", as described elsewhere in this section, can be used to further reduce aliasing artifacts at hard shadow edges.

On PlayStation 2 if previously-opaque objects begin to be rendered translucently, this is because base textures have been processed for lightmapping (affecting their alpha channel) yet the objects are no longer being rendered using the lightmaps pipeline. If, alternatively, lightmaps produce overly dark or 'burned-looking' results, then this base texture processing has *not* been done. See the PlayStation 2 API Reference documentation for further details.

23.6 Importing Lightmaps

Lightmaps can be generated externally and imported for use in RenderWare.

3ds max and Maya both have the ability to generate lightmaps. These lightmaps can be exported, just like materials and geometry, for use in RenderWare. For more information on how to export lightmaps, see the relevant Artist Guide for these packages.

To render these lightmaps, the lightmap plugin, **RpLtMap**, must be attached. Lightmaps exported by these packages are identical to those generated internally, so the setup procedure to render them is the same.

23.6.1 Manual Conversion

Lightmaps exported from 3ds max and Maya are converted automatically as part of the export process. If manual conversion is required, such as importing from a custom format or other sources, the **RtLtMapCnv** toolkit can be used.

RtLtMapCnv only performs the conversion of the external lightmaps. **RtLtMap** is required to create the internal lightmaps and assign the internal UV lightmap co-ordinates.

To import external lightmaps into RenderWare you need to:

- Export the external lightmaps, geometry and associate data.
- Create the internal lightmaps.
- Convert the lightmaps.

Exporting external lightmaps

Exporting a lightmap object involves exporting the lightmap textures and the object itself.

The lightmap textures can be exported as normal images, such as a Window bitmap images, or into a **RwTextDictionary**.

The lightmapped object is exported in the same way as a standard RenderWare object, but with additional properties attached. The user must provide the additional properties as additional input for the export process. For more information on how to export objects to RenderWare, see the World & Static Model chapter of the User Guide.

The additional properties are provided in two parts. The first part is in three **RpUserDataArrays**. These arrays are used to the.

- U component of the UV lightmap co-ordinate per vertex.

- V component of the UV lightmap co-ordinate per vertex.
- lightmap's name reference per triangle.
- The second part is a second set of UV texture co-ordinates. These are not used to store texture co-ordinates but, instead, the alignment axis of the vertices. This is, essentially, the major axis of the vertex's parent triangle's face normal.
- The encoding scheme for the alignment axis is:
- 0.0 represent the +ve X axis.
- 0.1 represent the +ve Y axis.
- 0.2 represent the +ve Z axis.
- 0.3 represent the –ve X axis.
- 0.4 represent the –ve Y axis.
- 0.5 represent the –ve Z axis.
- If a vertex is shared, but its parents' major axis is different, then the vertex must split.

Creating the internal lightmaps

Creating the internal lightmaps for conversion follows the same process as creating lightmaps for illumination.

A lighting session needs to be initialized using **`RtLtMapLightingSessionInitialize()`** for the **`RpWorld`** containing the imported lightmap data. The lightmaps for the **`RpWorld`** and any attached **`RpAtomics`** are then created using **`RtLtMapLightMapsCreate()`**.

During lightmap creation, the triangles are mapped onto internal lightmaps to generate RenderWare's own UV lightmap co-ordinates. The triangles may not necessary re-use the imported UV lightmap co-ordinates and may be re-scale and re-orientated.

Converting the lightmaps

Once the internal lightmaps have been created and new UV lightmap co-ordinates are generated for the triangles, the imported lightmaps are ready for conversion.

Unlike lightmap generation, lightmap conversion is a single pass and does not involve multiple 'illumination slice'. A second difference is **`RpWorld`** lightmaps and **`RpAtomic`** lightmaps are converted individually.

A lightmap conversion session, **RtLtMapCnvWorldSession**, must first be initialized using **RtLtMapCnvWorldSessionCreate().** A similar function, **RtLtMapCnvAtomicSessionCreate(),** creates a conversion session, **RtLtMapCnvAtomicSession**, for a **RpAtomic**. **RtLtMapCnvWorldSession** and **RtLtMapAtomicSession** are used to parameterized the conversion session. See the **RtLtMapCnv** API reference for more information.

The functions, **RtLtMapCnvWorldConvert()** and **RtLtMapCnvAtomicConvert()**, are used to perform the conversion process for **RpWorld** and **RpAtomic** respectively. These take the appropriate conversion session and a sample factor parameter. The sample factor is synonymous to super sampling in lightmap illumination. The internal lightmaps can be generated from a larger source image. The size of the source image can be a multiple of size larger, determined by the sample factor.

The external lightmaps are read in as required during conversion using the **RwTextureRead().** The location of the external lightmaps is set by **RwImageSetPath().**

After successfully converted the external lightmaps into internal form, the external data can be destroyed using **RtLtMapCnvWorldSectorCnvDataDestroy()** and **RtLtMapCnvAtomicCnvDataDestroy().** The conversion sessions can also be destroyed using **RtLtMapCnvWorldSessionDestroy()** and **RtLtMapCnvAtomicSessionDestroy().**

Likewise, the external lightmap images can also removed.

23.7 Summary

This chapter has covered many aspects of lightmaps; their purpose and mechanism, their strengths and weaknesses and costs and benefits, as well as techniques for their creation, importing and use in a RenderWare Graphics application.

It describes lightmaps as **RwTexture**s, mapped to the surfaces of **RpGeometry**s and **RpWorldSector**s in a scene by a second set of per-vertex UV coordinates. Using dual-pass rendering, this second texture is applied over the base texture for each surface, providing the appearance of detailed and realistic static lighting.

The lightmap toolkit provides routines to map surfaces into lightmaps and to light an entire scene, calculating the light values for each lumel (texel of a lightmap) and to storing them in the scene's lightmaps. The lightmap created by this process can then be loaded by the lightmap plugin later, and used in rendering the objects in the world.

The strength of this approach is that the time-consuming calculation of the brightness and color of light over the world can be performed once, as an offline process, imposing little run-time overhead.

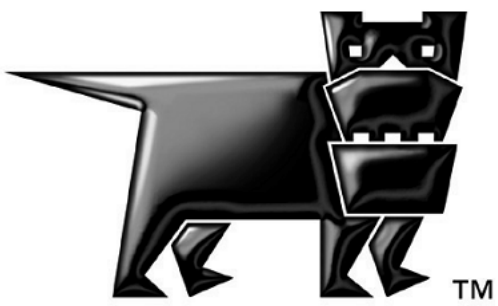
Lightmaps for PlayStation 2 must be postprocessed into darkmaps before they can be used, otherwise the rendered image will appear inverted.

The lightmaps example is covered in detail, which demonstrates, visually and in code, the majority of the functionality of the lightmap plugin and toolkit.

And finally, lightmaps can be generated in a third party package and exported for use in RenderWare.

Chapter 24

PTank



24.1 Introduction

This Chapter introduces the **RpPTank** plugin and the concepts of the particle and the particle tank. It describes the way they save processing time by side-stepping the usual 3D processes of RenderWare Graphics, and it explains how to use them.

24.1.1 What is a Particle?

In RenderWare Graphics a particle is a 2D shape.

It may have a single color that may have degrees of transparency. It may have an image or texture applied to it by UV coordinates.

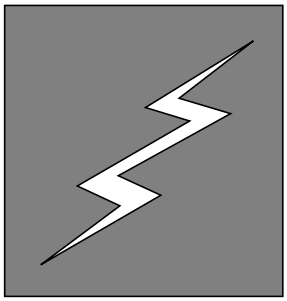
It is very similar to a sprite in early computer games.

A particle can be scaled up or down; it can be moved, even animated.

It does not exist as a separate data structure; it is integral to the particle tank.

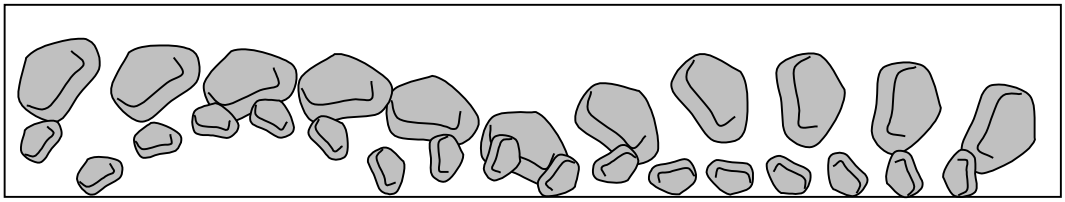
24.1.2 What Are Particles Used For?

It is often unnecessary to use RenderWare Graphics' usual 3D graphics capability. The memory required for a geometry, and the processing time required for each frame can be wasted on simple, dramatic or fleeting effects, and particles are good when transient, fast moving, small or flat images are required.



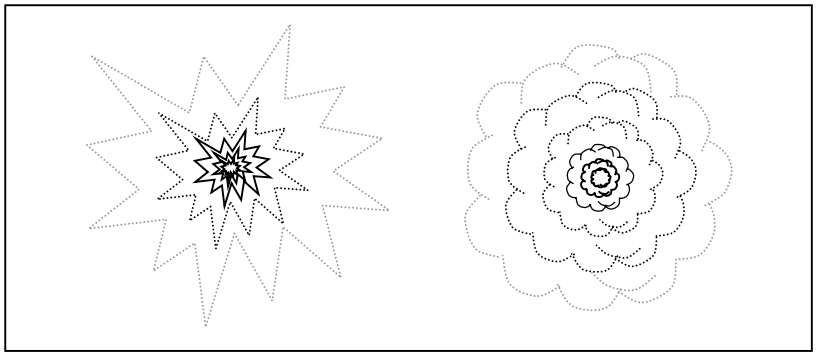
Particles are good for transient effects

An **RpPTank** particle can represent a moving rock. Particles can be scaled, so the image can get bigger as the rock approaches. The image can rotate. This is enough to convey rocks tumbling down a cliff or missiles hurled through the air.



One particle in two sizes represents two tumbling rocks

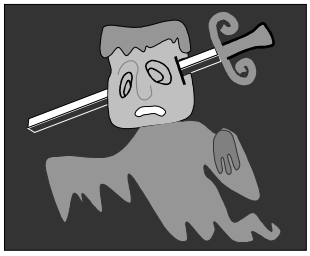
In some games, objects explode in a cloud. This sudden event requires only one image expanded rapidly. **RpPTank** particles can reduce their opacity (their alpha value) to disappear like a cloud and they can be scaled and rendered very quickly.



A single scaled particle can represent a crash or an explosion adequately

Flames can be conveyed by a series of semi-transparent images in front of the burning object. Particles provide the simple 2D animation that this effect needs, and 3D processing is not necessary. But particles can be displayed behind as well as in front of the burning object, so they can suggest 3D flames very effectively.

Tiny objects like feathers, snowflakes and sparkling highlights can be displayed, moved and replaced or removed very efficiently as particles, without any of the complexity of 3-D calculations.



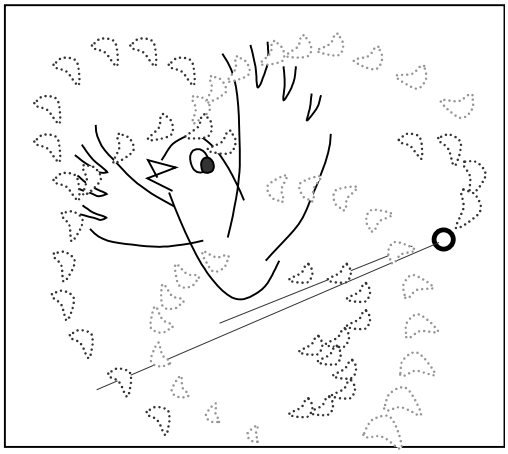
Particles are suited to cartoon-like transparent images

Simple ghosts, apparitions and specters require images that are bold but transparent, scaled, moving, and front-facing. If these images do not need to be articulated they can be rendered easily by particles.

The implementation of PTank allows the developer to store and animate data conveniently in simple ways. Controlling the movements of particles through space is a task for another plugin, like **RpPrtStd**.

24.1.3 What Is the Particle Tank?

The particle tank is a collection of particles. The word "tank" is used to suggest a container for particles.



Particles can be animated to represent feathers

The format of the particle tank or **RpPTank** is flexible. It can be organized as a structure or an array format and its data contents vary according to the image it conveys. Similar particles, like a flurry of feathers, or a sequence of transparent disks to represent smoke might be stored in the same **RpPTank**.

Transparent disk particles can represent smoke

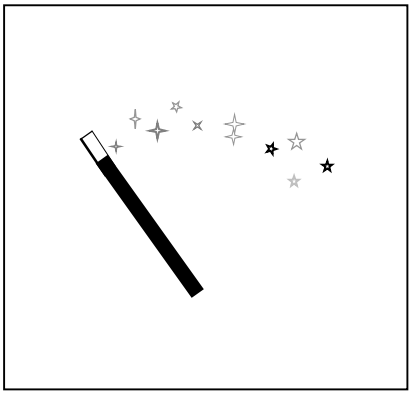
Particle tanks can hold particles of very different types, and several contrasting particle tanks can be active at the same time.

24.1.4 What Particles Are Not

Particles are not sophisticated or complicated. They are a simpler form of representation than RenderWare Graphics provides for 3D worlds and animated objects.

A particle is not three dimensional. It is a flat image like a sprite. So, by default, it is rendered as if it is parallel to the near Z plane, facing the camera. As a result, if the camera moves, the particle appears to turn to face the camera. Alternatively it must appear to be the same from all view points, like a glowing sphere.

The **RpPTank** plugin could be used to manipulate letters and titles and move them around the screen, but the **Rt2d** toolkit manipulates letters is designed to achieve these effects more easily.



Particles are suited to transient effects, like sparkle

Particles are also known for realistically representing galaxies, waterfalls, trees, vegetation and foliage, roman candles, rockets and other fireworks, realistic cloud formations, crowds and snow storms. These require not only particles, but a means to control their movements and interactions, and that is beyond the scope of **RpPTank**.

24.1.5 Other Documents

Little background knowledge is required for this topic.

- The API Reference gives a description of this plugin, each of its data structures and functions.
- Other concepts used in this Chapter are described elsewhere. Textures and their coordinates, colors and plotting modes are covered in *The Material Effects Plugin*, and *Immediate Mode* chapters.
- **RpPTank** interfaces with routines that differ widely across platforms. Be sure to consult the appropriate platform-dependent API Reference.

24.2 The Main Concepts

The concepts of a particle and the way it is stored in a PTank are central to this plugin. To use it, the developer also needs to understand the descriptor of the PTank's structure and its locking mechanism. They are described here.

24.2.1 The Particle

Particles are only defined in the particle tank, and there are two formats for the particle tank and several optional data fields that define it.

The definition structure, **RpPTankFormatDescriptor** holds a flag field. Two flags represent the PTank's data "organization". The other flags represent specific items of data. The preprocessor "PTank Data flag" values that follow represent the items of data that may be included. These values are passed to the **RpPTankAtomicCreate()** function to create the respective arrays.

Many of the following flags that describe a particle have an almost identical name containing the letter "L" for "lock" (rpPTANK***L***FLAG) in place of "D" for "data" (rpPTANK***D***FLAG). This similar set of flags is used with **RpPTankAtomicLock()** to lock, read and write the arrays created by the **RpPTankAtomicCreate()** function. The corresponding flags for locking and for creation are very closely related, so they are described together in the rest of this section on *The Particle*.

Flags for Spatial Description

- **rpPTANKDFLAGPOSITION** reserves space for a 3D point that will define the position of each particle. **rpPTANKLFLAGPOSITION** is used to lock, read and write it. The matrix value also contains position data, so this flag is not compatible with **rpPTANKDFLAGMATRIX**. If a particle has no position it must have a matrix to hold the equivalent position data and the debug version will assert if neither is present.

- **rpPTANKDFLAGUSECENTER** reserves space for vectors to position and rotate the particle by the center point specified by the function **RpPTankAtomicSetCenter()** rather than its default center or origin. (There is a shared value and there is no equivalent flag to lock, read and write shared values.)

- **rpPTANKDFLAGMATRIX** reserves space for a 3D **RwMatrix** to orient the plane of each particle. **rpPTANKLFLAGMATRIX** is used to lock, read and write it. By default the particle is plotted straight onto the view. This has the effect that it is facing the camera. (If there is more than one camera, it is facing every camera from every angle at the same time.) Setting the values of this matrix allows the particle to be reoriented, for example, like a door swinging on hinges. The **RwMatrix** contains position data, so it is not compatible with **rpPTANKDFLAGPOSITION**. and it contains dimension data, so it is not compatible with **rpPTANKDFLAGSIZE**.

- **rpPTANKDFLAGCNSMATRIX** reserves space for only a single shared or 'constant' 3D matrix to orient all the particles. (There is no equivalent flag to lock, read and write a constant.)

- **rpPTANKDFLAGSIZE** reserves space for a 2D vector that stores the dimensions of the rectangle for each particle. **rpPTANKLFLAGSIZE** is used to lock, read and write it. In essence, a particle is rectangular because it is defined in two dimensions. Applying other effects, in particular, applying an **RwTexture** can give it the appearance of any 2D shape within its rectangle.

- **rpPTANKDFLAG2DROTATE** reserves space for a real value to express the degree of 2D rotation, of the particle from its default orientation, in radians, clockwise, from –pi to pi. **rpPTANKLFLAG2DROTATE** is used to lock, read and write the rotation value.

- **rpPTANKDFLAGCNS2DROTATE** reserves space for only a single shared or "constant", real variable by which all the particles will be rotated. (There is no equivalent flag to lock, read and write a constant.)

- **rpPTANKDFLAGNORMAL** reserves space for a normal 3D vector. **rpPTANKLFLAGNORMAL** is used to lock, read and write it.

- **rpPTANKDFLAGCNSNORMAL** reserves space for only a single shared or "constant" 3D normal vector for all the particles. (There is no equivalent flag to lock, read and write a constant.)

Color Flags

- **rpPTANKDFLAGCOLOR** reserves space for an RGBA value for each particle. **rpPTANKLFLAGCOLOR** is used to lock, read and write it. The effect of the RGBA value is subject to the plotting mode. This flag operates differently from the others in the PTank because, by default, a shared color is created with the PTank. The **rpPTANKDFLAGCOLOR** flag overrides the default behavior and creates individual values in place of the shared one.

- **rpPTANKDFLAGVTXCOLOR** reserves space for an RGBA color for each vertex. **rpPTANKLFLAGVTXCOLOR** is used to lock, read and write it.

- **rpPTANKDFLAGCNSVTXCOLOR** reserves space for only a single shared or "constant" RGBA variable for all the particles. (There is no equivalent flag to lock, read and write a constant.)

Texture Coordinate Flags

- **rpPTANKDFLAGVTX2TEXCOORDS** reserves space for two coordinates representing the top left and bottom right texture coordinates. **rpPTANKLFLAGVTX2TEXCOORDS** is used to lock, read and write them.

- **rpPTANKDFLAGCNSVTX2TEXCOORDS** reserves space for only a single pair of shared or "constant" vertices for the top left and bottom right texture coordinates. (There is no equivalent flag to lock, read and write a constant.)

- **rpPTANKDFLAGVTX4TEXCOORDS** reserves space for four texture UVs corresponding to the four corners of the quadrilateral defined by the particle. **rpPTANKLFLAGVTX4TEXCOORDS** is used to lock read and write it.

- **rpPTANKDFLAGCNSVTX4TEXCOORDS** reserves space for a single group of four shared or "constant" UV coordinates that apply to all the particles in the PTank. (There is no equivalent flag to lock, read and write a constant.)

Organization Flags

- **rpPTANKDFLAGSTRUCTURE** reserves space for a flag to indicate that the PTank's organization is an array of structures. The flag exists in all PTanks, so there is no need to reserve space for it, and there is no equivalent flag to lock, read or write it. This setting is incompatible with that below, so both flags may not be set at the same time. But if neither is set **RpPTank** decides which organization to use, depending upon the current platform.

- **rpPTANKDFLAGARRAY** indicates that the PTank's organization is a structure of arrays. The flag exists in all PTanks, so there is no need to reserve space for it, and there is no equivalent flag to lock, read or write it. This setting is incompatible with that above, so both flags may not be set at the same time. But if neither is set **RpPTank** decides which organization to use, depending upon the current platform.

Several of these flags are incompatible. There follows a summary of groups of flags that are incompatible, and it is helpful to consider why they are incompatible. The reasons are all covered above, and the distinction between the shared, **CNS** values and the individual values is explained later under this heading on *The Particle*.

These flags and flag-pairs are mutually incompatible:

rpPTANKDFLAGCNSVTX2TEXCOORDS

rpPTANKDFLAGVTX2TEXCOORDS
rpPTANKLFLAGVTX2TEXCOORDS

rpPTANKDFLAGCNSVTX4TEXCOORDS

rpPTANKDFLAGVTX4TEXCOORDS
rpPTANKLFLAGVTX4TEXCOORDS

These flags and flag-pairs are mutually incompatible:

rpPTANKDFLAGCOLOR
rpPTANKLFLAGVTXCOLOR

rpPTANKDFLAGVTXCOLOR
rpPTANKLFLAGVTXCOLOR

rpPTANKDFLAGCNSVTXCOLOR

The flags and flag-pairs opposite each other in these two columns are incompatible:

rpPTANKDFLAGARRAY	rpPTANKDFLAGSTRUCTURE

rpPTANKDFLAGNORMAL	rpPTANKDFLAGCNSNORMAL
rpPTANKLFLAGNORMAL	rpPTANKLFLAGCNSNORMAL

rpPTANKDFLAG2ROTATE	rpPTANKDFLAGCNS2ROTATE
rpPTANKLFLAG2ROTATE	rpPTANKLFLAGCNS2ROTATE

rpPTANKDFLAGMATRIX	rpPTANKDFLAGPOSITION
rpPTANKLFLAGMATRIX	rpPTANKLFLAGPOSITION

rpPTANKDFLAGMATRIX	rpPTANKDFLAGSIZE
rpPTANKLFLAGMATRIX	rpPTANKLFLAGSIZE

This flag alone is compatible with all others:

rpPTANKDFLAGUSECENTER

In the debug version, incompatible flag settings will cause an assert.

A further distinction between the flags used in creating a PTank is that one value, "position", always applies to all particles, some values are shared between all particles and some can be either shared or independent. The table below summarizes these differences.

INDEPENDENT VALUES	VALUES THAT MAY BE EITHER INDEPENDENT OR SHARED	SHARED VALUES
position	size	vertex alpha
	orientation matrix	blend mode (1)
	normal vector	blend mode (2)
	2D rotation	an **RwTexture**
	color	an **RwMaterial**
	vertex colors	the UseCenter values
	vertex 2 coordinates	
	vertex 4 coordinates	

One last distinction between these flags is that the names of shared values contain the letters "**CNS**" for constant or "shared". For each "**CNS**" variable there is an equivalent non-**CNS** value indicating that an independent value is given to each particle. The two settings are incompatible. For instance, the PTank can have one color for all particles or individual colors for each particle, but not both. It may have one size for all, or individual sizes for each, but not both.

Because of this flexibility it is possible for the developer to make a mistake and address an individual value, like the color of the tenth particle, when there is only a single color, stored as a shared value. The **RpPTank** warns about these errors in one of two ways.

- If the developer addresses an individual particle's value, the address of the value must be found from **RpPTankAtomicLock()** and **RpPTankAtomicLock()** returns **NULL** when it is asked for the address of a non-existent value.

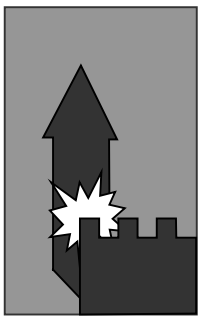
- If the developer addresses an shared value, the data must be returned from a Get function. Each of the functions that get shared values return **NULL** if their values do not exist.

So it is worth checking the return values of all these functions.

The **RpPTank** is most efficient when it stores similar particles, but the developer may maintain very different particles by adding multiple **RpPTank**s, with a different organizations and different data.

24.2.2 The Particle Tank

A particle tank, or **RpPTank**, is an extended **RpAtomic**. Its Create function, **RpPTankAtomicCreate()**, appends space to the atomic to hold the data used to store particles. There is an equivalent **RpPTankAtomicDestroy()** that destroys the atomic. Because it is an atomic, it has location and a bounding sphere within the **RpWorld** and this allows the sprites stored in the PTank to be processed for visibility and rendering in the same system as the rest of RenderWare Graphics. This means that particles can be masked or hidden completely like other objects.



A particle may be placed behind and in front of objects in the RpWorld

Because a PTank is an atomic, it is necessary to be able to test whether a particular atomic is also a PTank, and the function **RpAtomicIsPTank()** returns **TRUE** if it is a PTank.

The particle tank simply contains the data for a fixed maximum number of particles. The maximum number is returned by **RpPTankAtomicGetMaximumParticlesCount()** and is set by the **RpPTankAtomicCreate()** function. But sometimes the application may not use the maximum number of particles in the PTank to avoid processing particles that are not seen. In such cases, the functions **RpPTankAtomicGetActiveParticlesCount()** and **RpPTankAtomicSetActiveParticlesCount()** can be used to get and set the lower value.

Some effects, like explosions and lightening strikes appear only briefly, so they should be considered for rendering only when their visual effect is required. The developer can switch them off by setting their number of active particles to zero and switch them on again by restoring the number of active particles. This saves processing time, and the function **RpPTankAtomicGetActiveParticlesCount()** is provided for this purpose.

The PTank is platform specific, so **RpPTankAtomicCreate()** takes a third parameter for platform-specific flags. Refer to a platform-specific API Reference for details of this parameter.

The PTank has one of two formats depending on the setting of the two flags listed above:

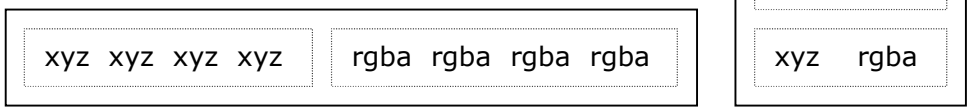
- **rpPTANKDFLAGSTRUCTURE** - array of structures

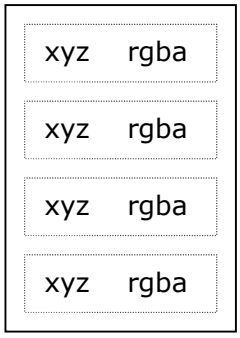
- **rpPTANKDFLAGARRAY** - structure of arrays

The diagram below represents four particles. They consist only of a vector and a color each. They are stored as an array of structures, "Structure Organization" and as a structure of arrays "Array Organization":

*A PTank formatted as an array of **structures**, right. (Structure Organized.)*

*A PTank formatted as a structure of **arrays**, below. (Array Organized.)*





Some platforms will perform significantly faster in one format than the other, so both formats are supported.

The function **RpPTankAtomicGetDataFormat()** returns the format descriptor, described below. It contains the flags field to show which of the two formats the PTank is in, together with settings to record which data it contains. But if the developer chose to allow PTank to adopt the more efficient format and set neither of the format flags, the Get Format function returns the flags for the format that the PTank actually adopted.

24.2.3 RpPTankLockStruct

The **RpPTankLockStruct** contains a pointer to data. The pointer is a pointer to an **RwUInt8**, but this type is chosen because it can be typecast and used for any other data type.

The other field in the lock structure is the "stride" of the data to be read or written. If the PTank is organized as an array of structures, the stride is equal to the size of the structure, in bytes. If the PTank is a structure of arrays, it is the size of the elements in the target array, in bytes. The value is used to step through the items of the target data.

The data pointer of the **RpPTankLockStruct** is passed to the **RpPTankAtomicLock()** function, which fills it with the address of the target data. The target data can then be read or written directly.

24.2.4 RpPTankFormatDescriptor

The **RpPTankFormatDescriptor** contains three **RwUInt32**s :

- the integer, **dataFlags**, contains the format flags that define whether it is to be a structure of arrays or an array of structures, and defines which data fields are to be included

- the integer, **numClusters**, the number of items of data in a particle. The data items are defined by the data flags that define a particle's contents and may include color, normal vector, position and matrix

- the integer, **stride**, is set to zero if the PTank is formatted in arrays. If it is formatted as an array of structures, the "stride" holds the size of the structure.

24.2.5 Locking and Unlocking

RenderWare Graphics usually requires the developer to

1. lock the data that is used for rendering before altering it

2. use the Get and Set functions to read or alter it

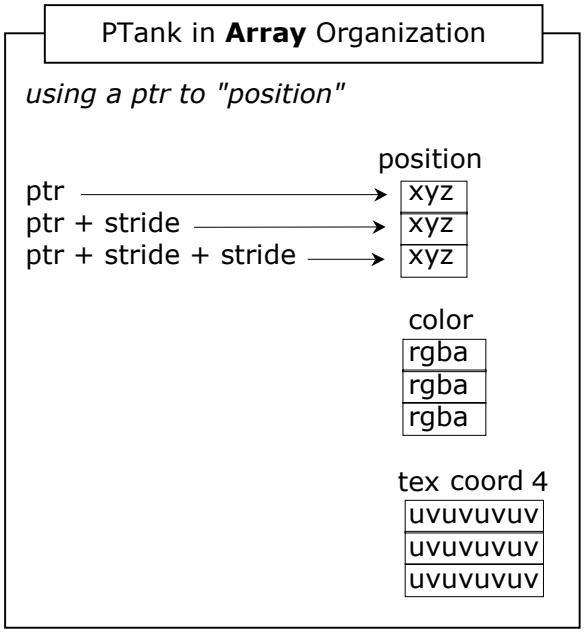
3. unlock it.

If you have locked data with the **rpPTANKLOCKWRITE** flag, the PTank has to re-instance the data for the current platform before rendering. So the developer should not use the Lock command needlessly. If it is accessed for reading only, it can be accessed with **rpPTANKLOCKREAD** to avoid this overhead.

The PTank plugin follows the same three steps, but the structure of the PTank makes it more efficient for the lock function to return a pointer to the locked data. So the second item in the sequence above is to address the data directly.

For this reason the PTank has many more values to be accessed than it has Set and Get functions. Under the heading *The Particle* above, there is a series of pre-processor constants that reserve space for various data items when a PTank is created. Each of the "`rpPTank`**`L`**`FLAG`**`...`**" constants listed there can be passed to the **RpPTankAtomicLock()** function to retrieve the address of its data so that the data must be read or written directly, without using Set and Get functions.

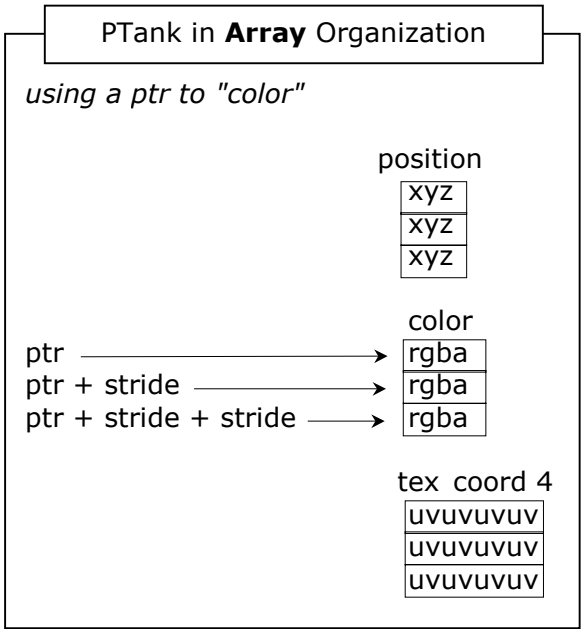
Whether the PTank uses Array Organization or Structure Organization, the developer can write code to read and write regardless of the organization.



Accessing PTank values in Array Organization

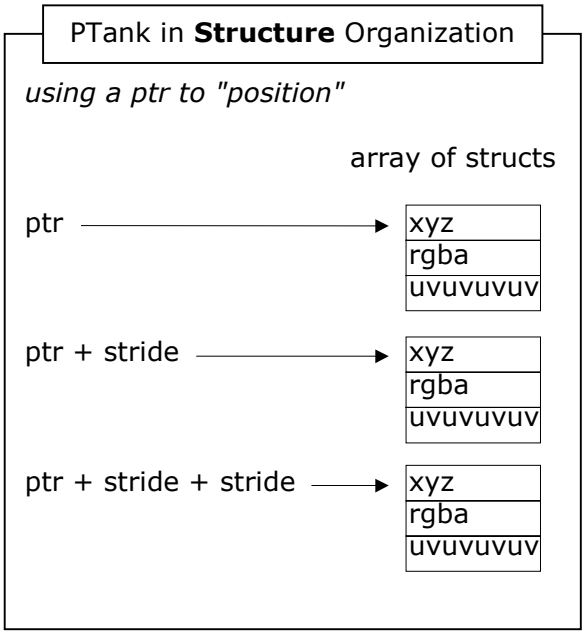
In the diagram above, the **`RpPTankAtomicLock()`** function has supplied a pointer ("ptr") to the first item of the position array. In "array organization" this is a pointer to an array of positions. The value "stride" is the size of the "xyz" vector, with any platform-specific padding.

In the diagram below, the same approach is used to read colors rather than positions. Only the initial value of the pointer is different.



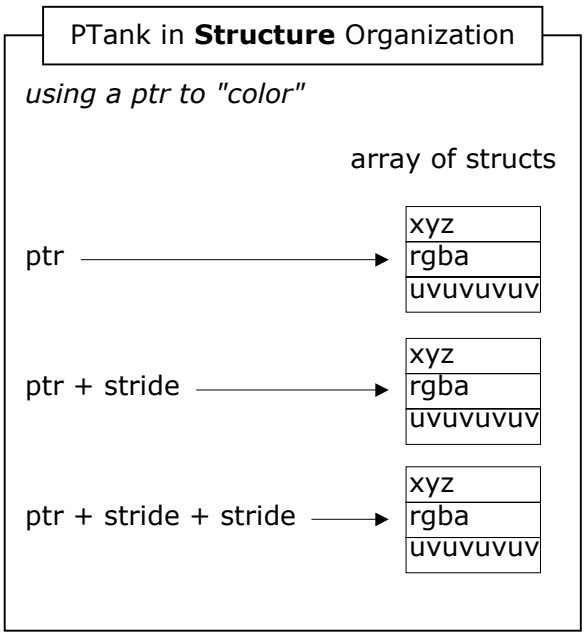
Accessing PTank values in Array Organization

The diagram below illustrates "Structure Organization", and the pointer refers to the first instance of the position in an array of structures. The value "stride" is the size of the structure, including any platform-dependent padding. So the value of the pointer, plus the value of stride is the address of the second position.



Accessing PTank values in Structure Organization

The same method works for colors, as illustrated, since the pointer points to the first color item, and pointer plus stride points to the second color item. And the same method can be applied to the four UV coordinates.



Accessing PTank values in Structure Organization

This means of addressing the data allows the developer to use the same code to calculate the pointer whether the PTank is an array of structures or a structure of arrays.

```
loop
    read the data at pointer (or write it)
    pointer += stride
end loop
```

This saves the developer duplicating code. But it allows the organization of the **RpPTank** to be determined automatically when the **RpPTank** is created, without the developer having to be aware which format was applied. If the **RpPTankAtomicCreate()** function is called without either the **rpPTANKDFLAGSTRUCTURE** or the **rpPTANKDFLAGARRAY** format flags being set, the format is chosen automatically. In this case, the developer need not know which organization is used, but the code will still address it correctly.

The advanced user may need to know whether the data is "structure organized" or "array organized". The answer can be found through the function **RpPTankAtomicGetDataFormat()** which returns the values in the **RpPTankFormatDescriptor**. The **RpPTankFormatDescriptor** contains the **dataFlags** and the **dataFlags** include the flags **rpPTANKDFLAGSTRUCTURE** and **rpPTANKDFLAGARRAY**. If the developer did not set these values in the **RpPTankAtomicCreate()** function, the **RpPTank** will have set them automatically, so the organization flags returned by **RpPTankAtomicGetDataFormat()** will accurately return the organization.

24.3 How to Use Particles Step by Step

These are the steps the developer needs to introduce particles into a developing application.

24.3.1 Initialization

- Append the header file **ptank.h** to the list of included files.
- Insert **RpPTankPluginAttach()** after **RwEngineInit()** and after attaching the World plugin, but before **RwEngineOpen()**.
- Use the function **RpPTankAtomicCreate()** to create a PTank.
- Set the number of particles that will be active with **RpPTankAtomicSetActiveParticlesCount()**.

24.3.2 Defining Particles

The function **RpPTankAtomicLock()** supplies a pointer to the data to write (or read) through the second parameter, called "dst" in the API Reference of the Lock command. The data pointer and the "stride" are supplied as elements of the dst structure.

```
RpPTankAtomicLock( pTank, &dst, rpPTANKDFLAGPOSITION,
                                                rpPTANKLOCKWRITE );
```

Loop through the PTank particles using the data pointer and adding "stride" to it at each iteration, as described under the previous heading. In this case, the data is a 3D vector and is cast to an **RwV3d ***, and it is transformed according to the state of other parameters.

```
if( dst.data )
    {
        RwV3dTransformPoints( (RwV3d*) dst.data,
                              &PositionsList[i], 1, Im3DmeshMatrix);
```

Write the required data to the address at **dst.data**.

```
        dst.data += dst.stride;
    }
```

The next step is to set the bounding sphere of the atomic. The developer may be used to using the function **RpMorphTargetCalcBoundingSphere()** but this is not suitable for particles. It is a safe starting point to set the sphere to be big enough to include everything. This will be inefficient and will have to be adjusted later. Also, particles usually present special effects and they are usually transient, so you will need to tune the bounding spheres for each effect. As there is only one bounding sphere for each PTank, you should consider grouping the particles by their position, in different PTanks. The bounding sphere is set with the function **RpMorphTargetSetBoundingSphere()**.

Finally unlock the PTank so that it can be used in the rendering process.

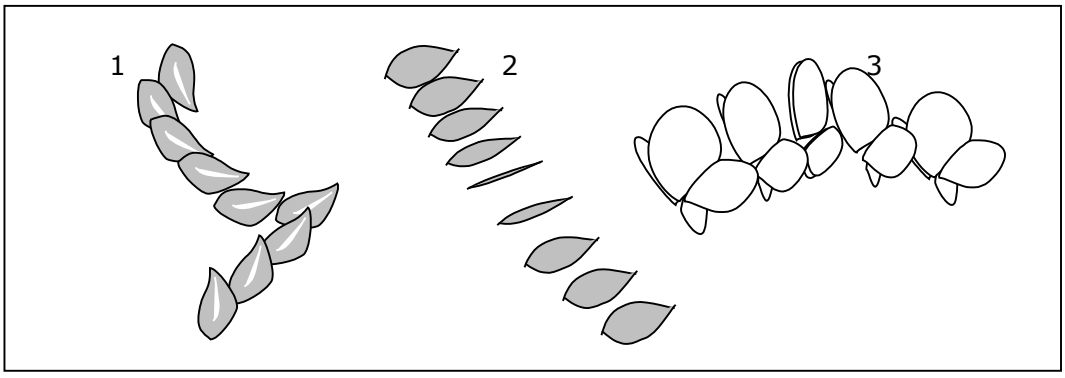
```
RpPTankAtomicUnlock( pTank );
```

As a minimum the user will have to define a position, a color and a size. This will define a visible rectangle in space.

There are other functions to define the particles displayed:

When a particle has to rotate (in 2D) the developer will often want to specify the point around which it will turn. This point will also be taken as its origin when it is positioned. **RpPTankAtomicConstantSetCenter()** sets this value but it will apply to all particles in the PTank.

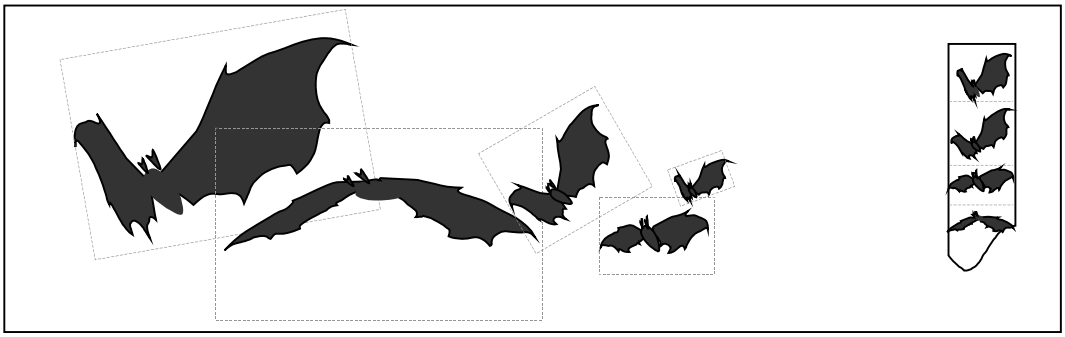
The function **RpPTankAtomicConstantSetMatrix()** sets a single matrix value that defines the orientation of all its particles to the screen. If there is a matrix for an individual particle it can be accessed with the **RpPTankAtomicLock()** function using **rpPTANKDFLAGMATRIX**.



1. Rotation of a particle to represent a falling leaf
2. Matrix changes to give 3D rotation
3. Matrix changed on two particles to animate a butterfly.

The function **RpPTankAtomicSetConstantRotate()** sets the angle of rotation (in radians) for all the particles in the PTank. If there is a rotation value for each particle in the PTank it can be accessed by the Lock function passing the constant **rpPTANKDFLAGROTATE**.

The function **RpPTankAtomicSetTexture()** sets the **RwTexture**, which may represent any 2D image for a given particle within the PTank. This can be used to used to display images from artists' tools on the particles. The alpha value can be used to make them partially transparent, or to alter the visible outline of the particle. (This method is implied in the two leaves and the butterfly in the illustration above.)



Separate areas of an RwTexture can be applied successively to a particle to animate its image

The function **RpPTankAtomicSetConstantVtx2TexCoords()** sets the UV values to be given to the top left and bottom right coordinates of the particle. An **RwTexture** will be applied to its surface with the coordinate values given as parameters mapped to points 0,0 and 1,1 on the particle. This can be used to map different section of the texture successively to the particle, rather like displaying successive frames of a film, for animation.

The function **RpPTankAtomicSetConstantVtx4TexCoords()** sets the UV values to be given to the four corner coordinates of the particle. An **RwTexture** will be applied to its surface with the coordinate values given as parameters mapped to points 0:0, 0:1, 1:0 and 1:1 on the particle. This too can be used for animation but is better used to stretch the texture out of its default, 1:1 aspect ratio.

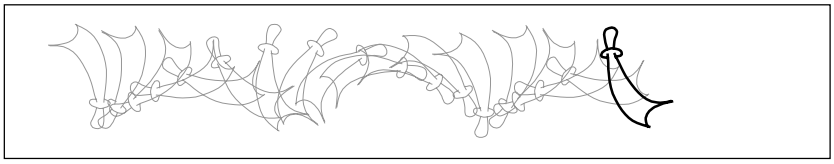
The function **RpPTankAtomicSetConstantVtxColor()** passes an array of four colors for the four corners of the particle. The color between is blended in proportion to its distance from each corner.

The function **RpTankAtomicSetVertexAlpha()** sets the opacity of all the particles in a PTank, and is useful for setting the transparency of smoke, clouds and ghosts.

24.3.3 Animation

For some effects the user may want simple animation.

An particle can be moved by updating its position. If its size is updated it will appear to get bigger or smaller, and therefore to move to or away from the camera.

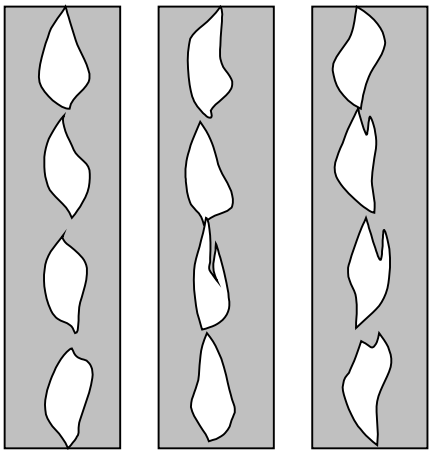


Position and rotation can be updated to give animation

If an image needs to be rotated like a scimitar thrown across the field of view, this can be achieved by updating the position and rotation value of the particle alone.

A particle can be rotated out of screen-alignment like turning a page of a book. A few useful effects may be achieved this way.

The vertex alpha value (or opacity value) of a particle can be altered to make it appear slowly like a ghost, or to make it disappear, like rising smoke, or the cloud that represents an explosion. The vertex alpha makes the whole particle increasingly more transparent or opaque, between its RGBA values and total transparency. It is addressed by the function **`RpPTankAtomicSetVtxAlpha()`**, and affects all particles in the PTank.



Flames can be animated as a series of images using RwTextures

An effect rather like movie film animation can be achieved by producing an **`RwTexture`** that resembles a film, consisting of successive images, like the frames of a movie. A particle is rendered with successive areas of the texture mapped to its UV coordinates. This is similar to projecting successive frames of a movie onto it, with the extra feature that the image can have transparency and adopt shapes other than a rectangle. This approach is useful for effects like flames, sparkle and specters.

24.4 Examples

The examples of **RpPTank** provided are "ptank2" and "ptank3".

"Ptank2" displays particles arranged as if on the surface of a rotating doughnut. All the particles in the example are identical. The user may change several parameters interactively from the menu and some from the code. It also adds the ability to rotate the particles and allows the user to alter their control parameters.

"Ptank3" is also based on the previous example and adds **RwTexture** to the particles and the user can adjust their parameters interactively.

24.5 Troubleshooting

- Use the debug version of the code. It will give an **rwDEBUGASSERT** when it finds potential conflicts, like incompatible flag settings.

- If values are written to particles but have no effect, or garbage values are returned, remember that PTank values exist in alternative forms. They may exist either as a single value shared by all the particles in the PTank, or as independent values for each particle in the PTank. Some values don't exist in all PTanks. If the developer addresses non-existent values nothing will happen and nothing will crash. But the constant values are address via functions that return NULL if invalid values are addressed, and all independent values are accessed via the Lock function. The Lock function will return NULL if it is asked for the address of a non-existent value. So make sure that these return values are checked.

- Some functions take pointers to an array of colors or vertices. If the array has too few elements the results will be unpredictable.

- Particles will not be rendered if they are outside the bounding sphere on their PTank. Bounding spheres need to be set by the developer, not by **RpMorphTargetCalcBoundingSphere()** when using particles. To see if the bounding sphere is clipping a particle, define an enormous bounding sphere for the particle's PTank.

24.6 Summary

Particles are simple, flat images, like sprites. They are much simpler to render than images that need the full 3D rendering processing. But they do exist in space in **RpWorld** and will be masked by objects in front of them.

Particles can be scaled, rotated, positioned and moved. They can have a texture applied to them. Their whole image can have a transparency level, so they can appear or disappear gradually, and their texture image can have transparency levels so they can appear as different shapes.

Particles are good for fleeting effects like explosions and for fast-moving or small objects that don't justify detailed 3D rendering.

The PTank or Particle Tank, is the object that stores particles. PTanks can have varied formats. Some values are held in common between all the particles in a PTank other values are stored independently for each particle. Multiple PTanks can be used at the same time to support widely different particles appearing concurrently.

There are two major variations on the internal organization of data in a PTank. The developer may chose the more efficient form or may allow PTank to decide. The code may be written to address the two different formats efficiently without the developer necessarily knowing which format is adopted.

By default, particles face the camera or cameras. Images that don't suit this form of image are probably not appropriate for particles. But they can be oriented relative to the screen plane. Lettering, titles and other 2D graphics are well supported in the **Rt2D** toolkit.

Snowstorms, galaxies, realistic clouds, flowing water and the more spectacular effects associated with particles require dedicated animation software.

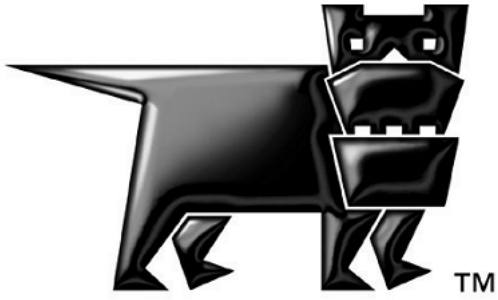
The locking function for particles differs from other elements of RenderWare Graphics in returning a pointer to internal data. The user updates the pointer and addresses the data directly. So, many properties of a particle do not have their own Get and Set functions.

Three pieces of Example code are provided for the user to compile and experiment with.

Particles can be used to produce animation by a few simple techniques. Most versatile is the use of successive images applied as textures rather like the successive frames of a movie.

Chapter 25

Standard Particles



25.1 Introduction

This chapter introduces the **RpPrtStd** plugin. It describes how the plugin can be used to create, animate and render an emitter and its set of particles.

Before you read this chapter, you should be familiar with particles in general and the **RpPTank** plugin.

The **RpPrtStd** plugin is used for animating a set of particles, not to perform any rendering. The **RpPTank** plugin is used for particle rendering.

25.2 The RpPrtStd Plugin

There are two main entities used in the **RpPrtStd** plugin, an emitter and a particle. They are, respectively, created from an emitter class and a particle class.

25.2.1 The Emitter

An emitter is an object that controls a set of particles. It is responsible for

- Emitting particles. New particles are created and added to the emitter's pool of active particles.
- Updating particles. The emitter's active particles are updated at regular intervals. This includes both animation and rendering data.
- Destroying particles. Particles that have exceeded their life cycle are removed from the active pool.

Each emitter has an emitter class and a particle class. The emitter class defines the emitter itself and the particle class defines the particles emitted from it. Once created, an emitter cannot change its emitter class or particle class.

Each emitter also has an **RpPTank**. This is because **RpPrtStd** only stores the animation for each particle. Rendering data is stored in **RpPTank**. The **RpPTank** plugin is private to each emitter and is not shared.

Like **RpPTank**, an emitter is an extension to an **RpAtomic**. The atomic is used to hold the emitter's bounding sphere and positional information. All other data is held internally in the emitter and particles.

25.2.2 The Particle

A particle can be described as a collection of data to represent a single entity in the world. For a more detailed description of a generic particle, see the **RpPTank** chapter of the user guide.

A particle in the **RpPrtStd** plugin should contain only its animation data. Rendering data, such as position and color, are stored within the **RpPTank**.

Particles in **RpPrtStd** are stored in batches as **RpPrtStdParticleBatch**, which is stored with the parent emitter. This provides a balance between creating all the particles at once and creating each particle individually. Grouping the particles into batches reduces the overhead of processing each particle individually. It also reduces the memory usage by only creating batches as required. Particle batches in the same emitter are always the same size.

Each batch contains a list of active particles, possibly containing fewer particles than the maximum size of the batch. Active particles are always stored together at the head of the batch. Particles in the remaining area are considered inactive and should not be processed.

Particles are never transferred between batches. This means that as the number of active particles decreases in a batch, particles from the next batch are not copied over to fill in the inactive area.

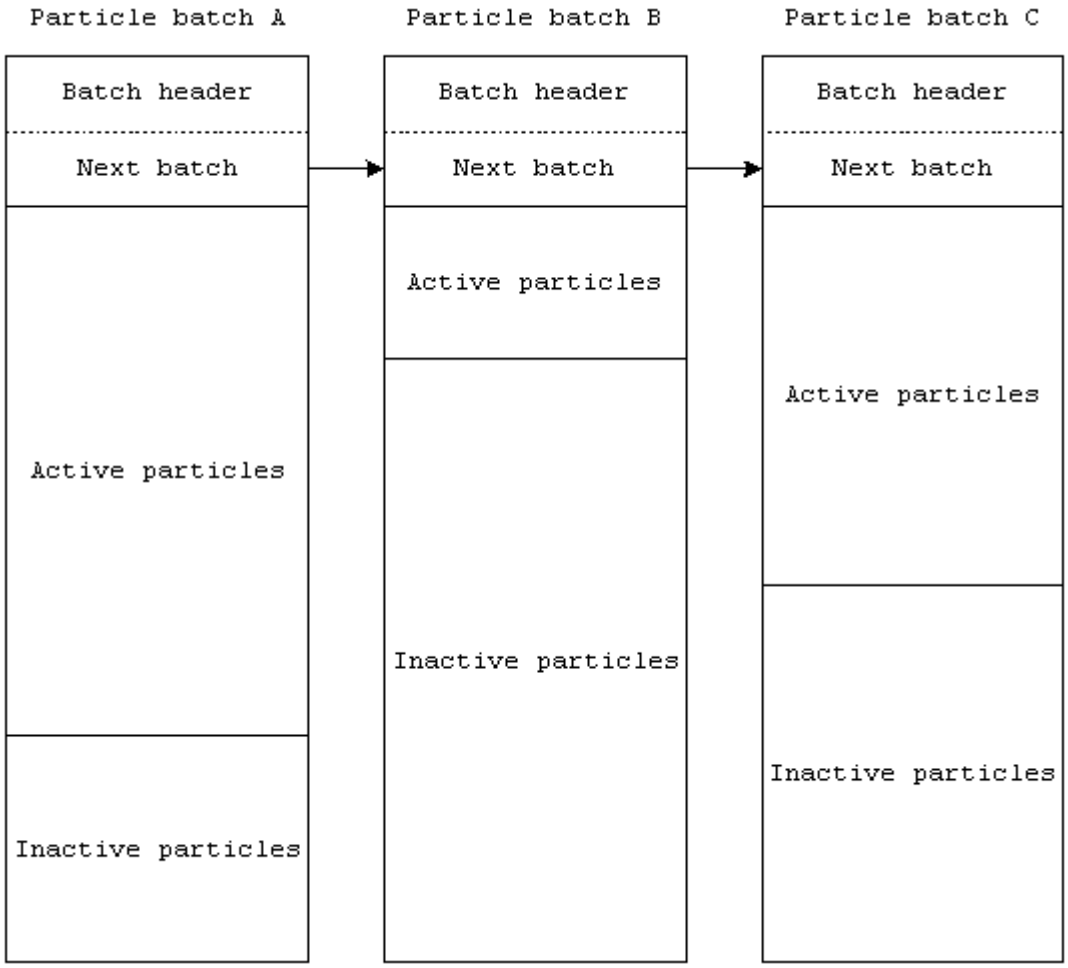


Fig 1. Example of a list of particle batches, each with different number of active particles.

25.2.3 The Emitter And Particle Classes

The emitter class , **RpPrtStdEmitterClass**, is a collection of callbacks and a property table. The property table describes the data structure within the emitter while the callbacks collection lists the functions for controlling the emitter.

Emitters created from the same class will share the same properties and callbacks. If an emitter requires a different set of callbacks but the same properties, a new emitter class needs to be created.

The particle class, **RpPrtStdParticleClass**, is similar to the emitter class. It contains a set of callbacks for controlling the particles and a property table to define the particle's data structure.

25.2.4 The Property Table

The property table, **RpPrtStdPropertyTable**, is used to define the data structure in an emitter and a particle. Properties are stored in a generic memory block of no fixed arrangement to allow emitters and particles to be created to a specific requirement. Properties not required can be omitted and user defined properties added.

The property table stores an identification number for each property with an offset to where the property's data is located in the memory block. This offset is from the start of the memory block where the data is held.

The emitter and particle both use the same property table structure, **RpPrtStdPropertyTable**, but they must not contain a mixture of emitter and particle properties. Emitter and particles must have separate property tables.

Property tables of similar types can be shared by more than one emitter class and particle class.

The properties are automatically aligned and padded to give the best performance on the current running platform.

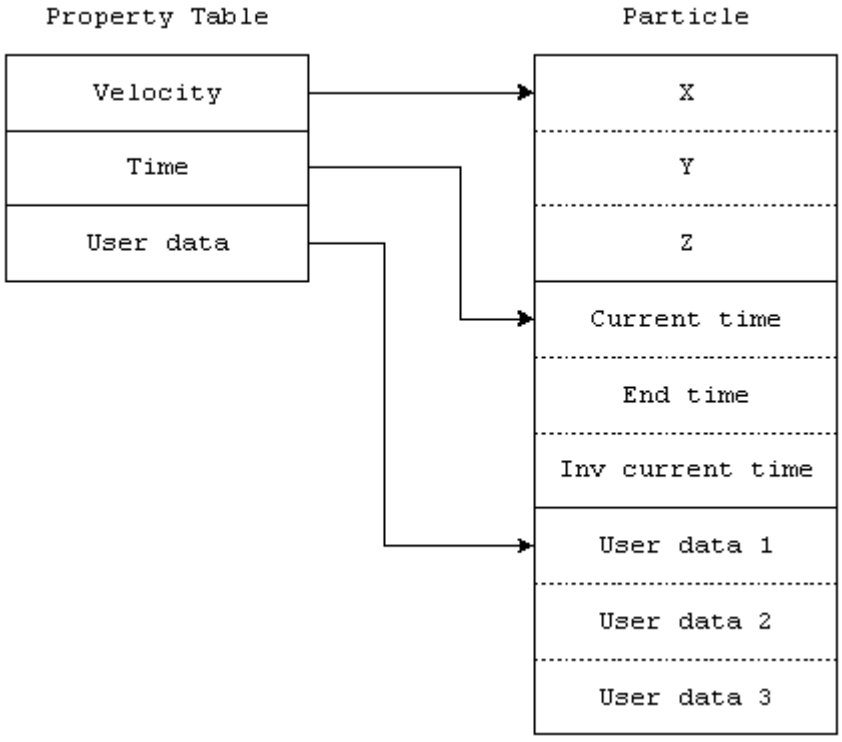


Fig 2. Example of a property table defining a particle.

25.2.5 The Emitter And Particle CallBacks

A set of callbacks is defined for controlling an emitter. These callbacks can be replaced with user defined equivalents if required. This could be for different behaviour or to support user defined properties.

The particle also has its own set of callbacks for controlling it. This set is different to the emitter's because of the slight difference in requirement between the two entities.

Each callback has a specific function and is called depending on the sequence of events of the emitter and particle.

One or several callbacks can be set to **NULL** if they are not required.

Emitter CallBacks

The default set of emitter callbacks is:

- **rpPRTSTDEMITTERCALLBACKEMIT** is the particle emission callback. New particles are created for the emitter in this callback.
- **rpPRTSTDEMITTERCALLBACKBEGINUPDATE** is called at the start of an update for an emitter.
- **rpPRTSTDEMITTERCALLBACKENDUPDATE** is called at the end of an emitter update.
- **rpPRTSTDEMITTERCALLBACKBEGINRENDER** is called at the start of a render for an emitter.
- **rpPRTSTDEMITTERCALLBACKENDRENDER** is called at the end of a render for an emitter.
- **rpPRTSTDEMITTERCALLBACKCREATE** is called when an emitter is created. This allows the user to set any user defined properties.
- **rpPRTSTDEMITTERCALLBACKDESTROY** is called when an emitter is to be destroyed. This allows the user to reset or destroy any user defined properties.
- **rpPRTSTDEMITTERCALLBACKSTREAMREAD** is called to read in an emitter from an input stream.
- **rpPRTSTDEMITTERCALLBACKSTREAMWRITE** is called to write out an emitter to an output stream.
- **rpPRTSTDEMITTERCALLBACKSTREAMGETSIZE** is called to return the size of the emitter when streamed out.

Particle CallBacks

The default set of particle callbacks is:

- **rpPRTSTDPARTICLECALLBACKUPDATE** is called to update the particles in the active pool. This will update both the animation and rendering data.
- **rpPRTSTDPARTICLECALLBACKRENDER** is called when particles are to be rendered.
- **rpPRTSTDPARTICLECALLBACKCREATE** is called when new particles are created. This is used to provide initial data to the particles. Animation and rendering are setup at this point.
- **rpPRTSTDPARTICLECALLBACKDESTROY** is called when particles are removed from the active pool.

The particle callbacks are called per batch rather than per particle.

25.3 Basic Usage

The basic operation of an emitter and particles can be divided into four groups:

- **Creation and destruction:** Emitters and particles are created and destroyed as needed.
- **Update:** Emitters and particles are updated at regular intervals.
- **Render:** Emitters and particles are rendered on screen.
- **Serialization:** Emitters are streamed in or out.

25.3.1 Creation And Destruction

Before an emitter and particles can be created, the respective class must first be set up. This, in turn, requires a property table.

Property Table

A property table is created using the function **RpPrtStdPropTabCreate()** and destroyed using **RpPrtStdPropTabDestroy()**.

```
RpPrtStdPropertyTable *propTab;
RwInt32 prop[4], propSize[4];

prop[0] = rpPRTSTDPROPERTYCODEPARTICLESTANDARD;
propSize[0] = sizeof(RpPrtStdParticleStandard);

prop[1] = rpPRTSTDPROPERTYCODEPARTICLEVELOCITY;
propSize[1] = sizeof(RwV3d);

prop[2] = rpPRTSTDPROPERTYCODEPARTICLECOLOR;
propSize[2] = sizeof(RpPrtStdParticleColor);

prop[3] = rpPRTSTDPROPERTYCODEPARTICLETEXCOORDS;
propSize[3] = sizeof(RpPrtStdParticleTexCoords);

propTab = RpPrtStdPropTabCreate(4, prop, propSize);
```

Example code for creating a property table.

Once created, the contents of a property table can be queried using **RpPrtStdPropTabGetProperties()**.

RpPrtStdPropTabGetPropOffset() is used to query an offset for accessing data within an emitter or particle. If the offset is non-negative, then it can be used as an offset into the generic memory block holding the data.

```
RpPrtStdPropertyTable *propTab;
RpPrtStdParticleBatch *prtBatch;
RwInt8 *prt;
RwInt32 offset;

offset = RpPrtStdPropTabGetProperties(propTab,
    rpPRTSTDPROPERTYCODEPARTICLECOLOR);
prt = ((RwInt8 *)prtBatch) + prtBatch->offset;

if (offset >= 0)
{
    prtStdCol = (RpPrtStdPrtColor *) (prt + offset);
}
```

Example code of using a property's offset to access data in a particle.

Emitter and Particle Classes

An emitter class is created and destroyed using **RpPrtStdEClassCreate()** and **RpPrtStdEClassDestroy()** respectively. Similarly, **RpPrtStdPClassCreate()** and **RpPrtStdPClassDestroy()** will create and destroy a particle class.

Both classes must be set up with a property table and a set of callback functions. **RpPrtStdEClassSetPropTab()** and **RpPrtStdEClassStdSetupCB()** will set the property table and callbacks for an emitter class. **RpPrtStdPClassSetPropTab()** and **RpPrtStdPClassStdSetupCB()** will set the property table and callbacks for a particle class.

```
RpPrtStdPropertyTable *propTab;
RpPrtStdParticleCallBackArray prtCB[1];
RpPrtStdParticleClass *pClass;
RwInt32 i;

pClass = RpPrtStdPClassCreate();

RpPrtStdPClassSetPropTab(pClass, propTab);

for (i = 0; i < rpPRTSTDPARTICLECALLBACKMAX; i++)
        prtCB[0][i] = NULL;

prtCB[0][rpPRTSTDPARTICLECALLBACKUPDATE] =
    RpPrtStdParticleStdUpdateCB;

RpPrtStdPClassSetCallBack(pClass, 1, prtCB);
```

Example code to set up a particle class.

Emitter

Emitters are created using **RpPrtStdAtomicCreate()**. This returns an extended atomic embedding an emitter object. This atomic does not contain any geometric data. The atomic can be destroyed using **RpAtomicDestory()**. This will also destroy all its particles.

Once created, the emitter needs to be initialized with default values. This involves getting the emitter from the atomic and setting the default properties value in the emitter. **RpPrtStdAtomicGetEmitter()** will return the emitter attached to the atomic. The default values are set by first querying the property table for the properties present and its offset. The data is then written into the emitter at the location given by the offset value.

The emitter must also be given the properties of its particles using the particle class, and the size of batches used to store the particles. This is done by the function **RpPrtStdEmitterSetPClass()**.

```
RpAtomic *atomic;
RpPrtStdEmitterClass *eClass;
RpPrtStdParticleClass *pClass;
RpPrtStdEmitterStandard *emitterStd;
RwInt32 prtBatchSize, offset;
RwPrtStdEmitter *emitter;

/* Create the particle atomic.
 * Assumes that both eClass and pClass were already created
 * elsewhere.
 */
atomic = RpPrtStdAtomicCreate(eClass, NULL);
emitter = RpPrtStdAtomicGetEmitter(atomic);
RpPrtStdEmitterSetPClass(emitter, pClass, prtBatchSize);
offset = RpPrtStdPropTabGetPropOffset(eClass->propTab,
    rpPRTSTDPROPERTYCODEEMITTERSTANDARD);
emitterStd = (RpPrtStdEmitterStandard *) (emitter + offset);

/* Set the emitter's maximum particles */
emitterStd->maxPrt = 6000;

/* Set the emitter's emission area */
emitterStd->emtSize.x = 0.0f;
emitterStd->emtSize.y = 0.0f;
emitterStd->emtSize.z = 0.0f;

/* Set the particle's size */
emitterStd->prtSize.x = 1.0f;
emitterStd->prtSize.y = 1.0f;

/* Set the particle emission gap : should not be bigger
 * than the batch size set during the creation code
```

```
 */
emitterStd->emtPrtEmit = 20;
emitterStd->emtPrtEmitBias = 0;
emitterStd->emtEmitGap = 0.0f;
emitterStd->emtEmitGapBias = 0.0f;

/* Set the particle's life span */
emitterStd->prtLife = 1.0f;
emitterStd->prtLifeBias = 0.0f;

/* Set the particles emission speed */
emitterStd->prtInitVel = 1.0f;
emitterStd->prtInitVelBias = 0.00f;

/* Set the particles emission Direction */
emitterStd->prtInitDir.x = 0.0f;
emitterStd->prtInitDir.y = 0.0f;
emitterStd->prtInitDir.z = 1.0f;

emitterStd->prtInitDirBias.x = 0.0f;
emitterStd->prtInitDirBias.y = 0.0f;
emitterStd->prtInitDirBias.z = 0.0f;

/* Set the force emission Direction */
emitterStd->force.x = 0.0f;
emitterStd->force.y = 0.0f;
emitterStd->force.z = 0.0f;

/* Set the default Color */
emitterStd->prtColor.red = 255;
emitterStd->prtColor.green = 255;
emitterStd->prtColor.blue = 255;
emitterStd->prtColor.alpha = 128;

/* Set the default Texture coordinate */
emitterStd->prtUV[0].u = 0.0f;
emitterStd->prtUV[0].v = 0.0f;

emitterStd->prtUV[1].u = 1.0f;
emitterStd->prtUV[1].v = 1.0f;

/* Set the texture */
emitterStd->texture = NULL;
```

Example code to create and setup an emitter with standard properties.

Particle

Particles are owned and controlled by the parent emitter. Because of this, the emitter looks after the creation and destruction of the particle batch.

New particle batches are created during particle emission. Empty particle batches are removed during update. Two callbacks are available for the user to perform additional actions when particle batches are created or destroyed.

The callback, **rpPRTSTDPARTICLECALLBACKCREATE**, is called whenever a new particle batch is requested for newly emitted particles. This callback is called after a new particle batch is created and passed to the callback to allow the user to perform any additional initialization.

rpPRTSTDPARTICLECALLBACKDESTROY is called just before a particle batch is removed to allow the user to perform any additional destruction.

25.3.2 Updating

Updating the emitter and the particles is done by the function **RpPrtStdAtomicUpdate()**. This function will call a series of callbacks to update the emitter and its particles.

```
Begin emitter update
For each batch in the emitter do
    If batch is empty do
        Remove batch from list
    Else
        Update particles in the batch
    Fi
Od
Emit new particles from the emitter
End emitter update
```

Pseudo code of an emitter cycle.

Emitter Update

There are two callbacks used to update the emitter. These are called before and after updating the particles.

The callback, **rpPRTSTDEMITTERCALLBACKBEGINUPDATE**, is called for the emitter at the start of an update cycle. This can be used to update properties necessary for updating its particles.

The callback, **rpPRTSTDEMITTERCALLBACKENDUPDATE**, is called at the end of the update cycle after the particles are updated. It can be used for any post particle update for the emitter, such as active particle count.

Particle Update

Particles are updated per batch rather than individually inside the emitter begin and end update.

Empty particle batches are first removed from the emitter's active batch list. The callback, **rpPRTSTDPARTICLECALLBACKDESTROY**, is called for each batch that no longer contain any active particles.

Non empty batches are updated using the callback, **rpPRTSTDPARTICLECALLBACKUPDATE**. The callback is responsible for updating the particle data and any data in the **RpPTank** that is used.

It is important that the data in the two areas are kept synchronized otherwise the wrong set of particles data maybe updated or removed.

Particles that are no longer active are removed by having their data area overwritten by the next active particle in the batch. This is also the case for data in the **RpPTank**.

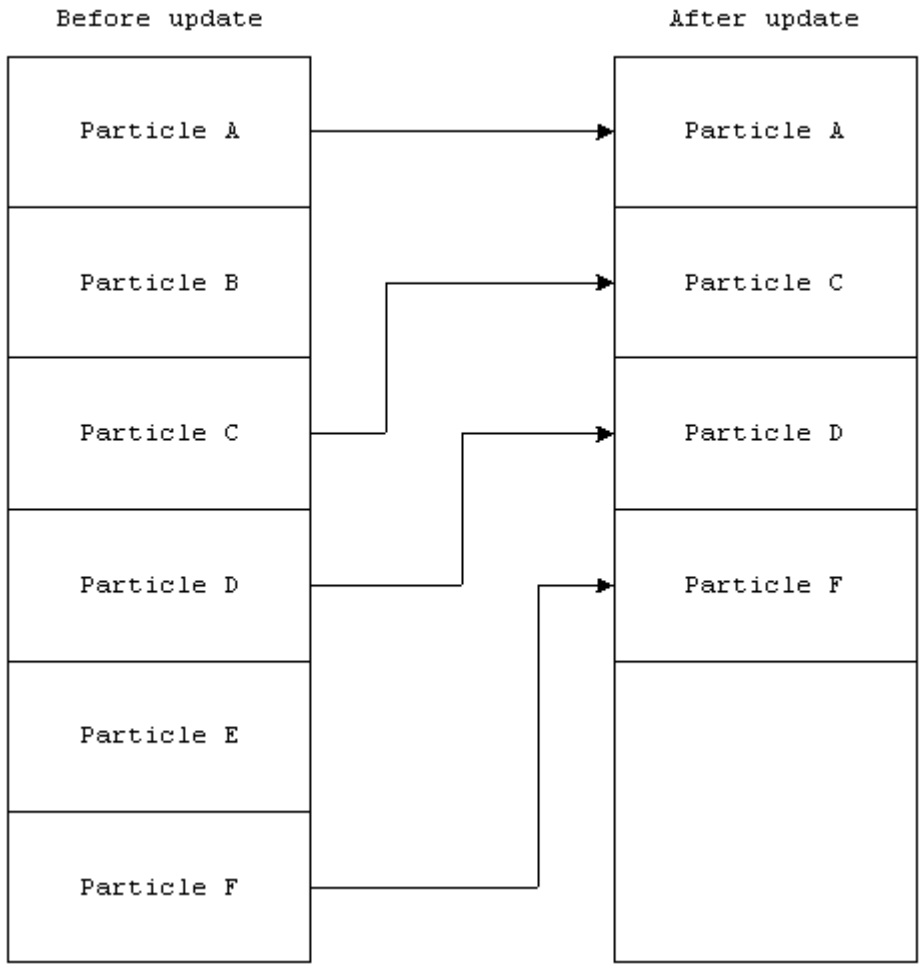


Fig 3. Example of particle removal. Particle B and E are removed by being overwritten by other remaining active particles.

New particles are emitted, i.e. created, emitted, by the callback, **rpPRTSTDEMITTERCALLBACKEMIT**. This is done after existing particles are updated. New particles can either be added to existing batches or a new batch created. New batches are created with **RpPrtStdEmitterNewParticleBatch()**. This will call the callback, **rpPRTSTDPARTICLECALLBACKCREATE**, and add the batch to the emitter's active list.

25.3.3 Rendering

The emitter and particles are rendered using the standard atomic rendering function, **RpAtomicRender()**. Depending on the type of emitter and particles, both entities can be rendered or not.

```
Begin emitter render
For each particle batch do
    Render particle batch
End emitter render
```

Pseudo code for an emitter render cycle.

Emitter Render

There are two render callbacks for the emitter. Similar to the update callbacks, these are called before and after rendering the particles.

rpPRTSTDEMITTERCALLBACKBEGINRENDER is called at the start of a render cycle.

rpPRTSTDEMITTERCALLBACKENDRENDER is called at the end of the render cycle.

Particle Render

Particles are rendered by the callback, **rpPRTSTDPARTICLECALLBACKRENDER**. This is called once per batch in the particle render loop.

25.3.4 Streaming

Streaming is supported by the plugin for some of the data types. There are two methods of streaming the data, embedded or non-embedded.

- **Embedded:** Embedded mode means the various data types are embedded into the emitter's chunk in the stream.
- **Non-embedded:** Non-embedded mode means various data types are placed in their own separate chunks in the stream.

The stream mode is set using **RpPrtStdGlobalDataSetStreamEmbedded()**. **RpPrtStdGlobalDataGetStreamEmbedded()** will return the current mode.

Property Table

The property table is streamed indirectly for both modes. If the property table is embedded into the emitter chunk, it is streamed with the emitter.

In non-embedded mode, the property table is streamed using the functions, **RpPrtStdGlobalDataStreamRead()** and **RpPrtStdGlobalDataStreamWrite()**.

Emitter Class and Particle Class

The streaming of the emitter and particle class is similar to property table streaming. It can be embedded with the emitter chunk or in a separate chunk with the property table.

In non-embedded mode, the emitter class and particle class is streamed with the property table in a single chunk. It uses the same function as the property table for streaming. **RpPrtStdGlobalDataStreamRead()** and **RpPrtStdGlobalDataStreamWrite()** will stream the property table, the emitter class and particle class.

Callbacks are not data and so cannot be streamed as such. In order to setup the emitter's callbacks correctly, the callback **RpPrtStdEClassSetupCallBack** is called after each emitter class is streamed in. It is the responsibility of this callback to setup the callbacks correctly for each emitter class. This callback is set by **RpPrtStdSetEClassSetupCallBack()**.

Similarly, the particle class must also be setup correctly by using the **RpPrtStdPClassSetupCallBack** callback. This callback is set by **RpPrtStdSetPClassSetupCallBack()**.

Emitter

The emitter atomic is streamed using the standard atomic streaming functions such as **RpAtomicStreamRead()** and **RpAtomicStreamWrite()**. These will call a set of callback functions to stream the emitter's property data within the atomic.

The layout of the data within the stream is user definable and does not need to be identical to that within the memory block.

The callback, **rpPRTSTDEMITTERCALLBACKSTREAMREAD**, is called when the emitter is read in from an input stream. This callback is used to read in the property data from the stream and store it in the appropriate location within the emitter's memory block.

The callback, **rpPRTSTDEMITTERCALLBACKSTREAMWRITE**, is used to write out the emitter's property data to an output stream. The callback may choose to omit some data from being written to the stream.

The callback, **rpPRTSTDEMITTERCALLBACKSTREAMGETSIZE**, is used to query the size of the data block that will be written to an output stream.

Particle

Particles are not streamed.

25.4 Standard Properties

The plugin provides a set of properties and callbacks for creating, animating and rendering simple particles.

These properties and callbacks can be used alone or with your custom properties and callbacks.

The standard callbacks are designed to handle all the standard properties and the possible combinations of those properties. They will also handle the rendering data stored within **RpPTank**.

Standard callbacks can also be used to look after a portion of the data in **RpPTank,** processing the remainder privately.

Two flags are stored within the emitter's **RpPTank** property structure, **RpPrtStdEmitterPTank**. These flags allow you to enable and disable **RpPTank** properties from being processed by the standard callbacks.

- **Emit Flag:** This allows you to selectively enable and disable properties within the **RpPTank** from being initialized by the standard emit callback, **RpPrtStdEmitterStdEmitCB()**.

  The emit flag allows you to initialize some data in the **RpPTank** privately while using **RpPrtStdEmitterStdEmitCB()** to perform the rest of the initialization.

- **Update Flag:** This is used like the emit flag but enables and disables properties from being updated in the standard particle update callback, **RpPrtStdEmitterStdUpdateCB()**.

  You can use this flag to select **RpPTank** properties that are updated by your own update callback.

You are responsible for synchronization of the render data in the **RpPTank** with the animation data in the particle for any **RpPTank** properties that were disabled with the use of flags.

More information on the properties and callbacks can be found in the **RpPrtStd** plugin's API reference manual.

25.5 Summary

The **RpPrtStd** plugin is for animating particles. It is used in conjunction with the **RpPTank** plugin for animating and rendering a set of particles.

The particles' data is stored across the two plugins. Data for animating particles is stored in the **RpPrtStd** plugin. Rendering data is stored in the **RpPTank** plugin. Like **RpPTank**, **RpPrtStd** uses an extended atomic to store the emitter and particle data.

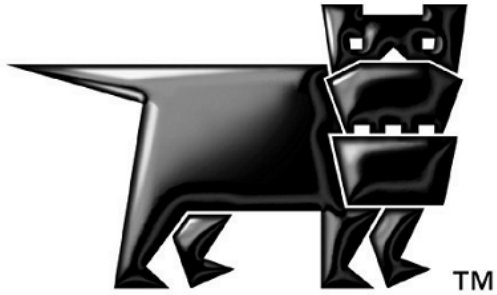
Emitters and particles do not use a fixed data structure. A property table is used to describe the data structure within these two entities, allowing them to be tailored to specific requirements.

Callback functions are used for manipulating the emitters and particles. These can be replaced by your own functions.

Emitters and particles can share emitter and particle classes respectively. These classes can share property tables but only those of the same type. Emitter classes and particle classes cannot share the same property table but can share property tables in their respective class.

Chapter 26

B-splines and Bézier Patches



26.1 Introduction

This chapter falls into three sections.

The first describes the way RenderWare Graphics adapts B-splines to deal with curved lines and paths, in the **RpSpline** plugin.

The second section deals with the way RenderWare Graphics encodes curved surfaces, through Bézier patches. It explains some of the methods and the API calls which render patches in the **RpPatchMesh** plugin.

The third section covers the **RtBezPat** Toolkit. Using the Toolkit, the developer who wants to get more out of the code can call the most useful functions hidden below the surface of the **RpPatchmesh**. It includes functions that find matrices of vectors for positions and tangents and other functions that speed up calculations. These calls require some knowledge of the mathematics of curved 3d surfaces, and they are a valuable resource in their own right.

26.1.1 Other Documents

- See the API reference for details of the code for **RpSpline**, **RpPatch** and **RtBezPat**.
- The Appendix, Recommended Reading, lists sources that give background in this subject.
- This chapter assumes you have some familiarity with the concepts of materials, skinning, atomics, clumps and streaming from their descriptions in this User Guide. They can be found in the table of contents. In particular, it is assumed that the reader knows the Chapter on *Dynamic Models*. The patch mesh code is designed to integrate with the other elements of RenderWare Graphics. So the same methods and API calls that were used with the flat polygons, in *Dynamic Models*, are re-used in the patch mesh section of this chapter to control the rendering, lighting and texturing of surfaces which are curved. Functions and data are named to parallel those used earlier, and the code example **patch**, used in this chapter, is based on the earlier example, **geometry**. (For completeness it should be noted that one area of functionality of **RpGeometry**, morphing, is not implemented for patches.)
- Several technical and mathematical topics touched on in this chapter are covered on specialist web sites and can be found by a web search.

26.2 B-splines

26.2.1 Introduction

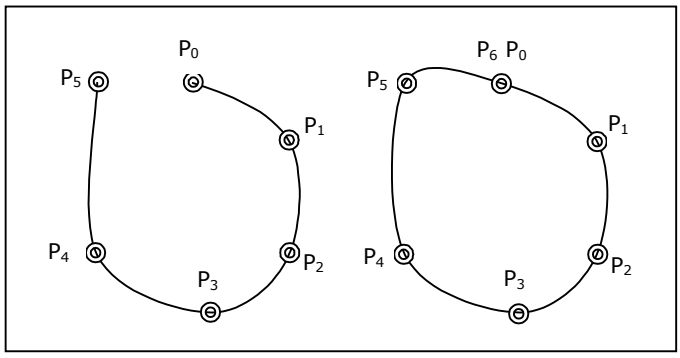
RenderWare Graphics uses uniform B-splines to define paths for cameras, lights and other objects. The **RpSpline** plugin implements B-splines and it is described here.

26.2.2 What Are B-splines?

The third degree B-splines that RenderWare Graphics uses are defined by a sequence of control points in 3d space. The sequence can have four or more control points. The position of any point on the B-spline that occurs between two control points in the sequence is determined by the two control points before and after it in the sequence; four points in all. Only the first and last points in an open B-spline sequence are on the curve. The curve bows towards each of the other control points in turn, but does not normally go through them. Expressed in intuitive terms, the curve is attracted to each of its control points in turn.

Control Point Numbering

In **RpSpline**, a B-spline's control points are numbered beginning from P_0. By convention P_{n-1} refers to the last point. In a closed spline, where the first point is also the last point, both P_0 and P_n refer to the same control point.



Numbering of **RpSpline** control points

Internally **RpSpline** may add extra control points at the start and end of the spline but this is transparent to the developer and does not affect the control point numbering in the API functions.

If a B-spline is defined by points numbered P_0, P_1, P_2 to P_n, then the points on the curve segment P_i to P_{i+1} are calculated from points P_{i-1}, P_i, P_{i+1} and P_{i+2}. The formula for these segments applies along the whole curve. It even applies on the first segment P_0 to P_1, and the final segment, P_{n-2} to P_{n-1}, because **RpSpline** intercepts the numbering of these control points as explained under Control Points at Open Ends, below.

26.2.3 Some Features of B-splines

Smoothness

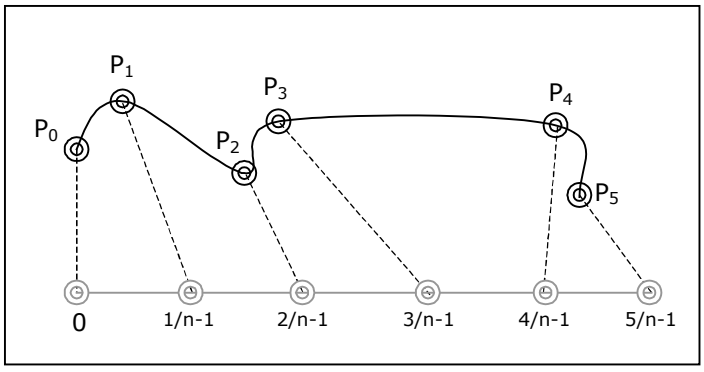
Perhaps the most common types of spline in computer graphics are B-splines, Bézier curves and the closely related Hermite curves. B-splines are rather smoother or rounder than the more angular Hermite curves, and the Bézier curves that are used in RenderWare Graphics' patches.

On-curve and Off-curve Control Points

Control points P_0 to P_{n-1} are an integral part of a B-spline and its mathematics, but on-curve points are more intuitive and more useful for many purposes. So RenderWare Graphics incorporates an algorithm to convert efficiently between off-curve points and their corresponding on-curve points and vice versa. So, as far as the developer is concerned, all the control points that **`RpSpline`** uses are on-curve points. In this section on **`RpSpline`** only, this document will refer to control points as if they were all points on the B-spline.

Real-world Space and Parameter Space

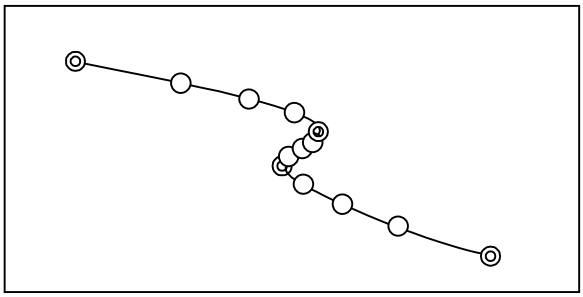
A B-spline, like other types of spline, can be seen as a sequence of points in real-world space, that define a curved line. For calculation purposes the B-spline can also be considered as a 1d progression where the control points are spaced uniformly or irregularly along its length. This second representation is referred to as "parameter space". In parameter space a uniform B-spline's control points are spaced uniformly (or evenly) in parameter space, at intervals of $1/(n-1)$ for a curve with open ends, and $1/n$ for a curve that forms a closed loop back to its starting point. Each point of the B-spline represented in parameter space corresponds exactly to one in the real-world B-spline.



Real-world space corresponds to parameter space

Velocity

The control points that define a curve can be spaced far apart or close together, in real-world space. If an application plots the progress of an object at intervals derived from the spacing of the control points in real-world space, the object's speed will be slower on the tighter bends, where the control points are closer. This can be useful because it can correspond to the behavior of racing cars on a circuit and to the number of straight segments or facets needed to represent curves where the curvature is tighter.



Control points, and points derived from them,
are closer where the spline curves more steeply.

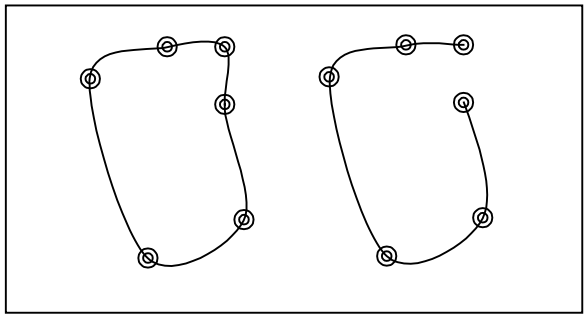
This is not the sense in which "velocity" is used with **RpSpline**; the developer decides how often to sample the curve and re-draw the object moving along it. But RpSpline introduces "velocity" in two enumerated values that affect the behavior at the ends of B-splines.

If the curve represents a path for a solid object it may also be useful to specify that the object accelerates at the start and decelerates toward the end. This feature is described in RenderWare Graphics as velocity. The values **rpSPLINEPATHSMOOTH** and **rpSPLINEPATHNICEENDS** can be passed to the functions **RpSplineFindMatrix()** and **RpSplineFindPosition()**. The first returns values for the developer who wants progress to be defined in parameter-space intervals that are even, and the second returns parameter-space intervals that are spaced to accelerate and decelerate towards the ends of the spline. The developer may choose to apply these to the spline's real-world.

RpSpline will support "constant velocity" only if a B-spline's control points are spaced at equal intervals.

Open and Closed B-splines

A curve can be "closed" in a continuous loop, or it may have two "open" ends.



A "closed" and an "open" **RpSpline**

Any curve can come back to its first point, but to say that a curve is "closed" usually means that the join is just as smooth as the curve at each of the other control points. This is seen in the diagram above, where the direction of the line at the start and end points changes according to whether the curve is open or closed.

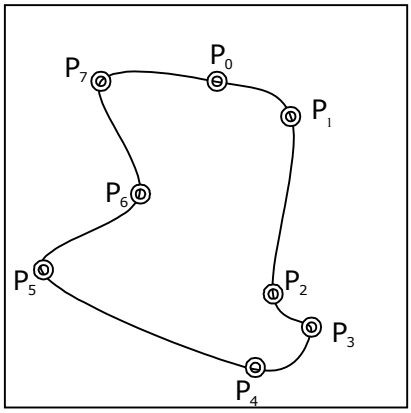
Control Points at Open Ends

RpSpline has a problem with open curves. The first and last segment do not form a sequence of four control points of which the segment runs between the middle two. **RpSpline** simplifies things by introducing two extra points. It does this behind the scenes adding an extra first and last point to each open curve, at the start and the end of the curve. These two points have exactly the same position as the first and last points, but they make it possible to apply the same "base function" to the first and last segments and provide a suitably smooth start and end of the curve.

These points are introduced and processed transparently and do not affect the control point numbering.

Closed B-spline Closed Curve Joins

In order to join a curve P_0 to P_7 in the diagram below, **RpSpline** treats P_0 and P_1 as points P_8 and P_9 as if they followed P_7 at the end of the spline. In this way it can apply the same function as it applies to all the curve segments in the spline. If the current curve segment begins at point i, then the function will take the same parameters, P_{i-1}, P_i, P_{i+1} and P_{i+2}, and this function will calculate the curve that joins the end points so that it is just as smooth as the curve between the other control points.



An *eight*-point closed **RpSpline**

At the beginning of the curve it does much the same thing, inventing a P_{-1} and P_{-2} which coincide with the last point (P_7) and the point before that (P_6).

An **RpSpline** has *n* control points, and they are numbered from P_0 to P_{n-1}. **RpSpline** calculates the last segment of a closed B-spline, the segment from P_{n-1} to P_0, by using points P_{n-2}, P_{n-1}, P_0 and P_1. Similarly the first segment P_0 to P_1 is calculated using P_{n-1}, P_0, P_1 and P_2. Thus a closed curve always joins smoothly.

26.2.4 Why Use B-splines?

There are other curve formats and they have different strengths and weaknesses, but B-splines are now well understood and their mathematical problems are largely solved. They allow us to calculate tangents. Long splines of many points are continuously smooth by their nature, without the need to smooth the joins of many shorter curve segments. And sometimes it's convenient to convert them mathematically to or from Bézier and Hermite curves.

B-splines have "local control". That means that if a control point is moved in order to move part of the spline, it affects the path of the spline no further than two control points behind or ahead in the sequence. So if point P_i is moved, the B-spline will be affected as far as P_{i-1} and P_{i-2} and in the other direction, as far as P_{i+1} and P_{i+2}. In other types of spline, the curve can be affected one or more control points further in both directions.

Other curve formats have the advantage of being easy to control by specifying their start and end points, and they move toward their off-curve control points in a way that is fairly intuitive. **RpSpline** has a greater advantage in that the control points all lie on the curve, so the spline goes exactly where the control points specify.

The convenience of making the curve join back on itself in a continuous loop without extra code or extra calculations is also an advantage.

Basic Mathematics

Earlier pages in this chapter referred to some processes that **RpSpline** performs internally. **RpSpline** uses matrix inversion to convert between its on-curve control points and the off-curve control points used in the conventional mathematics of uniform B-splines. We have seen how **RpSpline** alters the control point sequence so that the ends of loops can use the same function, so the first and last curve segments join as smoothly as the other curve segments.

In effect then, **RpSpline** can apply the same formula to all curve segments. This can be implemented in a single "basis function" or "smoothing function". The two control points in the control point sequence that enclose the curve segment are described as P_i and P_{i+1}. The basis function uses the four control points P_{i-1}, P_i, P_{i+1} and P_{i+2} to calculate any position on the spline between P_i and P_{i+1}. The value "u" increases linearly from zero to one representing positions on the spline between P_i and P_{i+1}. The formula that calculates positions on the spline is

$$\frac{P_{i+2}\,u^3 + P_{i+1}(3u^2 - 3u^3 + 3u + 1) + P_i(3u^3 - 6u^2 + 4) + P_{i-1}(3u^2 - u^3 - 3u + 1)}{6}$$

It applies varying proportions of the positions of each of the four control points to the point calculated. The amounts vary according to how far the point is from P_i to P_{i+1}.

26.2.5 How RenderWare Graphics Processes Two-dimensional B-spline Curves.

The routines to handle B-splines are contained in the **RpSpline** plugin. The **RpSpline** plugin must be attached using **RpSplinePluginAttach()** before an application can use the functionality described in this chapter. The attach function must be called after **RwEngineInit()** and before **RwEngineOpen()**. The header file **rpspline.h** must be included in files that use these functions and structures.

Struct RpSpline

RenderWare Graphics maintains a datatype, **RpSpline**, to handle each B-spline. It stores:

- file and path name (a null-terminated array of **RwChar**)
- control points (a pointer to an array of **RwV3d**)
- number of control points (an **RwUInt32**)
- and the spline type, including information like whether the spline is open or closed (an **RwUInt32**)

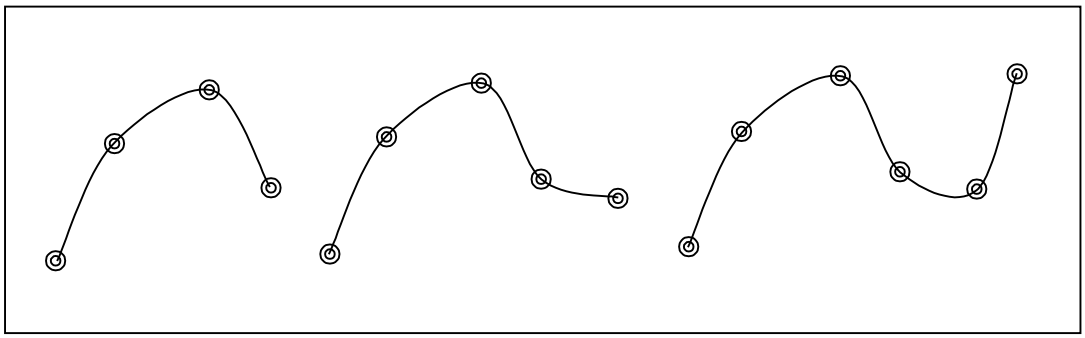
- and other data.

API functions are provided to address the structure and maintain its integrity. The developer should not address its members directly. Its members are listed in the API reference.

Creating a Spline

The function **RpSplineCreate()** takes three arguments. If it is successful, it returns a pointer to a new **RpSpline** structure.

The first argument, an **RwUInt32** gives the number of control points. This must be four or more.



B-splines defined by 4, 5 and 6 points

The second argument indicates whether the spline is to represent an open curve, or whether it is a closed loop. **rpSPLINETYPEOPENLOOPBSPLINE** and **rpSPLINETYPECLOSEDLOOPBSPLINE** specify the type of spline.

The third argument is a pointer to an array of three dimensional vectors, a *** RwV3d**, which constitutes the sequence of control points.

Cloning

RpSplineClone() provides an easy way to create or copy splines. The function takes only one argument, a pointer to the spline to clone. It returns a pointer to the new spline, which has a pointer to a new copy of the control points so that the new spline is entirely independent of the first.

A spline which has been created or cloned can have its control points adjusted by **RpSplineSetControlPoints()**. This function takes a pointer to the spline to adjust, an integer (>=0) to specify which control point is to be adjusted and a pointer to a three dimensional vector that specifies its new position. It returns a pointer to the new position, or **NULL** if it is unsuccessful.

Finding Frames, Positions and Control Points

A B-spline may be used to define a path. If it is the path of a racing car, then the developer will need the tangent of any point on the path in order to point the racing car in the right direction at that point. The function **RpSplineFindMatrix()** returns the position of the point and its direction.

RpSplineFindMatrix() takes five arguments, of which four pass data to the function and one returns it. The first argument is a pointer to the spline. The second is a bit-significant **RwUInt32** that describes the type of path as of constant velocity, with **rpSPLINEPATHSMOOTH** or with acceleration at both ends, with **rpSPLINEPATHNICEENDS**. The third is an **RwReal** that specifies how far along the path (from P_0 to P_{n-1}) the point is as an **RwReal** between zero and *n-1*. And the fourth is a pointer to the frame's look-up vector. The last argument is a pointer to the Frenet matrix that can be used to orient the frame of an object at the given point on the spline. The function returns an **RwReal** that is the Gausian description of the tightness of the spline's curvature at the point specified.

RpSplineFindPosition() finds the position in space of a point somewhere along a given B-spline. It also returns its direction at that point.

The function takes five arguments. The first three pass data to the function the other two are used to return data. The first argument is the address of the spline, the second is a bit-significant **RwUInt32** that defines the type, specifically the velocity of the path, and the third is the 'u' value from 0 to 1 which determines how far along the path (from P_0 to P_{n-1}) the specified point occurs (so the i^{th} control point is at i/n).

The path argument is an integer equal to **rpSPLINEPATHSMOOTH** or **rpSPLINEPATHNICEENDS**. The first will take the 'u' value, 0 to 1, to mean that 1/3 and 2/3 correspond to the control points on the curve, even if the first three control points are close together and far away from the fourth. The second defines the start and ends with increasing resolution which accelerates and decelerates between the first two and the last two points of the spline.

The fourth and fifth arguments are **RwV3d** vectors to receive the position of the object, expressed in world space, and the tangent of its direction. These allow the frame of any object to be found on the curve at the angle queried. The return value is the position along the curve, or null if the function fails.

The function **RpSplineGetNumControlPoints()** takes a pointer to the **RpSpline** and returns an **RwUInt32** that specifies the number of control points.

The function **RpSplineGetControlPoint()** returns the position of a specified control point on the specified spline. It takes a pointer to the spline, a zero-based integer to define the control point and a pointer to an **RwV3d** that will receive the position of the control point. The return value is the position vector.

The complement of this function is **RpSplineSetControlPoint()**, which takes a pointer to the spline, a zero-based integer to specify the control point and a pointer to an **RwV3d** vector to specify its position. It returns **NULL** on error, and the pointer to the spline if it is successful.

Destruction

The function **RpSplineDestroy()** destroys the **RpSpline** structure as it is created by **RpSplineCreate()**.

Serialization

Four functions support reading and writing **RpSpline** data. **RpSplineRead()** takes a pointer to the file's path and file name and returns a pointer to the spline data that it has read in, or **NULL** if it fails. **RpSplineWrite()** takes a pointer to the **RpSpline** that it is to write, and a pointer to the file name.

Two other functions **RpSplineStreamRead()** and **RpSplineStreamWrite()** read and write to a RenderWare Graphics binary stream. They both assume that the stream is already opened.

The user may find out how much memory an **RpSpline** occupies, to check that there is enough disk space to hold the data.

26.2.6 Spline Summary

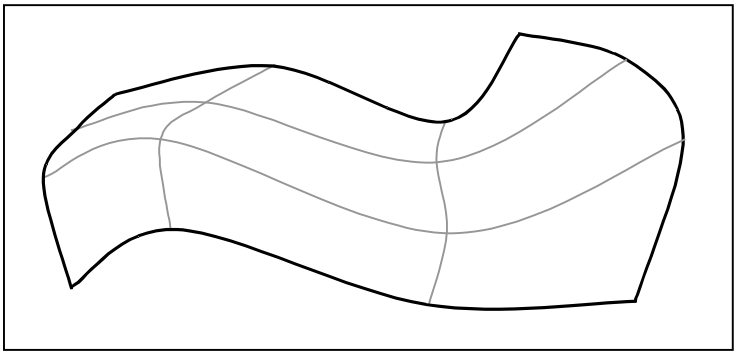
RpSpline is based on uniform B-splines. In RenderWare Graphics, they have on-curve points that appear to work as control points. They can be closed loops, in which case they are made to join smoothly automatically, and they can return their direction and position at any point. There is a full range of functions to stream them to or from files.

The rest of this chapter is concerned with patches, which are 3d shapes not curves. In RenderWare Graphics they are based on the mathematics behind Bézier curves rather than B-splines.

26.3 3D Bézier Patches

26.3.1 Introduction

One method of rendering a solid object for the computer screen is to divide the object's surfaces into polygons, usually triangles. Then the color of each visible triangle is calculated and each triangle is drawn onto the screen. This approach needs lots of triangles to represent curved surfaces and it still leaves surfaces looking unacceptably facetted when shown in close detail. RenderWare Graphics solves this by using patches.



A patch defined by eight curved lines.

The **RpPatch** routines use Bézier patches, shapes that curve in three dimensions and are bounded by Bézier curves on each side. They correspond to the simpler **RpGeometry** used elsewhere in RenderWare Graphics, that uses flat polygons bounded by straight lines.

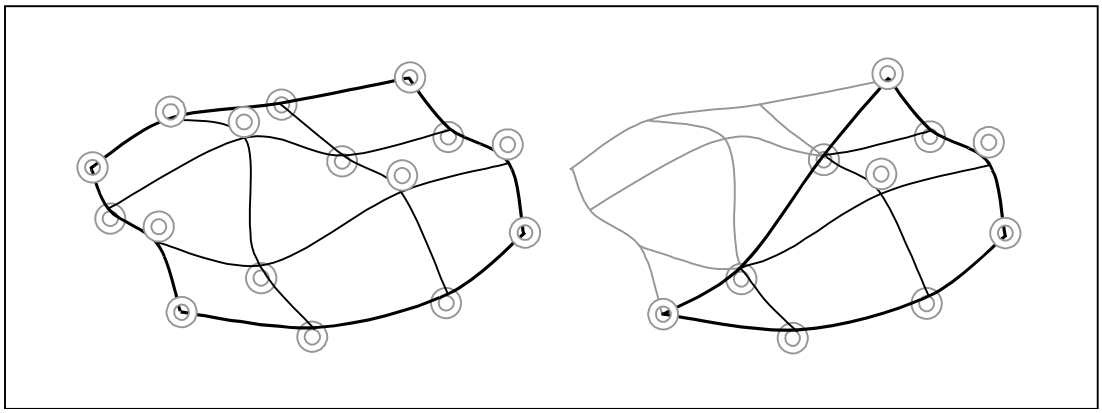
This middle section of the chapter describes how the **RpPatch** plugin implements several processes to take advantage of curved patches and integrates them with rendering processes; the developer needs to know only the broad framework of how they work.

26.3.2 What Are Patches?

In RenderWare Graphics, patches are quadrilateral or triangular shapes that curve in three dimensions. Each side of the shape is a Bézier curve. The patches are defined by control points like those which define Bézier curves and similar to the off-curve control points of B-splines. These control points define the Bézier curves that define the edges of the patch. But patches calculate the continuous surface from the control points whereas the simpler mathematics of Bézier curves need only calculate a single line.

Quad Patches and Tri Patches

A patch with four sides is a called a "quad patch". Each side is a Bézier curve and therefore has four control points. This implies four other curves (shown in the diagram in gray) linking control points on the edges. This results in four curves running across the patch in one direction, and four more crossing them in the other direction. These curves cross at 16 points on the surface corresponding to the 16 control points of the patch. RenderWare Graphics calculates the continuous surface defined by these 16 control points.



A quad patch with 16 and a tri patch with 10 control points

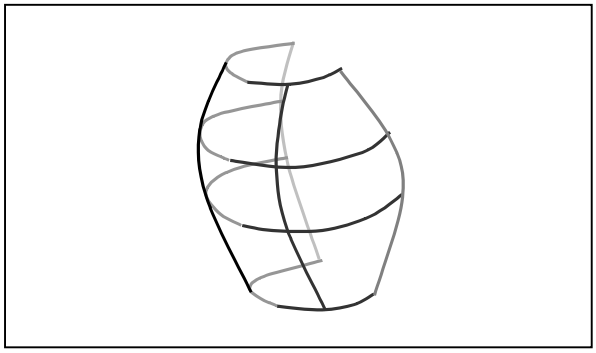
A patch with three sides is called a tri patch. Its three sides are Bézier curves, so each side is defined by four control points.

A control point at the patch's corner defines the corner's position. But the other points are not on the surface they define. So, in a quad patch, 12 control points are normally above or below the surface they define. In a tri patch seven control points are off the surface. To describe it more intuitively, it is as if the surface is attracted towards the other control points but fixed only to its corner points.

26.3.3 Why Use Patches?

It is clear that patches are more complicated than flat polygons. They have more control points and the control points are inter-related. They need more complicated mathematics and more calculations to derive useful information from them. The code to do this is proportionately harder to develop and to integrate with the rest of the rendering system. Different platforms support patches to very different extents, and RenderWare Graphics must accommodate the differences to support each of them efficiently. So, why use patches?

Each patch takes more data to define it, but a single patch can replace very large numbers of flat polygons, reducing storage space and processing time.



An example of a single patch that would take many triangles to approximate

In practice, these patches are rendered as small triangles, but it is the graphics processor that refines them, so it doesn't slow the main processor or use excessive amounts of memory.

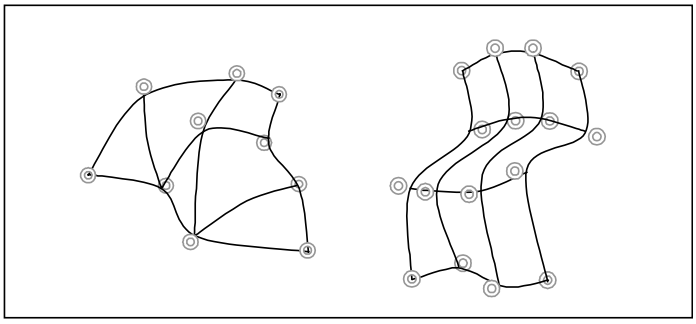
Patches have another advantage; they can be divided into many small triangles for rendering, or, just as easily, into a few big triangles. So they can be rendered at varying levels of detail. Usually software improvements involve a trade-off. But introducing patches allows RenderWare Graphics to reduce render time and at the same time to reduce memory usage and still improve visual quality, all to produce the optimal visual effect.

It is possible that future graphics processors will handle patches directly, either as curved surfaces, or by splitting them into facets. If they do then patches will give even greater efficiency to the rasterization process.

26.3.4 How RenderWare Graphics Handles Patches

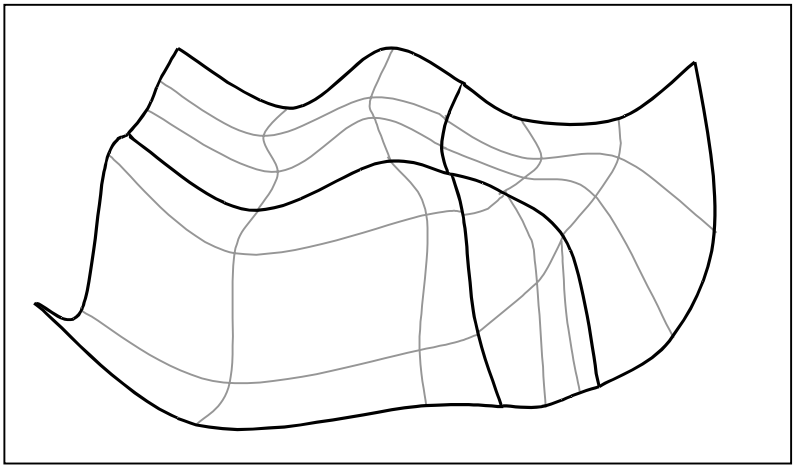
RenderWare Graphics defines two types of patch:

- **RpQuadPatch** represents a quad patch as 16 indices to control points
- **RpTriPatch** represents a tri patch as 10 indices to control points



An tri patch with 10, and a quad patch with 16 control points.

Both types of patch are stored in an **RpPatchMesh** to record the way the patches fit together. A single patch describes only a simple curved surface, but when patches are fitted together in a patch mesh they can define complicated surfaces.



Three quad patches and one tri patch in a patch mesh.

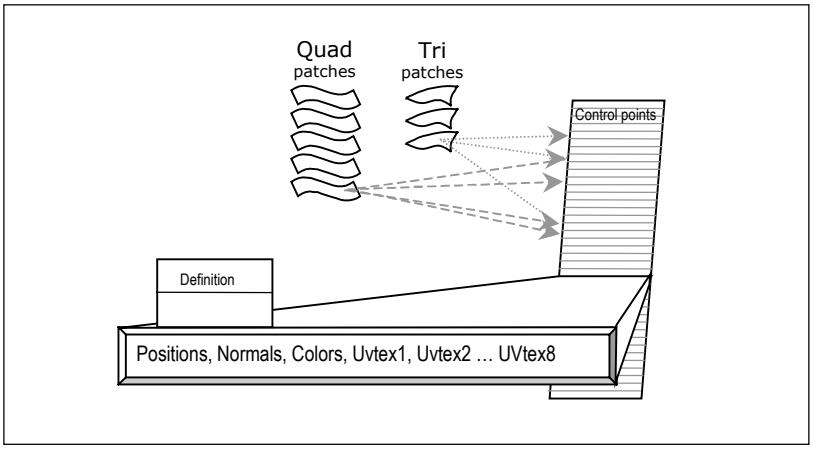
In RenderWare Graphics, these control points' coordinates are held in the "positions" array in an **RpPatchMesh**.

Quad patches and tri patches are arrays of **RwUInt32** indices that refer to an unknown array. The patches will only be meaningful when they specify which array they refer to. However, both quad and tri patch data types must be defined separately before they are added to a patch mesh by the function **RpPatchMeshSetQuadPatch()** or **RpPatchMeshSetTriPatch()**.

The Patch Mesh

The **RpPatchMesh** is similar to the **RpGeometry** that was used to store triangular facets in the chapter on *Dynamic Models*.

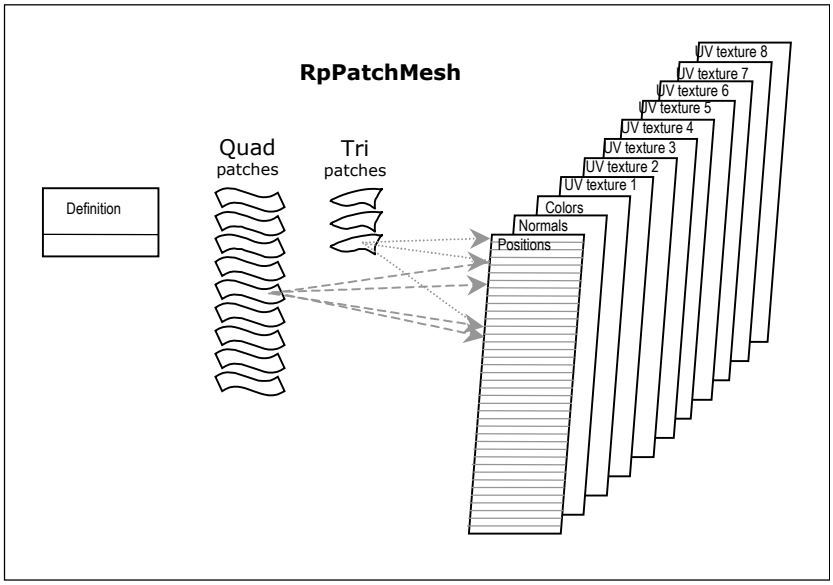
The patch mesh can be seen in two ways. From a high level, and ignoring the detail of the code for a moment, it is an array of tri patches and an array of quad patches which index an array of control points.



High-level view of **RpPatchMesh**

The control point in this sense is flexible. It has its own **RpPatchMeshDefinition** structure and that determines which data the control point holds. This may include the coordinates of its position, its pre-calculated normal vector, its pre-light color (its color before it is lit) and various sets of coordinates to stretch textures over it. But these can be present in any combination that the developer requires. (In principle none of them needs to be present.) This is the underlying concept and the most helpful way to see the patch mesh. In this sense a control point would probably have positions and normals to define the shape of the patch mesh. However, the same patch mesh could be defined with its control points containing only positions, if the normal vectors are not required.

The C language does not represent **RpPatchMesh** so economically, as is shown in the diagram below. The diagram illustrates only part of the **RpPatchMesh** and much of it is "opaque", or hidden from the developer. Where API calls are provided to access it, the developer should use them to maintain the consistency of data (and because the API functions will assert if they detect invalid data).



A more literal representation of **RpPatchMesh**

A more literal explanation is that the **RpPatchMesh** contains an **RpPatchMeshDefinition** structure, arrays of quad patches and of tri patches, and up to 11 parallel arrays.

They are:

- a structure called "**RpPatchMeshDefinition**" explained below
- an array of quad patch control point indices (in groups of 16) each control point being an index into the array of positions
- an array of tri patch control point indices (in groups of 10). Again, each control point is an index into the array of positions

  Each of the remaining items is optional:

- an array of control point coordinates called positions
- a parallel array of the control points' normal vectors
- a parallel array of the control points' pre-light colors: these are the base colors of the patch control points
- and up to eight parallel arrays of which each element consists of a UV texture coordinate set, to map a texture to each control point.

RpPatchMeshDefinition

The **RpPatchMeshDefinition** holds some binary settings in **RpPatchMeshFlag**. It also holds the maximum number of control points, the maximum number of tri patches, the maximum number of quad patches and the number of arrays of texture coordinates (0 to 8) and the **RpPatchMeshFlag**. These numbers are used to calculate and reserve the amount of memory required when an **RpPatchMesh** is created. The number of control points is used to determine the size of the array of control point coordinates and thus the size of each parallel array when the developer needs to address them.

RpPatchMeshFlag

The user defines the flags by passing them to the Create function which stores them in the **RpPatchMeshDefinition** structure. The flags can also be accessed by **RpPatchMeshSetFlags()** and **RpPatchMeshGetFlags()**.

Most of the **RpPatchMeshFlag**s indicate which of the arrays are provided. If the flag is set, then memory is reserved for its array and it exists; if it is clear then no memory is reserved for it and it does not exist. The last three flags describe other aspects of the mesh:

- **rpPATCHMESHPOSITIONS**: the control points have positions (i.e. there is an array of control point positions)
- **rpPATCHMESHNORMALS**: the control points have normals (i.e. there is an array of control point normals)
- **rpPATCHMESHPRELIGHTS**: the control points have pre-light colors (i.e. there is an array of control point pre-light colors)
- **rpPATCHMESHTEXTURED**: the control points have texture coordinate sets (i.e. there is at least one array of control point texture coordinates) (If this flag is set, then the macro **rpPATCHMESHTEXCOORDSETS()** must be used to specify how many)
- **rpPATCHMESHLIGHT**: the mesh will be lit by lights in the **RpWorld**
- **rpPATCHMESHMODULATEMATERIALCOLOR**: the patches' color will be modulated with the material color
- **rpPATCHMESHSMOOTHNORMALS**: Where adjacent patches do not join smoothly this restores smooth shading and disguises the join

Two of these flags need a little more explanation.

The flag **rpPATCHMESHLIGHT** is very similar to the **rpGEOMETRYLIGHT** flag which is an enumerated value of the **RpGeometry** flag in the *Dynamic Models* chapter. If light effects are not applied, the pre-light colors are used unmodified, which reduces rendering time significantly. This light effect is well illustrated in the "**patch**" example: the sail and the teapot become brighter as they turn towards the light.

The flag **rpPATCHMESHSMOOTHNORMALS** is designed to solve the problem of adjacent patches that should join smoothly but whose shading does not. This arises when the normal vectors on two patches are not pointing in the same direction on the edge where they join. The brightness of directional light falling on the patches is calculated from the normal vectors, so the discrepancy shows up as an obtrusive edge in the shading, like a crease in a sheet of paper. When the **rpPATCHMESHSMOOTHNORMALS** flag is set, RenderWare Graphics searches the patch mesh for shared control points on shared edges and it checks to see if their normal vectors conflict. Whenever it finds one it alters the conflicting vectors to their average value. This gives smooth shading across the join.

The flag **rpPATCHMESHTEXTURED** indicates that there is at least one UV texture coordinate set present. The macro **rpPATCHMESHTEXCOORDSETS(***<number>***)** must be called to set the number of UV texture coordinate sets, between 1 and 8, into the flags field.

In the **RpPatchMesh**, the control point positions are stored separately and indexed by the patches so that patches can share control points. This saves memory and makes it quicker to alter the control point data.

The mesh can hold up to eight arrays, but the number of arrays that can be *used* will be platform dependent. Xbox supports four UV texture coordinate set arrays, GameCube supports eight and all platforms support two.

There is more data and functionality in the **RpPatchMesh**, but this outline covers the features that the developer needs in order to use the API functions.

26.3.5 How To Use Patches

The pages that follow describe the steps to create and use **RpPatchMesh**. Example code found in **examples\patch** is provided showing how patches are integrated into an example application that will compile and run, and this example is discussed under the heading *Example Code* at the end of this section.

The sequence of steps to use a patch mesh correspond closely to the sequence for building an **RpGeometry** described in the chapter on *Dynamic Models*. The reader should refer to that chapter if the concepts of **RpGeometry**, **RpAtomic**, **RpClump**, **RpMaterial**, **RwTexture** and **RwTexCoords**, pre-light colors and material color are not familiar.

Attaching the Plugin

Before patch meshes are supported, the **RpPatch** plugin must be attached by calling the function **RpPatchPluginAttach()**.

Creating a Patch Mesh

The **RpPatchMeshCreate()** function returns a pointer to the patch mesh it creates. It takes as arguments the number of tri patches, the number of quad patches, the number of positions and **RpPatchMeshDefinition**. With these values **RpPatchMeshCreate()** calculates the memory required for each of the arrays at the size specified, reserves enough memory for these and the **RpPatchMeshDefinition** structure and returns a pointer to the **RpPatchMesh**.

The **RpPatchMesh** also writes the flag field which is a member of its **RpPatchMeshDefinition** structure. The function **RpPatchMeshCreate()** sets the mesh's locking flags to ensure that it is created in its locked state. This means that patches, positions, normals, colors and texture coordinates can be written into it. (The concepts of locking and unlocking are described later.)

After calling the **RpPatchMeshCreate** function the mesh holds data only for the flag settings and array sizes.

The Patch Mesh's Flags

RpPatchMeshCreate() must set the mesh's flags to determine which arrays will be present in the **RpPatchMesh**. These bit settings are described above, under **RpPatchMeshFlag** and further documented in the API Reference.

The flag **rpPATCHMESHTEXTURED** indicates that there is a number of UV texture coordinate sets present. If **rpPATCHMESHTEXTURED** is set, then the macro **rpPATCHMESHTEXCOORDSETS()** must be called to define the number of texture coordinate sets present, and enter it in the flags field. The number in the example below, must be between 1 and 8. For example:

```
RpPatchMeshCreate( quads, tris, positions,
                   rpPATCHMESHPOSITIONS    |
                   rpPATCHMESHNORMALS      |
                   rpPATCHMESHTEXTURED     |
                   rpPATCHMESHTEXCOORDSETS( <number> ) )
```

RpPatchMeshSetFlags() is provided in case the developer wants to alter the flags later. They can be read with **RpPatchMeshGetFlags()**.

Destroying a Patch Mesh

When the patch mesh is no longer needed, **RpPatchMeshDestroy()** releases the memory reserved by **RpPatchMeshCreate()**.

Locking and Unlocking the Mesh

Before a patch mesh can be rendered, RenderWare Graphics must transform patches, copy and re-sort them into fast internal format and render them. This uses the processor heavily and takes extra memory. The locking functions allow the developer to control and reduce this demand on resources. When a patch mesh is "unlocked" with **RpPatchMeshUnlock()** RenderWare Graphics gets the chance to do the extra processing. The developer must not alter the arrays that define the positions, normals, colors or texture coordinates when the patch is unlocked. When it is "locked", with **RpPatchMeshLock()** the arrays of positions, normals, colors and texture coordinates can be updated and RenderWare Graphics will not attempt to process them until they are in a complete and consistent state, indicated by calling **RpPatchMeshUnlock()** again.

When a patch mesh is created its arrays hold no data, and the locked state leaves it ready to accept data in all its arrays. At other times the **RpPatchMeshLock()** can be used. This takes, as an argument, any of the enumerated values below which refer to the arrays individually. This mechanism reduces the processing time still further by telling **RpPatchMesh** which data it can safely ignore because it, or the values from which it is derived, have not been changed.

The enumerated **RpPatchMeshLockMode** values (which can be combined with the "or" operator) are:

- **rpPATCHMESHLOCKPATCHES**
- **rpPATCHMESHLOCKPOSITIONS**
- **rpPATCHMESHLOCKNORMALS**
- **rpPATCHMESHLOCKPRELIGHTS**
- **rpPATCHMESHLOCKTEXCOORDS1**
- **rpPATCHMESHLOCKTEXCOORDS2**
- **rpPATCHMESHLOCKTEXCOORDS3**

  . . .

- **rpPATCHMESHLOCKTEXCOORDS8**
- **rpPATCHMESHLOCKTEXCOORDSALL**
- **rpPATCHMESHLOCKALL**

Although **RpPatchMeshLock()** may be called with different values to lock individual arrays in the mesh, **RpPatchMeshUnlock()** unlocks all of them. The **RpPatchMeshUnlock()** function enables the routines to recalculate and to prepare the mesh for refinement.

Filling the Positions Array

The array of positions is a simple array of **RwV3d**s that hold the coordinates of the control point positions. **RpPatchMeshGetPositions()** returns the address of the array and the developer addresses the array directly. The number of **RwV3d** vectors in the array is the same as the number of control points, and can be found with **RpPatchMeshGetNumControlPoints()**.

```
RwV3d * const positions =
                    RpPatchMeshGetPositions(&<PatchMesh>);
int i;
for( i=0; RpPatchMeshGetNumControlPoints(&<PatchMesh>); i++)
{
            positions[i] = <source>;
}
```

Filling the Patches Arrays

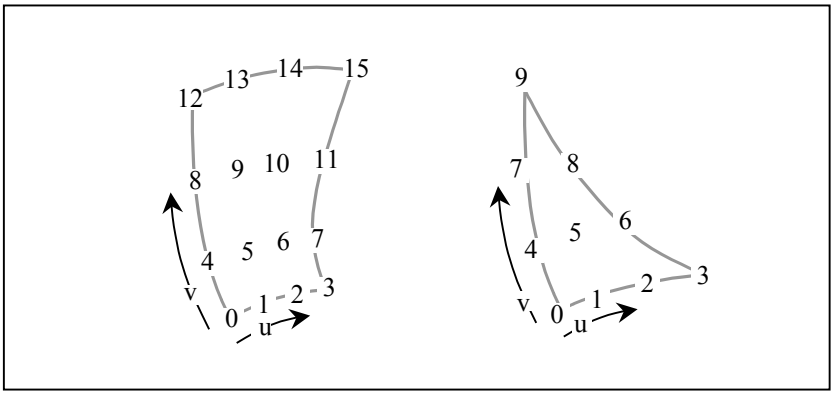
The two functions **RpPatchMeshSetQuadPatch()** and **RpPatchMeshSetTriPatch()** copy patches into their respective arrays. These arrays should be treated as opaque, and the developer cannot access them directly and must use the functions provided.

The developer trying out these routines for the first time will probably make each new **RpPatchMesh** index a new set of control point indices.

It is often desirable for identical control points to share the same index, but this is not always possible. For instance two patches may join at four control points, but if they join along an edge, like a fold in a piece of paper, their control points will have different normal vectors. Some patches may join on common control points but have different pre-light colors or different textures. So control points can only share an index if the information in every one of their parallel arrays is identical.

The **RpQuadPatch** or **RpTriPatch** must be defined first.

Each **RpTriPatch** or **RpQuadPatch** is an array of indices to **RwV3d** vectors in the positions array. Each patch's control point zero will be used as the point where u=0 and v=0. Control point one will be where u=1/3 and v=0 and so on, as illustrated. These *u* and *v* values become significant in the Bézier Toolkit functions described in the last section of this chapter.



Order of control points for a patch

```
RpQuadPatch quadPatch;
quadPatch[0] = <controlpointindex> ;
quadPatch[1] = <controlpointindex> ;
. . .
RpPatchMeshSetQuadPatch( <&patchmesh>, <patchindex>,
                                                    &quadPatch);
```

The functions **RpPatchMeshSetQuadPatch()** and **RpPatchMeshSetTriPatch()** take three arguments,

- a pointer to the patch mesh to add the patch to
- the index of the element in the array to copy the patch into
- a pointer to the patch to be copied.

Filling the Normals Array

The normals array, like the positions array, is a simple array of **RwV3d**s that holds the normal vectors at each control point. The developer should address it directly, getting its address from **RpPatchMeshGetNormals()**. The number of **RwV3d** vectors in the array is the same as the number of control points and is returned by **RpPatchMeshGetNumControlPoints()**. Each **RwV3d** appended to the array of normal vectors must have the same index as its coordinate in the array of positions. So the process of filling this array is almost the same as that for the positions array, and its code sample, under Filling the Positions Array, above.

The calculation of normal vectors can be done by the function **RtBezierQuadMatrixGetNormals()** which is part of the Bézier Toolkit, documented in the next section of this chapter. It calculates the normal vectors for each position vector in its second argument and returns them in the respective vectors of its first argument.

Filling the Colors Array

The colors array stores the default color of a patch at each control point. This is the color in which it will be rendered in an **RpWorld** with no light, fog or other effects. (The **RpPatchMeshFlag rpPATCHMESHPRELIGHTS** must be set for pre-light colors to be brought into effect.)

Like the positions and normals arrays it is addressed directly, and the function **RpPatchMeshGetPreLightColors()** returns the start address of the array. The number of **RwRGBA** values in the array is the same as the number of control points and is returned by **RpPatchMeshGetNumControlPoints()**.

Filling the UV Textures Set Arrays and Materials

UV Texture Coordinate Sets

RpPatchMeshGetTexCoords() returns the start address of the arrays of **RwTexCoords** values. The **RpPatchMeshDefinition** stores the maximum possible number of texture coordinate set arrays that **RpPatchMesh** supports (the number that the hardware can use is platform dependent). So, unlike the functions for the other parallel arrays, this function must specify which array the pointer is to refer to, by passing an **RwTextureCoordinateIndex** argument (>=1) to request the address of array 1 up to array 8.

```
RpPatchMeshGetTexCoords(<&patchmesh>,<whichtexcoordarray>);
```

Apart from this, the texture arrays are addressed in the same way as the parallel arrays of positions, normals and colors. The developer must access the texture coordinate set array directly, calling the **RpPatchMeshGetTexCoords()** to return the base address of the appropriate texture coordinate set array. The number of UV coordinates, or **RwTexCoords**, in the array is the same as the number of control points and can be found with **RpPatchMeshGetNumControlPoints()**.

The coordinates refer to the UV coordinates of the **RpMaterial** that is to be mapped to the patch. RenderWare Graphics stretches the **RpMaterial** over the patch so that the material's UV coordinates coincide with their respective coordinate positions in the patch mesh as specified by this array. The process is similar to that used on the facets of a geometry described in the *Dynamic Models* chapter.

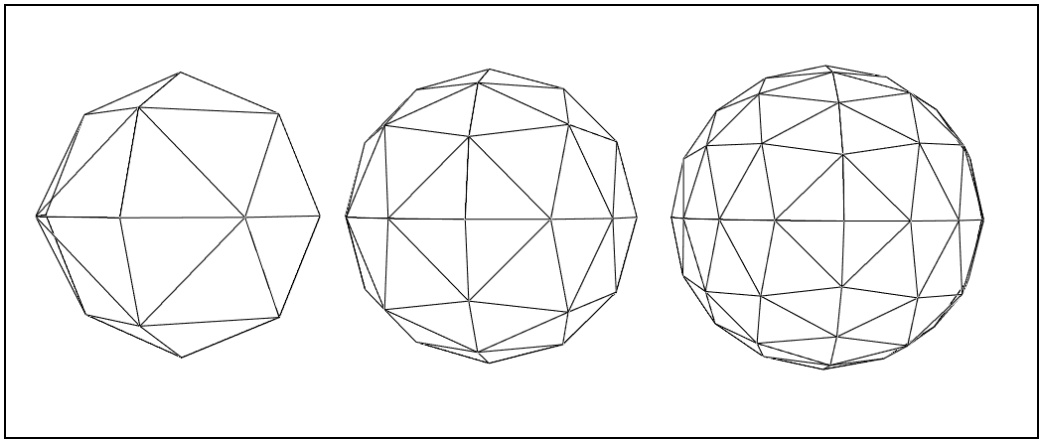
Materials

Each patch can have an **RpMaterial** allocated to it by the functions **RpPatchMeshSetQuadPatchMaterial()** or **RpPatchMeshSetTriPatchMaterial()**. The function **RpPatchMeshGetNumMaterials()** returns the number of unique **RpMaterial**s stored in the whole patch mesh.

Two functions, **RpPatchMeshGetQuadPatchMaterial()** and **RpPatchMeshGetTriPatchMaterial()** return a pointer to the material associated with the specified patch.

Setting the Level Of Detail (LOD)

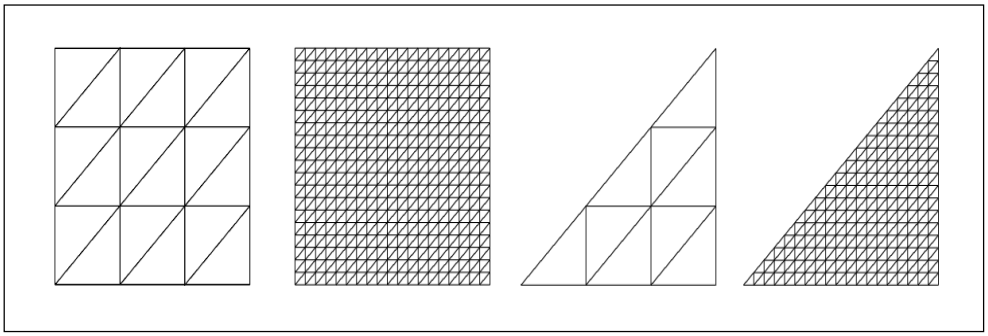
Objects may be rendered to different LODs. The greater the number of triangles that are used to represent an object, the greater the detail, and generally the more processor time and memory are used.



A ball represented at different levels of detail

Objects generally need only a few triangles to represent them when they occupy half a dozen pixels, and generally need many more triangles when they fill the screen.

Once every frame, RenderWare Graphics calls a default function that calculates the distance between an atomic and the camera. By default, this is used to determine the LOD at which the object is facetted.



The sides of quad patches and tri patches with 4 and with 20 divisions, to form flat quadrilaterals and triangles

The LOD is represented internally by an integer value between 4 and 20, in inverse proportion to the object's distance from the camera. The number represents the number of divisions of each side of each patch to split it into facets. If the LOD is 4, then each quad patch is split by 4 divisions vertically, creating 3 columns, and it is split by another 4 divisions horizontally, creating three rows. Thus 9 facets, (3 rows and 3 columns) are produced by an LOD of 4. The facets are split into two triangles each making 18 triangles for the quad patch.

If the LOD is 20, then each quad patch is divided into 19 rows and 19 columns, making 361 facets and 722 triangles. These two values are **#define**d as **rpPATCHLODMINVALUE** (4) and **rpPATCHLODMAXVALUE** (20). The number four is chosen as the lowest LOD as it requires no extra calculation; the four control points of the Bézier curves that bound each patch already divide the patch into four.

The same applies if the patch mesh is used for skinning but the maximum value is **rpPATCHSKINLODMAXVALUE** which is **#define**d as 18.

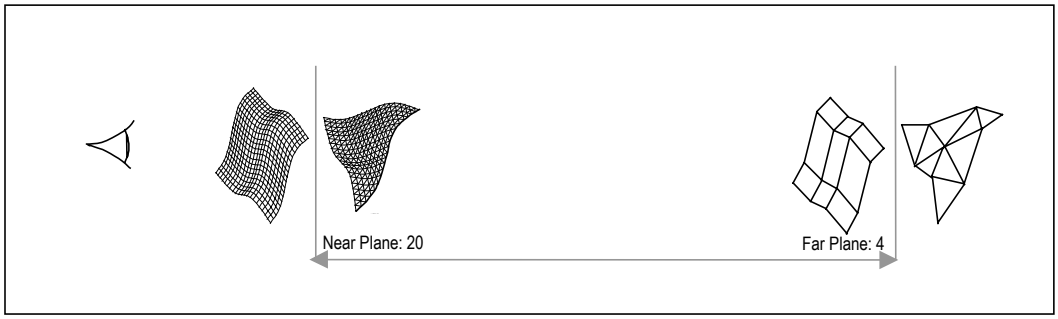
The structure **RpPatchLODRange** stores four values:

- **RwUInt32** **minLOD** minimum number of facet divisions (default 4)
- **RwUInt32** **maxLOD** maximum number of facet divisions (default 20)
- **RwReal** **minRange** minimum distance from camera
- **RwReal** **maxRange** maximum distance from camera

The function **RpPatchSetDefaultLODCallBackRange()** takes this structure as its argument, and copies these values into the structure used by the current LOD call back function. And **RpPatchGetDefaultLODCallBackRange()** returns a pointer to the current **RpPatchLODRange**.

The default function provided by RenderWare Graphics to calculate the required LOD checks that the distance value falls between **minRange** and **maxRange**, altering it if necessary. Then it calculates an LOD value which falls between **minLOD** and **maxLOD** in the same ratio as the distance fell between **minRange** and **maxRange**.

There are limitations to this function. It produces the same number of facets for large as for small patches, for deeply curved patches and for almost flat ones. So RenderWare Graphics allows developers to supply their own custom functions to be called in place of the default function. The custom function's address can be passed, in place of **NULL**, to the callback specifier function, **RpPatchAtomicSetPatchLODCallBack()**. This function and **RpPatchAtomicGetPatchLODCallBack()** set and return the address of the developer's function.



LOD varies from 20 to 4 depending on
an object's distance from the camera

Exposing the Pipeline

Attaching the **RpPatchMesh** to an **RpAtomic** is not sufficient to render the patch mesh instance. The rendering pipeline attached to the **RpAtomic** needs to be overloaded with a custom patch pipeline. The default rendering pipeline knows nothing about rendering **RpPatchMesh**es and will try to render the **RpAtomic** as if a triangle mesh is attached.

The **RpPatchType** enumeration lists the different rendering pipeline *types* available within the **RpPatch** plugin. At present these are:

- **rpPATCHTYPEGENERIC** – pipeline renders generic patch meshes
- **rpPATCHTYPESKIN** – pipeline renders skinned patch meshes
- **rpPATCHTYPEMATFX** – pipeline renders material affected patch meshes
- **rpPATCHTYPESKINMATFX** – pipeline renders skinned material affected patch meshes.

A patch mesh rendering pipeline is attached to the **RpAtomic** with **RpPatchAtomicSetType()**. The type of the present patch mesh rendering pipeline can be queried from an **RpAtomic** with **RpPatchAtomicGetType()**.

RpPatch Libraries:

rppatch.lib, **rppatchskin.lib**, **rppatchmatfx.lib** and **rppatchskinmatfx.lib**.

There are presently four versions of the **RpPatch** libraries in the RenderWare Graphics SDK. They are all fully featured versions of the **RpPatch** plugin and they contain identical APIs. However, because the rendering pipelines are large, different versions have been compiled so that the user can select precisely the pipelines they will be using to link against.

The **rppatch.lib** library only contains the **rpPATCHTYPEGENERIC** pipeline.

The **rppatchskin.lib** library contains *both* the **rpPATCHTYPEGENERIC** and **rpPATCHTYPESKIN** pipelines.

The **rppatchmatfx.lib** library contains *both* the **rpPATCHTYPEGENERIC** and **rpPATCHTYPEMATFX** pipelines.

The **rppatchskinmatfx.lib** library contains all the pipelines, **rpPATCHTYPEGENERIC**, **rpPATCHTYPESKIN**, **rpPATCHTYPEMATFX** and **rpPATCHTYPESKINMATFX**.

Only one of the patch libraries should be used in an application at once.

Patch Mesh Serialization

The functions **RpPatchMeshStreamRead()** and **RpPatchMeshStreamWrite()** are provided to read or write an **RpPatchMesh** to or from the stream. The pointer if the stream, which must have been opened first, is passed to the function.

The stream functions are often used to load the patch meshes as they are exported from graphics applications. (The **patch** example shows the data typed into the code for maximum transparency but this is not the way it would be done normally.)

The size of the patch mesh must be stored in its header in the stream. The size of the data in memory, excluding the header size, can be found by calling **RpPatchMeshStreamGetSize()** which returns its size in bytes.

If the user streams out any **RpClump** or **RpAtomic** that contains an **RpPatchMesh**, then either object will automatically stream out any **RpPatchMesh** that it contains.

See the chapter on *Serialization* for a fuller treatment of streaming.

Skinning

Patch meshes can be skinned. The **RpSkin** plugin attaches in the same way as other plugins. The API provides one Get function and one Set function to add skinning: **RpPatchMeshSetSkin()** and **RpPatchMeshGetSkin()**. RenderWare Graphics integrates the process so that it works in the same way as it did with **RpGeometry**s as described in the chapter on *Dynamic Models*.

Transforming a Patch Mesh

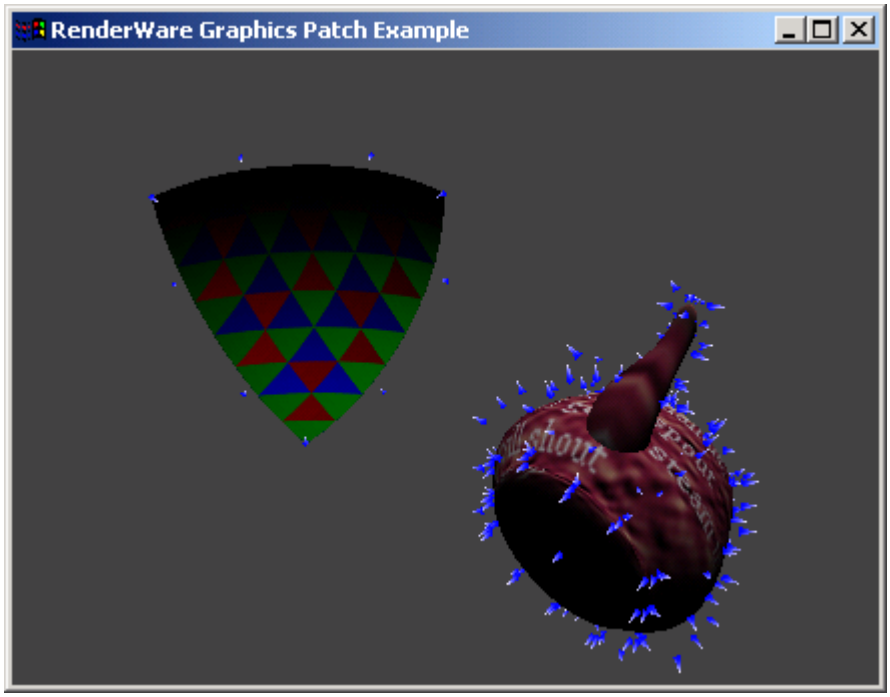
RpPatchMeshTransform() performs a transformation on a patch mesh. It takes a pointer to an **RpPatchMesh** and a pointer to a transformation matrix, and it applies the transformation matrix to all control point positions and normals in the patch mesh. It returns a pointer to the altered **RpPatchMesh** on success, or **NULL** on failure. The mesh is locked automatically before and unlocked after the transformation as required, and the mesh's bounding sphere is recalculated before it finishes.

Patch Meshes and their Atomics

RenderWare Graphics manages each patch mesh through an **RpAtomic**. So each patch mesh should be attached to an **RpAtomic** just as the earlier chapter on *Dynamic Models* needed each **RpGeometry** to be attached to an **RpAtomic**. The **RpPatchMesh** holds all the details of the object's shape. The details of its orientation are stored in a frame which is pointed to from the same **RpAtomic**. The function **RpPatchAtomicSetPatchMesh()** attaches the **RpPatchMesh** to an **RpAtomic**. And **RpPatchAtomicGetPatchMesh()** retrieves a pointer to the patch mesh attached to an **RpAtomic**.

An **RpPatchMesh** may be attached to more than one **RpAtomic**.

26.3.6 Example Code



Screenshot from: **examples\patch**.

The previous section, on RpPatchMesh has drawn attention to some features of the example, **patch**, and to its code. The key sections are in **main.c** and **patch.c** of the example code in **examples\patch**.

The code has been written to echo the chapter on *Dynamic Models* which describes how to build a matrix of flat triangles to render a solid object. The example code follows the same sequence closely, but uses it to build a matrix of patches that are curved. Only the first step in the sequence, attaching the plugin, is added for patch meshes.

The example includes two objects, a teapot and a triangular sail, to show both quad patches in the teapot, and tri patches in the sail.

The example code shows how to build a patch mesh's positions and patches arrays. The sail uses only tri patches, the teapot is built entirely of quad patches. The tri patch code is easier to follow. The process takes only four lines of code. An **RtBezierMatrix** called **triControl** is defined at the head of the function **PrimePatchMesh()**. Much later, after the comment **/* Tri Control Points */**, the coordinates of ten points on a section of an imaginary sphere are copied into **triControl**. These points on the surface of the sphere are converted to control points (3 on the surface, 7 off the surface) by the **RtBezPat** Toolkit function, **RtBezierTriMatrixControlFit()**. The tri patch is converted to a functionally equivalent quad patch by **RtBezierQuadFromTriangle()**. This function takes two **RtBezierMatrix** arguments. The second is the source data, **triControl**, and the first receives the coordinates of a quad patch whose normal vectors are now set.

The values for the teapot's vertices are held in the array, **QuadpotVertex[]**, also in **patch.c**. Notice that it has been decided beforehand which coordinates can be shared in the positions array, and also which position values will be stored at which index. There is nothing in the code nor in the plugin's functions to make these decisions or simplify this process. Only the developer can know enough to decide.

The example code will give the reader a better understanding of the way these functions are used together in RenderWare Graphics and how to draw up some working code. Some of the functions that simplify this process belong to the Toolkit which is described in the next section of this chapter.

26.3.7 Summary

RpPatchMesh hides complex functionality behind a small number of API calls. They allow the speed and detail of curved patch rasterization to integrate with the features of materials, skinning, animation and other rendering processes already supported. The next section explains some ways in which the plugin has been optimized for efficiency reducing processing times dramatically. The present section shows how the developer can further reduce the overhead of this extra functionality by locking and unlocking the **RpPatchMesh** structure between frames.

26.4 Bézier Toolkit

26.4.1 Introduction

The **RtBezier** Toolkit is a group of mathematical functions that serve the **RpPatchMesh** in the **RpPatch** plugin. They are also useful in their own right, for instance in fitting a Bézier patch to an arbitrary surface.

The patch mesh is designed to simplify the rendering of shapes at different scales and different LODs to a much higher standard than was possible with facetted triangle meshes. But some of the data derived for patches has much more potential.

RenderWare Graphics uses the normal vector of a point on a patch's surface to find how much directional light that point receives. The vector can also be used to find how the direction in which missiles will bounce off the surface, how the surface reflects light, and what it will reflect if it is shiny. So this functionality is exposed to the developer in the Toolkit.

RpPatchMesh supports a relatively thorough form of rendering with great efficiency, but games designers often invent objects and characters precisely because they can be represented more simply. Having access to lower level code allows the developer more flexibility in manipulating Bézier patches.

The **RtBezPat** Toolkit allows the developer to use RenderWare Graphics' tried and tested routines, and they can be used selectively to support methods of rendering appropriate to unusual objects, allowing the developer to concentrate on new code.

The Toolkit functions fall into four groups. One utility function is distinct and converts tri patches into quad patches. A second group converts between control points and points on the surface of the patch. A third large group applies forward differencing to speed up calculation, and the fourth group calculates tangents and normal vectors for a whole patch.

Three data types are used throughout the toolkit and are explained below. Before that is an explanation of the concepts of parameter space and real-world space which underlie much of the code.

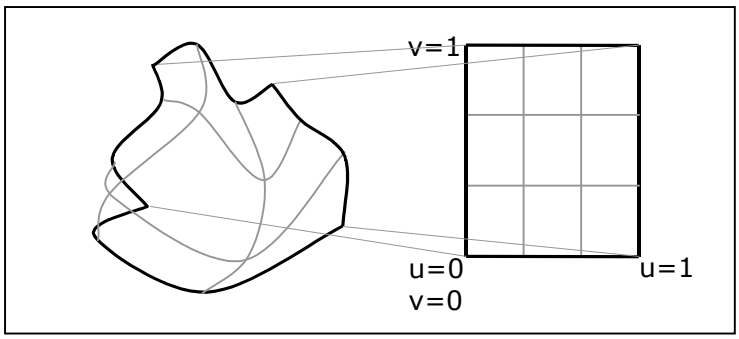
Parameter Space and Real-world Space Revisited

These concepts of parameter space as opposed to real-world space occurred in the earlier section on B-splines. They apply in three dimensions as well as in two.

A patch can be seen in two ways.

- It is a three-dimensional surface, bounded by curves whose control points are bunched or dispersed at irregular intervals. This is referred to as "real-world space".

- It is also referred to by its UV coordinates which progress linearly from zero to one, as on a simple graph. This is referred to as "parameter space". These UV coordinates map to the curved, bunched, non-linear progress of coordinates along the curves on the surface of the patch.



A patch can be thought of as a curved surface in space, but also as a set of linear UV values, like a graph.

The "*u*" and "*v*" coordinates referred to in this section refer to evenly spaced measurements of the parameter space coordinates, where control points are spaced along the axes at 0, 1/3, 2/3 and 1. We can refer to a point 1/3 of the way along a curved edge, meaning that it falls on a point on the surface corresponding to the first control point. This is obvious in parameter space. But if P_0, P_1, and P_2 are close together, and far away from P_3, then P_1 will be much less than 1/3rd of the way along the curve in real-world space. `RenderWare Graphics` refers to both real-world space and parameter space and some of the functions that follow translate between them.

Because the word "parameter" has this particular meaning in this section of the chapter, the values passed to functions are described, in this section, as "arguments" rather than "parameters".

26.4.2 Data Types

The Bézier Toolkit functions use three data types not visible in the higher level code described earlier in this chapter. **`RtBezierV4d`** is a `RenderWare Graphics` vector of four dimensions. They are *x*, *y* and *z*, used as 3d coordinates, and *w*. The *w* argument is interpolated in the same way as the others and can usefully process values of quite different types. For instance, a lighting value might be interpolated to correspond to its position on the curved surface.

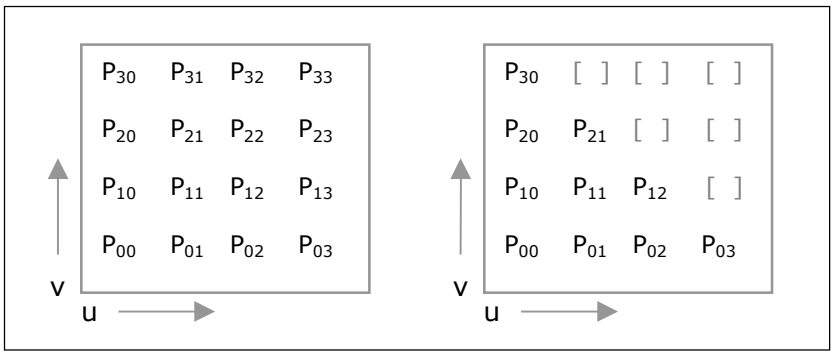
An **`RtBezierV4d`**can represent any of the control points on a quad patch.

RtBezierRow

An **`RtBezierRow`** is an array of four **`RtBezierV4d`**s. They typically represent a single row of the control points that define a patch at *u*=0, *u*=1/3, *u*=2/3 and *u*=1.

RtBezierMatrix

An **RtBezierMatrix** is an array of four **RtBezierRows** one for each position across a patch, where v=0, v=1/3, v=2/3 and v=1. So it contains an **RtBezierV4d** for each control point in a quad patch, in a known order.

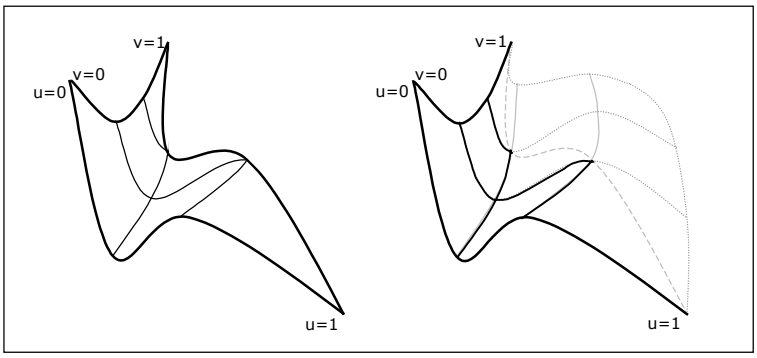


Positions of control point numbers for quad and tri patches in an **RtBezierMatrix**

A **RtBezierMatrix** can store a range of other data. It can hold a tri patch, as the vertex u=0, v=0 is known to be in the first row, in the first vector of the matrix, so it knows which ten vertices are part of the tri patch and their order. The vectors can specify the coordinates of the points of the surface of the patch that correspond to their respective control points. They can specify the normal vectors at each respective surface point, or vectors of tangents to the respective surface points. They can store the "difference" values of the control points or surface points. So **RtBezierMatrix** is flexible and used in various ways in the Toolkit functions.

26.4.3 Quad Patch from Tri Patch

In **RpPatchMesh** tri patches are closely related to quad patches. The diagram below adds the u and v axes. The first patch position stored in the **RpQuadPatch** and **RpTriPatch** determines the control point where u and v both equal zero. The same element is retained as control point zero when a quad patch is created from a tri patch, so it is the side opposite control point zero that is coincident over 0≤u, 0≤v and 0≤(1-(u+v)).



A quad patch can retain coordinates coincident with a tri patch

RtBezierQuadFromTriangle

The function **RtBezierQuadFromTriangle()** takes two **RtBezierMatrix** arguments. The second is a pointer to a matrix containing control points that describe the tri patch. The first is a pointer to memory already reserved for the quad patch's control points. The quad patch data is returned at the address in the first argument.

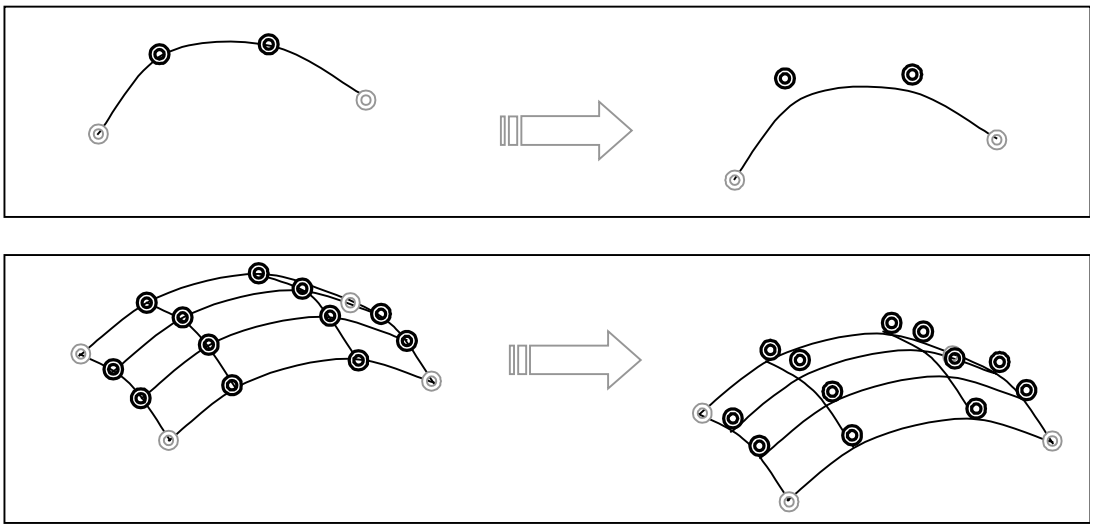
The function calculates the control points to complete a quad patch from the given tri patch. The quad patch is coincident only over the part of the tri patch illustrated above where $0 \le u$, $0 \le v$ and $0 \le (1-(u+v))$.

During this process the control points may be re-positioned, but the original triangular surface that they define will not be altered.

26.4.4 Surface Points to Control Points and Back

RtBezierQuadControlFit3d()

A patch is defined by its control points. The **RtBezierQuadControlFit3d** function derives the control points from a patch specified by the corresponding "sample" points on its surface. In each Bézier curve that defines an edge of a Bézier patch, points P_1 and P_2 are off the curve, and off the surface of the patch. The first and last points are on the curve, and on the surface of the patch. An application that needs to calculate control points from the surface of a patch may do so using the function **RtBezierQuadControlFit3d()**.



Control points derived from surface points

RtBezierQuadControlFit3d() takes the coordinates on the surface of the quad patch in the form of a pointer to an **RtBezierMatrix** and returns the equivalent control points for each in a second **RtBezierMatrix**. The first argument is a pointer to memory reserved for a quad matrix and the second points to the source data. The result is returned in the **RtBezierMatrix** pointed to by the first argument.

RtBezierTriangleControlFit3d

RtBezierTriangleControlFit3d() derives a tri patch from surface points. It takes the coordinates of points on the curved surface and returns the control points for a patch that goes through the source points. It is the equivalent of the function **RtBezierQuadControlFit3d()** for tri patches. It takes a pointer to a source tri patch in the format of an **RtBezierMatrix** and a pointer to memory reserved for a second tri patch in the same format, in which the results are returned.

RtBezierQuadSample3d

Calculates a point anywhere on the surface of a quad patch, given the control points of the patch and the UV coordinates of the point. An example of its use is given in the API Reference.

The function **RtBezierQuadSample3d()** takes four arguments. The first is an **RwV3d**, a 3d vector, that receives the coordinates of a point on the surface of the patch. The second is a pointer to an **RtBezierMatrix** that holds the coordinates of a patch. The third and fourth are **RwReal**s which pass the *u* and *v* coordinates of control points (in parameter space) of a point on the patch whose real-world coordinates are returned in the first argument.

The previous two functions worked at the control points of the patch. This function calculates any point, anywhere on the patch.

There is no tri patch equivalent for these functions; the developer must convert tri patches to quad patches to get the source data.

26.4.5 Forward Differencing

When a patch has to be facetted, the coordinates of each facet must be calculated. The number of coordinates depends on the LOD (it will be between 4*4 and 20*20). Each coordinate of each point in the matrix requires multiplication in *x*, *y* and *z*. This includes multiplication to the power of two and of three. This would require a lot of processing time. The application of Bernstein coefficients make the processing more efficient. But in addition to this, RenderWare Graphics uses the method known as forward differencing to reduce the processing required to a fraction of itself.

Forward differencing avoids the repeated multiplication required to calculate the shapes of Bézier patches, replacing many expensive multiplications by a smaller number of inexpensive additions. This is possible because patch surfaces curve progressively. It is beyond the scope of this document to summarize the mathematics behind this subject but at its simplest, forward differencing calculates and stores the differences between positions that are spread at regular intervals across the patch.

The points on the patch are always generated in the same order. The row of points 00, 01, 02 and 03 (incrementing in u) are processed first. Then row 10, 11, 12 and 13, and so on (incrementing row by row in *v*). See the code samples given in the API Reference.

Initially the coordinate values still need to be established with repeated multiplication. But when the matrix of difference values has been calculated from them, coordinates can then be calculated using a small number of additions.

The effectiveness of this approach further reduces the processing time, to about a tenth.

Naming

The names of many of the functions below indicate that they derive the "Difference" between positions on the patch.

Two functions are involved in calculating the weights; the values that determine how much of each local control point should be added to calculate a given surface position, depending on its coordinate values in *u* and *v*. This fast method of calculating a surface point from its control points is based on Bernstein polynomial equations. These functions have the phrase "Bernstein Weights" in their names.

Some of these functions need to operate in only three dimensions. But it is sometimes useful to be able to modify another value, like a color or light value, according to its position in the patch. For this purpose, this group of functions contains alternatives that operate on either "3d" or "4d" vectors. The fourth value in each vector is processed by the same mathematical functions as the positional values.

The API Reference contains sample code showing how these functions are used to calculate, store and apply the values used for forward differencing. The calculations are applied to each vertex along the *u* axis, step by step. When a row of calculations is complete, the process is repeated on each row in the *v* axis, step by step. These steps are performed by the "StepU" and "StepV" functions whose names indicate that they operate horizontally along rows in the "U" axis or vertically in the "V" axis.

Sequence

The functions described in this Toolkit section are presented in a sequence that reflects the order in which they are used to implement forward differencing of patches.

1. The "Bernstein Weight" functions calculate a matrix of vectors that corresponds to a weighted matrix of control points on a patch. This weighting is constant for a given matrix of control points. These pre-calculated results are used later, to find difference values for given step sizes.

2. The "Difference3d/4d" functions calculate the coordinates of points on the patch at regular intervals determined by the LOD and store the differences of their coordinate positions.

3. The "DifferenceStepU" functions update a separate row of values which, when added in the right sequence and factoring in a coordinate in *u* and *v*, will provide differences of positions at any interval across the patch. This calculates and stores the horizontal difference values.

4. The "DifferenceStepV" functions update a matrix vertically corresponding to one step in the *v* axis. It does the same calculations as the DifferenceStepU functions applying them to the differences in *v*.

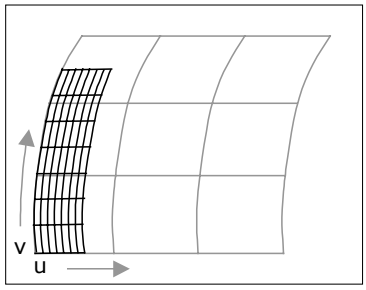
RtBezierQuadBernsteinWeight3d and RtBezierQuadBernsteinWeight4d

The two functions, **`RtBezierQuadBernsteinWeight3d/4d()`** calculate the weighted matrix to be used when a matrix of differences is found for a patch. The two alternative variations on the same function are provided to calculate a matrix of 3d vectors or 4d vectors.

When a patch surface is split into facets, five arrays have to be multiplied to calculate coordinates of surface points. The values of three of these arrays are constant for given control points. So they can be multiplied first and the results stored for later use. This extra processing is handled by the functions **`RtBezierQuadBernsteinWeight3d/4d()`**. Both take two arguments, both are pointers to an **`RtBezierMatrix`**, the first is to return the result, and the second is to receive the patch data to process.

RtBezierQuadOriginDifference3d and RtBezierQuadOriginDifference4d

These functions calculate the difference values for forward differencing. They fill a matrix of vectors with the differences of each point from the one before it, working first in the *u* and then in the *v* directions across the patch. The versatile **`RtBezierMatrix`**, `as it is used in the first parameter`, does not represent the layout of the patch here. It stores difference values in its two dimensions so that they can be summed progressively within the matrix to produce a series of difference values across the patch.

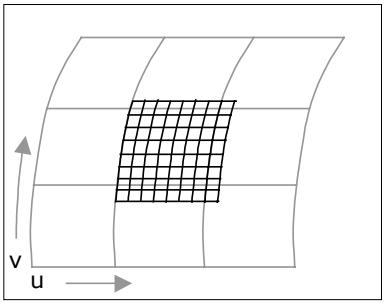


The differences are calculated from the origin, *u*=0, *v*=0, and the step size in the instance in this diagram does *not* cover the whole patch.

Both the functions **RtBezierQuadOriginDifference3/4d()** take four arguments. The first two are pointers to **RtBezierMatrix** structures, the first receives the results and the second holds the source data. The third and fourth are the step size in *u* and in *v*, which, in practice, are probably both the same value and correspond to the LOD value (4 to 20). They return the data by filling the matrix pointed to in the first argument with values that will be used to produce the differences of coordinate positions. This function assumes that the area of the patch to be covered will begin at the origin, *u*=0, *v*=0, and the calculations are optimized for this.

RtBezierQuadPointDifference3d and RtBezierQuadPointDifference4d

These two functions are variations on the preceding pair. They take the *u* and *v* values passed as arguments as the start point of the area to be tessellated.



The difference values are calculated from *u*=1/3, *v*=1/4, and the step size in the instance in this diagram does *not* cover the whole patch

The two functions take six arguments in all. The first two are pointers to **RtBezierMatrix** structures, the first receives the results and the second holds the source data. The third and fourth are the *u* and *v* coordinates of the start point, and the fifth and sixth are the step size in *u* and in *v* as before. The step size does not have to fill the whole patch, and this allows the developer to sample only part of the patch, as illustrated above. They return the data by filling the matrix pointed to in the first argument, with the difference values calculated at a point other than the parameter origin.

RtBezierQuadDifferenceStepU3d and RtBezierQuadDifferenceStepU4d

RtBezierQuadDifferenceStepU3/4d() and **RtBezierQuadDifferenceStepV3/4d()** are used in the process of applying the difference values calculated in the four functions above, **RtBezierQuadOriginDifference3/4d()** and **RtBezierQuadPointDifference3/4d()**. The "StepU" function works across the patch, and the "StepV" functions works up it. The "StepU" function is called on each iteration of a row of coordinates that samples one cross section of a patch. It updates values in the current row of a matrix. An example of the use of this function is found in the API reference. The "StepU" function takes a pointer to a single argument, an **RtBezierRow**, which is one row of an **RtBezierMatrix**.

RtBezierQuadDifferenceStepV3d and RtBezierQuadDifferenceStepV4d

The functions **RtBezierQuadDifferenceStepV3d/4d()** update the difference row (**RtBezierRow**) returned by **RtBezierQuadDifferenceStepU3d/4d()**. These rows are arrays of differences, and they are updated into a matrix. The API Reference gives an example of the functions' use. The function takes a single pointer to an argument, an **RtBezierMatrix**, which updates values throughout a matrix.

26.4.6 Patch Tangents and Normals

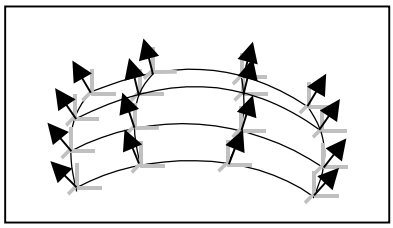
Curved surfaces are brighter or darker as they curve toward or away from the light. RenderWare Graphics supports this sort of shading in patch meshes by calculating how far a point on the surface curves toward a given light source. In order to do this the patch mesh stores an array of normal vectors for each control point to calculate how much the surface faces the light.

The function that calculates a matrix of normals is **RtBezierQuadGetNormals()** and the previous section on Bézier patches showed how it was used in the example code to fill the array of normals after creating a new patch mesh.

In order to find the normal vectors, RenderWare Graphics has to find two tangents for each point at which the normal is to be calculated. The tangents can be useful for other things. For instance, if two objects represented by RenderWare Graphics are to be placed one on the other, the tangents of the parts that touch can determine the frame for the upper object. So two functions to find tangents to patches are exposed.

RtBezierQuadGetNormals

RpBezierQuadGetNormals() takes two arguments. The second is the source **RtBezierMatrix** and it fills a second **RtBezierMatrix** with vertices that describe the normal vector for each respective control point. The **patch** example code demonstrates this function when it adds a short line at each vertex, pointing out from the teapot. This line is the vector calculated by this function.



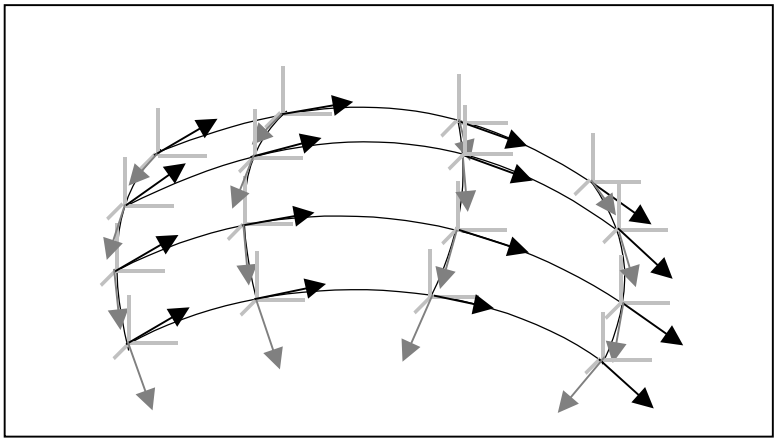
Normal vectors at the 16 surface points
that correspond to control points.

A tri patch can be converted efficiently to a quad patch using **RtBezierQuadFromTriangle()** to apply this function to tri patches.

RtBezierQuadTangent() and RtBezierQuadTangentPair()

RtBezierQuadTangent() takes three arguments. The first is a pointer to an **RtBezierMatrix** to receive the tangent values that the function will calculate. The second is an **RwReal** that represents an angle in radians from the *u* axis, and the third is a pointer to another **RwBezierMatrix** that holds the control point source data. The function calculates control points for the tangents at each control point and stores them in the first matrix in the direction indicated, and the second matrix in a direction at right angles to it.

RtBezierQuadTangentPair() takes four arguments. The first two are pointers to **RtBezierMatrix** to receive the tangent values that the function will calculate. The third is an **RwReal** that represents the direction of measurement, and the fourth is a pointer to another **RtBezierMatrix** that holds the control point source data. The function calculates control points for the tangents at each control point and stores them in the first two matrices; the first matrix in the direction indicated, and the second matrix in a direction at right angles to it. The tangents given by the control points can then be evaluated efficiently with forward differencing.



Two perpendicular curve tangents, at each of the 16 surface points that correspond to control points.

There are no tri patch equivalents for these functions; the developer must convert tri patches to quad patches.

26.4.7 Toolkit Summary

The **RtBezPat** Toolkit exposes valuable functions that **RpPatchMesh** already uses. It converts tri patches into functionally equivalent quad patches, it converts between surface points and control points, it exposes a set of functions for forward differencing and calculates the normal vectors and tangents to a quad patch's surface.

26.5 Summary

This chapter deals with three modules which work with curves in different ways.

RpSpline is a plugin which uses B-splines to calculate 3d curves that can be used in many ways, and commonly represent the paths for various objects. It supports closed and open splines. It allows them to be created, modified and destroyed. It can calculate the position and angle of the curve at any point and does so efficiently.

RpPatchMesh implements curved surfaces as Bézier patches linked into a mesh to encode complicated curved shapes. It renders them to a wide range of LODs. It integrates them with the refinement and rasterizing processes of RenderWare Graphics, it incorporates code to deal with colors, textures and lighting values and it can smooth patches that join awkwardly.

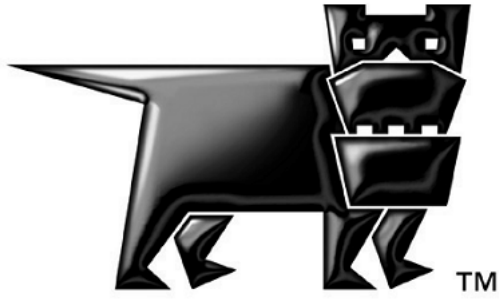
The **RtBezPat** Toolkit exposes some useful functions already in **RpPatchMesh** so that the developer can take advantage of much of the functionality selectively, and incorporating it in new code.

Part E

World Management Libraries

Chapter 27

Collision Detection



27.1 Introduction

Collision detection is an important part of most 3D graphics applications. It is primarily concerned with preventing model geometry from intersecting the geometry of other models, but has a number of related uses.

For example:

- Picking objects or geometry;
- Object collisions for physics;
- Object placement, such as dropping a spline onto a surface;
- Use in tools and user interfaces;

27.1.1 Plugins & Toolkits

Collision detection, intersection tests and picking tools are supplied by:

- **`RpCollision`** – Collision detection tools;
- **`RtIntersection`** – Intersection testing tools;
- **`RtPick`** – Atomic / geometry picking tools;

All three are covered in this chapter.

Reacting to Collisions

The tools described in this chapter will only detect a collision, but will not perform any actions as a result of a collision.

For instance, functions are provided to determine whether two atomics intersect, but RenderWare Graphics will not move the atomics apart if a collision has occurred.

Any necessary actions as a result of a collision must be performed by the application.

27.2 Detecting Collisions

27.2.1 The RpCollision Plugin

This plugin provides most of the higher-level tools for detection of collisions with the RenderWare Graphics **RpWorld** and **RpAtomic** objects.

Usage revolves around manipulation of the **RpIntersection** primitive. This is a transparent object with two primary elements:

```
RpIntersectData  t
RpIntersectType  type
```

The intersection **type** determines the contents of **t**.

In practice, the steps involved in testing for an intersection are:

1. Define a variable of type **RpIntersection** and set the elements accordingly;
2. Call the appropriate intersection function;
3. Interrogate the returned **RpIntersection** and **RpCollisionTriangle** structures and act upon the results.

The **RpIntersection** object supports the following types of intersection:

- **rpINTERSECTLINE** – line intersection;
- **rpINTERSECTPOINT** – point intersection;
- **rpINTERSECTSPHERE** – sphere intersection (e.g. a bounding sphere);
- **rpINTERSECTBOX** – box intersection (**RwBBox** type);
- **rpINTERSECTATOMIC** – atomic intersection (based on bounding sphere).

The **RpCollision** plugin's primary use is to compare target geometry with the intersection object.

The plugin supports optimized collision data for both static and dynamic geometry types (world sectors and atomics) to speed up the collision detection process.

27.2.2 The RtIntersection Toolkit

This toolkit contains low-level intersection testing functions which test for intersections between triangles and three other geometry types: lines, spheres and bounding boxes. (This toolkit is also used by **RpCollision** and so must be included and linked into your application if you are using this plugin.)

You can use the functions contained in this toolkit if you require only the collision tests it provides. If you need the high-level API, use **RpCollision**.

The functions available are:

- **RtIntersectionBBoxTriangle()**

  The simplest of the functions, this returns **TRUE** if the triangle—supplied as three vertices (**RwV3d**)—intersects the bounding box (**RwBBox**).

- **RtIntersectionLineTriangle()**

  This function returns **TRUE** if the line—specified as a starting point (**RwV3d**) and a *line delta* (**RwV3d**), defining the displacement vector—intersects with the triangle (**RwV3d**).

  The *line delta* parameter is used to reduce calculation overheads when processing a large number of triangles. It can be obtained by **RwV3dSub(lineDelta, &line.end, &line.start)**.

  This function also takes another parameter, *distance*, which will hold the parametric distance to the intersection if an intersection was found.

  This function uses back-face culling.

  This means:

  a. the order of the triangle vertices is very important;

  b. you will need to make two calls for two-sided tests.

- **RtIntersectionSphereTriangle()**

  This checks for an intersection between a sphere and a triangle. The sphere is an **RwSphere** object; the triangle is specified by its three vertices (**RwV3d**).

  If an intersection has taken place, the function will return **TRUE** and set two variables: a *normal* (**RwV3d**) for the triangle and the perpendicular *distance* of the sphere center from the plane of the triangle.

Parametric Distances

This term means the distance reported will be a value between 0.0 and 1.0, essentially producing a result scaled for a line of unit length.

For instance, if the line is 10 units long and the intersection occurs at a point 7 units along the line, the parametric distance returned will be 0.7

27.3 Picking

If your application only needs to select atomics or other objects on screen at a specific pixel location, the **RtPick** toolkit is ideal for the purpose.

RtPick extends two RenderWare Graphics components, **RpWorld** and the Core Library, adding functions to their respective APIs.

Dependencies

RtPick requires both the **RpWorld** and **RpCollision** plugins.

27.3.1 The RtPick Toolkit

This toolkit exposes the following functions:

- **RwCameraCalcPixelRay()**

  This function determines the parameters of the line passing through the specified pixel. The line starts and ends on the camera's near and far clip planes and is specified in world units.

  This line can then be used for intersection tests or with **RpWorldPickAtomicOnLine()** (see below).

- **RwCameraPickAtomicOnPixel()**

  This function returns a pointer to the atomic which is being rendered to form the specified pixel. If no atomic is rendered at the pixel, the function returns **NULL**.

  This is probably the most suitable for selecting onscreen atomics in applications.

- **RpWorldPickAtomicOnLine()**

  This function is used to determine the atomic in the specified world that intersects the given line closest to its start point. The parameters of the line, its start and end positions, are specified in world units.

  This function determines intersections based on the atomic's bounding sphere. If you require more accurate picking, you will need to **RtIntersection** toolkit to determine which specific triangle has been picked.

RtPick has clear and obvious applications in user interface design for both tools and computer games.

For instance, a space combat simulator usually includes lasers fired from the player's ship. The **RwCameraPickAtomicOnPixel()** function could therefore be used to determine which atomic—and thus, which spaceship—is targeted when the player presses fire.

The "picking" example

The SDK includes the "**picking**" example, which demonstrates the **RtPick** toolkit.

27.4 Static Geometry Intersections

The **RpCollision** plugin extends the **RpWorld** plugin to support collision detection. The additional functionality is provided to give developers the means to check for collisions with three types of object in a world: world sectors, atomics, or triangles making up the static geometry. Due to the potentially large number of triangles in the model, the **RpCollision** plugin supports additional collision data to speed up the latter type of test.

27.4.1 Collisions with World Triangles

Building the Collision Data

For collision tests with world triangles the **RpWorld** collision detection functionality assumes optimized collision data has been created to speed up the process.

Building the collision data is an offline process and is performed as either a modeling package exporter step or as part of a developer's custom tool-chain. The modeling package exporters supplied with the RenderWare Graphics SDK include this facility as standard.

The collision data is a type of BSP-tree structure which further subdivides the world sectors using axis-aligned planes. This speeds up the collision detection routines by allowing irrelevant geometry to be eliminated from consideration very quickly.

Disabling collision data generation

If your application will not require collision data, artists should disable the generation of this data in the exporter. This will reduce the memory footprint required by the exported data.

Building the Data Yourself

There are two functions provided to generate the requisite collision data.

The most commonly used is **RpCollisionWorldBuildData()**, which generates the BSP structures for all world sectors within the specified world object. In most cases, this is the only function you will need.

If you require finer control over which world sectors have collision data, an additional function is available, **RpCollisionWorldSectorBuildData()**. This function builds collision data for the specified world sector.

The collision data may also be removed from a world or world sector using the **RpCollisionWorldDestroyData()** and **RpCollisionWorldSectorDestroyData()** functions. The existence of collision data may be checked using **RpCollisionWorldQueryData()** or **RpCollisionWorldSectorQueryData()**.

Using the Collision Data

The **RpCollisionWorldForAllIntersections()** function is used to perform collision detection tests against the triangles in a world with the assistance of the pre-generated collision data.

This function requires the developer to instantiate and initialize an **RpIntersection** object. This object is transparent, meaning its individual elements are exposed and documented. It has two elements:

type – an enumerated value which specifies the type of geometry to be used for collision testing;

t – is where the physical geometry object is stored, be it a point, line, sphere, bounding box or atomic.

It is defined as a **union**:

```
union RpIntersectData
{
    RwLine              line;
    RwV3d               point;
    RwSphere            sphere;
    RwBBox              box;
    void               *object;
};
```

The following code fragment illustrates usage of **RpIntersection** by setting it up with a sphere and calling the collision detection function:

```
RpIntersection intersect; /* instance of RpIntersection object */
RwV3d center; /* the center of a sphere to test */
RwReal radius; /* the radius of a sphere to test */

... some code ...

intersect.type = rpINTERSECTSPHERE
intersect.t.sphere.center = center;
intersect.t.sphere.radius = radius;


/* RpIntersection object is now prepared, do the tests... */
RpCollisionWorldForAllIntersections( myWorld, &intersect,
IntersectCallback, NULL);
```

IntersectCallBack() is a function pointer. It represents a callback function which will be called whenever the collision testing results in an intersection being found with a triangle. The callback function can return **null** to cease further collision testing.

The callback function prototype is defined with a **typedef** as follows:

```
typedef
```

```
RpCollisionTriangle *(* RpIntersectionCallBackWorldTriangle)
        (RpIntersection *intersection,
         RpWorldSector *sector,
         RpCollisionTriangle *collTriangle,
         RwReal distance,
         void *data)
```

The callback function must match this prototype.

27.4.2 Collisions with World Sectors

A more coarse-grained collision detection algorithm is also available in **RpWorldForAllWorldSectorCollisions()**. This can be used to determine which world sectors are intersected by the primitive specified in the **RpIntersection** object.

This function also requires a callback, the prototype for which is defined as follows:

```
typedef
RpWorldSector *(* RpIntersectionCallBackWorldSector)
        (RpIntersection *intersection,
         RpWorldSector *worldSector,
         void *data)
```

27.4.3 Collisions with World Atomics

The **RpIntersectionData** element can also store a pointer to an atomic (type **RpAtomic**) for collision testing by setting the **RpIntersectionType** element to **rpINTERSECTIONATOMIC** and setting the **RpIntersectionData** element with a pointer to the atomic.

Only the bounding sphere of an atomic will be referenced during the testing process; individual triangles are ignored. The result is the same as would have been achieved had the atomic's bounding sphere been passed directly, but is more efficient since internal knowledge of which atomics lie in a particular world sector may be used. For fine-grained test with atomic triangles, see section 27.5.

Examples

The "**collis1**" example supplied with the RenderWare Graphics SDK demonstrates the **RpCollision** plugin in action. The example uses the plugin to keep a user controlled camera a fixed distance above a landscape.

The "**collis2**" example also illustrates the plugin in action. In this example, an atomic – a modeled as a simple sphere – is shown bouncing around the inside of a buckyball. This atomic itself is passed through the **RpIntersection** object for testing.

27.5 Atomic & Geometry Intersections

The **RpCollision** plugin provides a means of testing for collisions between an intersection primitive and the triangles of an atomic or, more directly, a geometry.

Collision is even possible with morphed atomics, but not with skinned atomics where the interpolated vertex data cannot be accessed by the CPU on certain platforms. However, some alternative suggestions are given later.

27.5.1 Collision Data

The **RpCollision** plugin supports the extension of dynamic geometry with collision data for improved performance. This is applicable whenever an atomic may be considered "rigid", i.e. its geometry is never modified but its frame may be transformed. The geometry collision data is of the same type as world sector collision data and is particularly useful for:

- Fast, precise collision tests with detailed models;
- Object collision with moving parts of a world (e.g. platforms);
- Custom collision worlds built from atomics rather than conventional world sectors.

Collision data construction, like its world sector counterpart, is intended to be performed offline, but may be performed during initialization if the geometry is small enough. The data construction function is **RpCollisionGeometryBuildData()**.

The following usage illustration is from the **CollisionDataBuildCallback()** function of the "**collis3**" example:

```
RpCollisionGeometryBuildData(RpAtomicGetGeometry(SpinnerAtomic),
NULL);
```

Once built, the collision data is then automatically used in collision detection tests.

27.5.2 Performing Collision Tests

Collisions with Geometry Triangles

Collisions tests may be performed directly on the triangles of an **RpGeometry** using the function

RpCollisionGeometryForAllIntersections().

It will check for intersections between the **RpIntersection** primitive specified and the geometry, executing a callback for each intersection found.

The callback function must match the following prototype:

```
typedef
RpCollisionTriangle *(* RpIntersectionCallBackGeometryTriangle)
        (RpIntersection *intersection,
         RpCollisionTriangle *collTriangle,
         RwReal distance,
         void *data)
```

If geometry collision data exists, this will automatically be used for improved performance.

The advantage of using this function is that the **RpIntersection** is specified and the calculations performed in object space, potentially saving unnecessary transformations. This also allows the **rpINTERSECTBOX** to be available since this uses a bounding box aligned with the local space.

On the other hand, this function cannot be used for a morphed atomic, since the current state of interpolation is unknown at the geometry level.

Collisions with Atomics

Collision tests with the triangles of a geometry may be performed at the atomic level using the function

RpAtomicForAllIntersections(). This function checks for intersections between the **RpIntersection** primitive specified *in world space* and a specified atomic.

If the atomic has only one morph target and contains geometry with pre-calculated collision data, this data will be used directly, thus speeding up the intersection detection. Morphed atomics are supported by testing against triangles constructed by the atomic's **RpInterpolator** object.

As with the other collision detection functions, this too requires the developer provide a callback function which will be executed on each intersection. The prototype is identical to that for **RpCollisionGeometryForAllIntersections()**.

Note that the **RpCollisionTriangle** passed to the callback function is given in object space.

RpAtomicForAllIntersections() will not work with atomics used for skinned animations.

Suggested alternatives are:

- use spheres or bounding boxes around bones and joints;
- collide with a low-poly rigid body collision model attached to the same frame hierarchy as the skinned model.

27.5.3 Example

The "**collis3**" example supplied with the SDK illustrates generation and use of geometry collision data.

The example shows several spheres being pushed around by a set of spinning objects.

It should be stressed that the simulation is for demonstration purposes and not intended to be realistic; the algorithm is therefore crude and is used purely to illustrate the collision testing process.

On startup, the geometries of the bowl and spinners do not have collision data. During a sphere intersection test, every triangle in every spinner is individually tested. In this state the frame-rate of the application is limited by this testing.

Collision data may be built via the menu, and once created the performance should improve. This data provides information to enable fast isolation of the triangles in the geometry which potentially intersect the sphere before individual tests are performed.

Collision data generation is intended for offline use in custom tools and exporters. The code provided with this example shows the simple steps required to load an atomic, build geometry collision data, and resave.

27.6 Summary

27.6.1 APIs

RenderWare Graphics provides three APIs for collision detection, intersection testing and picking:

RpCollision

- Mid-level collision detection API
- Concerned with intersection testing between primitives and atomic, geometry and world sector objects
- Primitives represented by **`RpIntersection`** primitive
- Can operate directly on static or dynamic geometry
- Can use optimized collision data for optimal speed
- Requires world plugin

RtIntersection

- Low-level intersection test API
- Works at triangle level
- Standalone – does not depend on other plugins being present

RtPick

- Picking API
- Primarily intended for use in user interfaces
- Requires both world and collision plugin

27.6.2 Hints & Tips

A number of techniques can be used to get the best results:

- Specify coordinates of intersection primitives in world space for world and atomic intersections, and object space for geometry intersections
- Do not modify the intersection primitive in the intersection callback

- The collision triangle in the callback is in object space for atomic and geometry intersections. See "**collis3**" example for how to transform this to world space if required.

- Make sure you understand the precise meaning of the distance in intersection callback before using it. In most cases, and for efficiency reasons, the distance is scaled to unit length, such that the maximum possible distance is 1.0.

- Geometry collision data assumes static geometry – do not use it for geometries that will be procedurally modified.

- **RpWorldForAllAtomicIntersections()** has a number of requirements:

  - Atomics must have a frame and have been added to a world.

  - They must have their **rpATOMICCOLLISIONTEST** flag set.

  - They must *not* have been relocated since last update.

  - Changes to the **rpATOMICCOLLISIONTEST** flag are only registered after a resync.

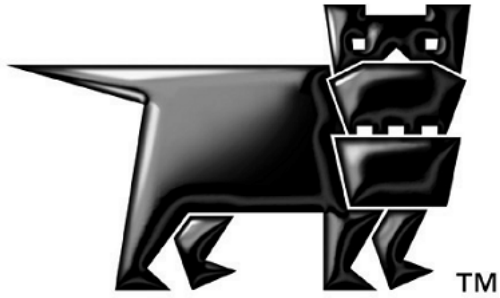
- **RpCollisionWorldForAllIntersections()**: collision data must exist otherwise no intersections will be found.

- **rpINTERSECTATOMIC** primitive: intersections are with the bounding sphere of the atomic *only*. For world tests, the atomic must have a frame and exist in a world, and must not have been modified since the last resync. If this is a problem, use the **rpINTERSECTSPHERE** type, taking the bounding sphere for the primitive.

- Line-triangle tests (including **Rp...ForAllIntersections()** functions using the **rpINTERSECTLINE** primitive) only check for collisions with the front face of triangles. For two-sided tests, do a second pass with the direction of the line reversed.

Chapter 28

Potentially Visible Sets



28.1 Introduction

In this chapter, Potentially Visible Sets (PVS) are covered. In this section the details and applications of PVS are introduced. In Section 28.2, details of the generation of PVS data is discussed and in Section 28.3 the usage of PVS described. A summary is presented in Section 28.4.

28.1.1 What are Potentially Visible Sets?

Potentially Visible Sets (PVS) provide a means of optimizing the rendering of static world data by culling hidden world sectors from consideration as quickly as possible.

PVS data is handled by a plugin which extends RenderWare Graphics' world sector objects (Refer to chapter World & Static Models) with a *visibility map* that defines which world sectors are visible from that sector.

These visibility maps are not calculated at run-time as the processing takes an appreciable amount of time. Instead, the scenery data is preprocessed by *sampling* the scene at specific locations within each sector to determine the absolute visibility of the other world sectors – hidden or potentially visible.

28.1.2 The APIs

PVS data generation and usage is handled through the **RpPVS** plugin. This plugin contains functionality for both preprocessing and run-time processing.

In addition, there is the **RtSplinePVS** toolkit. This contains a single utility function for generating PVS data for a camera tied to a spline.

28.1.3 Applications for PVS functionality

PVS works best with environments which contain a lot of occluding structures, e.g. walls. This makes it ideal for static worlds containing structure interiors, caves, tunnels and similarly enclosed spaces where views are heavily restricted.

PVS also works well in certain outdoor environments. A racing game could be preprocessed with useful PVS data if the geometry has been designed with it in mind. For example, a twisting mountain road may have enough occlusion, courtesy of the surrounding valleys and mountains, to ensure that only a small part of the entire level is visible at any one time.

The **RtSplinePVS** toolkit was designed with racing games in mind. It lets you use a spline to determine where the PVS sample points are located. Setting the spline to follow the road means only areas the player would actually see would be sampled.

By contrast, the **RpPVS** plugin's default algorithm sets up sampling points in a grid pattern which covers the entire scene.

28.1.4 Reasons for NOT using PVS data

It follows from the above that PVS processing is of little advantage in static worlds with very little occlusion. A landscape consisting of broad, open flatlands will see few benefits from PVS.

The static nature of PVS visibility maps raises another unsuitability. As these are linked to world sectors, the PVS processing will take no account of occlusion of scenery by dynamic models. Similarly, if you are not using static world data for your scenes—for instance, when simulating deformable scenery using atomics—then PVS processing will be of no use.

28.2 Building PVS Data

Constructing PVS data can be achieved in a number of ways. Each of the following recommended ways should be run from a Windows platform:

- Using the "**pvscnvrt**" tool supplied with the SDK.
- Using the "**pvsedit**" tool supplied with the SDK.
- Programmatically, using the **RpPVS** API.

We'll look at each of these in turn now. (Note, the viewer for static worlds, "**wrldview**", can also be used to generate PVS data, but it provides little control as is not recommended.) RenderWare Visualizer can be used to view PVS data.

28.2.1 Using the PVS Converter

A useful *Windows platform* tool resides in the tools folder called "**pvscnvrt**" – this is the recommended method for generating PVS data. The tool allows for the deletion of PVS data, generation of PVS data with a sample-point density control variable, and regeneration/enhancement of PVS data, as described below.

The tool allows for the important conversion from old format **.bsp** files to the new format. (The format of PVS data has changed since release 3.10, and old **.bsp** files are no longer recognized.)

Check the **readme.txt** file that accompanies the "**pvscnvrt**" program as this describes the windows and command-line user interface for it.

28.2.2 Using the PVS Editor

This program provides a visual interface to the operations that PVS converter provides and is also useful for manually editing PVS data. It can generate PVS data for the entire world and show the results immediately on completion. It also provides a mechanism for exporting and importing PVS data alone (i.e. unattached to any world).

See the **PVSEdit.doc** documentation in the docs' tools folder for additional information.

28.2.3 Using RpPVS

The **RpPVS** plugin provides the functionality needed for generating PVS data. This can be handled either by the default scene sampling function (**RpPVSGeneric()**), or by a function of your own devising.

The function where all this happens is **RpPVSConstruct()**. The API Reference contains a detailed breakdown of how this function works which proceeds as follows:

1. The plugin is given a world object to process.
2. The function applies a callback function to each world sector within the world.
3. The callback fills in the supplied visibility map with the visibility data according to its own algorithm.

The **RpPVS** plugin must be linked against and attached (using **RpPVSPluginAttach()**) in the usual manner.

This plugin is dependent on the **RpWorld** plugin as it extends the world sector object, and the **RpCollision** plugin for selecting sampling points, so these must also be linked against and attached.

RpPVSConstruct() is really a housekeeping function, creating and managing the visibility maps for the world sectors. The actual generation of the visibility map data is achieved by the callback function. A default callback is supplied, **RpPVSGeneric()**, but you are free to substitute it with your own.

The RpPVSGeneric() CallBack

This function creates a strategically arranged grid of points and takes a visibility sample at each. It very easy to use as the processing is self-contained and no further intervention is required on the part of the programmer.

This callback takes a user data parameter—an **RwReal** between 0.01 and 1.0—which defines the density of the sampling points.

Usually, collision detection means that the viewpoint will never exist in certain areas of a sector (such as beneath the floor or terrain) and thus the samples are only taken inside valid regions of the sector. This is good as it results in more culling. However, for a small number of special environments where collision detection is not used, this is not a desirable enhancement to the sampling process. The function **RpPVSSetCollisionDetection()** is provided to control the use of this feature; by default collision detection is set to **TRUE**.

Similarly, back-faces in the scene are culled before PVS generation takes place. However, in some scenes on some platforms do render with back-face culling turned off, and **RpPVSSetBackFaceCulling()** can be used to tell this to the PVS generator. Note, this function must be called before any PVS generation takes place.

While the grid-based approach is perfect for many environments—particularly for those commonly seen in first-person shooter games—it is not ideal for all types of environment.

For instance, many racing games have an essentially two-dimensional environment, so it makes little sense to take samples at points above or below the road surface. For these, it makes sense to use alternative sampling algorithms, such as a two-dimensional grid of points. An alternative algorithm is provided by the **RtSplinePVS** toolkit, which has been designed with racing games in mind. It is covered on page 273.

Writing your Own CallBack Function

Nobody understands your data design like you do, so RenderWare Graphics allows you to write your own callback function for **RpPVSConstruct()**.

The approach taken will obviously vary according to your needs, but there are two common processes that need performing. For each world sector, you will need to:

1. Determine where the sampling points should be placed.

2. Calculate the PVS from each sampling point.

The first step is dependent on your application's data design and its specific requirements. The callback function is given the world sector and a bounding box (**RwBBox**) which defines the area your function is to process.

The second step is performed by calling the **RpPVSSamplePOV()** function. (This function takes a parameter to specify whether collision detection should be used, as discussed in the previous section, but temporarily overrides that set by **RpPVSSetCollisionDetection()**)

The sampling process involves taking a "snapshot" of the scene around the specified point of view (POV). The rendering takes in a complete 360º spherical view—the virtual camera "seeing" in all directions at once.

The rendering process is tracked so that the visibility of other world sectors from this point of view can be checked and appropriate entries made in the visibility map.

The visibility map is maintained by the **RpPVSConstruct()** function, so your application is not required to perform this processing itself.

Tweaking PVS data

RpPVSSamplePOV() can be used to add extra samples to update a visibility map. This is useful if you have a convoluted scene and some world sectors are being incorrectly marked as "invisible". Note, if you are calling this function multiple times, you should make sure your world has collision data.

RpPVSConstruct() may be called more than once. The new information generated updating that which exists already. The density parameter supplied does not necessarily need to be adjusted since sample points are taken in-between those that exist already. Note, if you are calling this function multiple times, you should make sure your world has collision data.

RpPVSConstructSector() can be used to create PVS data for a single sector. This is useful when PVS data needs enhancing in just one sector. Note, if you are calling this function multiple times, you should make sure your world has collision data.

RpPVSSetWorldSectorVisibility() function explicitly sets the visibility of a particular world sector in the current visibility map. For this, the observer is assumed to be located in the world and **RpPVSSetViewPosition()** has been called.

Likewise, but rather more versatile, **RpPVSSetWorldSectorPairedVisibility()** is supplied for explicitly setting cell-to-cell visibility data. Valuable if the results don't give you what you expect, or for some reason, the correct visibility data needs to be overridden.

Writing the Data Out

This is achieved through the RenderWare Graphics binary stream API. Use **RwStreamOpen()** to create the output stream, call **RpWorldStreamWrite()** on your world data to write it out, then close the stream using **RwStreamClose()**.

Destroying PVS Data

If you need to delete PVS data from a world, you can use the **RpPVSDestroy()** function for this purpose.

28.2.4 Using RtSplinePVS

RtSplinePVS is a toolkit containing a single function, **RtSplinePVSConstruct()**, intended to be substituted for **RpPVSConstruct()**.

RtSplinePVSConstruct() is aimed at relatively open, outdoor environments such as those found in traditional racing games. Unlike genres which emphasize environment exploration, racing games rarely define any static model geometry that isn't directly relevant to the racing.

By limiting the samples only to the part of the world which the player is going to see, **RtSplinePVSConstruct()** prevents unnecessary rendering of model data outside the player's view. (This said, it should be noted that PVS will provide little advantage if most of the scene data will be visible at all times.)

RtSplinePVSConstruct() takes a spline as one of its parameters. The spline describes a path through the world along which samples will be taken. No PVS generator callback function is required. However, the progress callbacks, described below, are still called as appropriate.

Once PVS data has been created, you can store the converted world data to a file or other supported storage device using the RenderWare Graphics binary stream API, as described on page 273.

RtSplinePVS relies on the spline functionality provided by the **RpSpline** plugin and this must be attached before use. (More information on this plugin can be found in the *B-splines and Bézier patches* chapter.)

This toolkit is only required for PVS generation with **RtSplinePVS**. *Rendering* worlds with PVS data requires only **RpPVS** to be attached.

28.2.5 Generation Progress CallBacks

The **RpPVS** plugin provides for callback functions intended to convey progress information to the user. These are needed because the PVS generation process can take a while.

The mechanism used is similar to Windows 98's event-driven model whereby the callback is passed messages describing the progress of the generation process. **RpPVSSetProgressCallBack()** is used to set the callback function that will receive these progress messages. This function must match this prototype:

```
RwBool (*RpPVSProgressCallBack) (RwInt32 message, RwReal value);
```

CallBack messages

The callback function takes two parameters: **message** and **value**.

The messages available are:

- **rpPVSPROGRESSSTART –** signifying that the PVS generation process is about to commence. The argument, **value**, is 0.0.
- **rpPVSPROGRESSUPDATE –** signifying that a sample has been processed. The argument **value** is equal to the percentage of the total number of samples (over all sectors) processed up to this point.
- **rpPVSPROGRESSEND** – signifying that the PVS generation progress is complete. All world sectors have been processed and **value** is equal to 100.0.

The callback function should return **FALSE** if the processing should be terminated, or **TRUE** if it should continue.

Percentages

It is perfectly possible that the same percentage is reported on multiple occasions, depending on how the world is structured. In fact, some PVS generation algorithms make it impossible to accurately determine the percentage completed, so the progress returned should be considered only as a rough approximation.

In general, the progress monitoring mechanism should be treated more as a means to provide a "heartbeat" to the user and avoid simply freezing the system until the processing has completed.

28.3 Using PVS Data

Using PVS data is comparatively easy compared to constructing it. The **RpPVS** plugin is hooked into the RenderWare Graphics rendering engine and takes care of the culling process on its own.

The first step, as usual, is to attach the plugin with the **RpPVSPluginAttach()** function.

The second step is to load the world and ensure it contains PVS data. This is achieved using **RpPVSQuery()**, which returns **TRUE** if valid PVS data is found.

At this point, the PVS system needs to be hooked into the rendering engine. This is achieved with **RpPVSHook()**, which takes a pointer to a world containing PVS data.

RpPVSHook() saves the current render callback and replaces it with its own **RpWorldSectorCallBackRender()** function. It still uses the current render callback for rendering. The PVS render callback checks if the current sector is visible, if so then renders the sector by calling the previous render callback function. If the sector is not visible, then it stops rendering the sector at that point.

It is usual to call to **RpPVSHook()** before any rendering of the world. However, if you have multiple worlds to render and only some have PVS data, it will be necessary to hook and unhook the PVS plugin as appropriate before each call to **RpWorldRender()**.

The next link in the chain is to tell the PVS plugin where the camera is positioned prior to rendering. This is achieved with **RpPVSSetViewPosition()**, which takes a pointer to the world in question and a vector describing the position. This gives the culling functions the information needed to select the correct visibility map and determine which world sectors are visible. There is also a partnering function, **RpPVSSetViewSector()**, that be called with an **RpWorldSector**.

The world can then be rendered as usual with **RpWorldRender()**; the culling is performed transparently.

28.3.1 Unhooking the PVS Subsystem

There are two reasons for unhooking the PVS subsystem from the rendering engine:

1. When a world with PVS has been rendered, it may be necessary to render another which does not have PVS data available.

2. All rendering has been completed and the engine is to be closed.

In the first case, the PVS subsystem needs to be disconnected otherwise the rendering engine will produce unexpected results. In the second case, the PVS subsystem needs to be disconnected to allow the RenderWare Graphics engine to be shut down cleanly.

In both cases, the function call required is **RpPVSUnhook()**. It is most common to call this function at the end of a rendering cycle.

RpPVSUnhook() removes the PVS sector render callback and restores the previous sector render callback. Care must be taken if the sector render callback is changed in between **RpPVSHook()** and **RpPVSUnhook()**. Otherwise unexpected behavior will result due to **RpPVSUnhook()** replacing the render callback set by the user.

28.3.2 PVS Runtime Utility Functions

The **RpPVS** plugin provides some useful utility functions which can be used at runtime.

Atomic Visibility

One of the most useful utility functions is **RpPVSAtomicVisible()**, which accepts a pointer to an atomic and determines whether it is visible from the current view position. The function returns **TRUE** if so.

View Position

This is the position set using the **RpPVSSetViewPosition()** function. It is a common mistake to confuse this with the position of the camera object, which may have changed.

It is very important to note that, because of the way the visibility maps are generated, you may get a **TRUE** value returned even if the atomic is behind the camera. Remember, the PVS samples take a full, spherical 360° snapshot.

World Sector Visibility

A similar function exists for world sectors: **RpPVSWorldSectorVisible()**. Given a pointer to a world sector, it will return **TRUE** if the world sector can be seen from the current view position.

(The same notes apply as for atomic visibility.)

Statistics

The **RpPVS** function **RpPVSStatisticsGet()** can be used to obtain basic performance information. This function returns:

1. The number of triangles that would have been rendered if PVS had been disabled.

2. The number of triangles rendered with PVS enabled.

The latter should be a substantially smaller number than the former if the PVS data has been generated properly. If this is not the case, the PVS generation process may need to be adjusted (or the environment is not suitable for PVS as discussed in Section 28.1.4).

28.3.3 Writing Your Own PVS Render CallBack Function

The PVS render callback function performs simple culling of sectors. It does not render visible sectors. Instead it passes the visible sectors to the render callback it replaced. Culled sectors are rejected and are never passed on to be rendered. This causes problems when culled sectors need to be rendered. For example, it would be useful to render visible and not visible sectors in different colors in wireframe.

In such circumstances, the **RpPVSHook()** and **RpPVSUnhook()** functions are unsuitable and should not be used. A custom render callback function is required. Such a function, in addition to performing the rendering, needs to test the sector's visibility. This is achieved by calling the utility function, **RpPVSWorldSectorVisible()**. This returns **TRUE** or **FALSE** which the custom render callback function can use to decide how to render the sector.

In the example mentioned earlier, the render callback function would switch the color used to render the sector.

28.4 Summary

28.4.1 Potentially Visible Sets

Potentially visible sets:

- can drastically improve run-time culling calculations
- are most useful for static models with a high number of occlusions
- must be generated at the preprocessing stage

28.4.2 Generating PVS Data

Using SDK supplied programs

- Use "**pvscnvrt**" in the tools folder to convert PVS data from the old format to the new format, delete PVS data, or generate and enhance PVS data.
- Use "**wrldview**" in the viewers folder to generate PVS data using the default density parameter, to view the results immediately afterwards, and to "repair" PVS data from specific viewpoints.
- RenderWare Visualizer can be used to display PVS.

Using RpPVS alone

Generating PVS data using the **RpPVS** plugin requires the following steps:

1. Attach **RpPVS** plugin
2. Setup callbacks for progress reporting
3. Setup callback for PVS generation. Use either own algorithm or **RpPVSGeneric()**
4. Call **RpPVSConstruct()**
5. Write the converted world to a stream using RenderWare Graphics' binary stream API

Using RtSplinePVS

Generating PVS data using the **RtSplinePVS** toolkit requires the following steps:

1. Attach both **RpPVS** and **RpSpline** plugins

2. Link against the **RtSplinePVS** toolkit
3. Prepare the spline(s) you intend to use for the generation process
4. Setup callbacks for progress reporting
5. Call **RtSplinePVSConstruct()** for each spline to generate the PVS data
6. Write the converted world to a stream using RenderWare Graphics' binary stream API

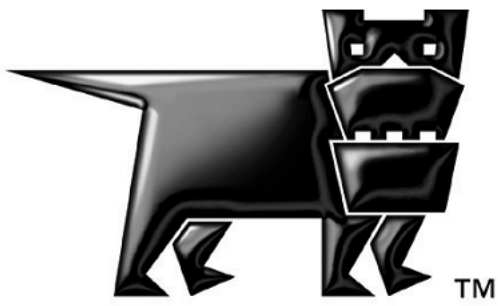
28.4.3 Rendering

The steps needed to render worlds with PVS data are:

1. Attach the **RpPVS** plugin and any others you require (**RtSplinePVS** is not required for rendering)
2. Load your worlds
3. Use **RpPVSQuery()** to check worlds for PVS data if necessary
4. For worlds that have PVS, remember to hook **RpPVS**' PVS fast culling system into the rendering engine using **RpPVSHook()**
5. Just before rendering, set the view position using **RpPVSSetViewPosition()**
6. Unhook the PVS culling system using **RpPVSUnhook()** when done

Chapter 29

Geometry Conditioning



29.1 Introduction

A world can be constructed in a variety of different configurations of topological and aesthetic primitives whilst each variation might look identical. The performance of a 'good' world over that of a 'bad' world can be significant. Good artwork is the principal key to achieving a good world, ultimately leading to performance optimized rendering, and knowledge of tri-stripping, vertex pipelines, and efficient use of UV values, to mention just a few, are important issues that must be addressed. For advice regarding the construction of good scenes, see the white paper titled, *Optimizing Static Geometry*. Whether a scene is good or bad, the geometry conditioning toolkit, **RtGCond**, and its partner toolkit, **RtWing**, have been designed to improve artwork.

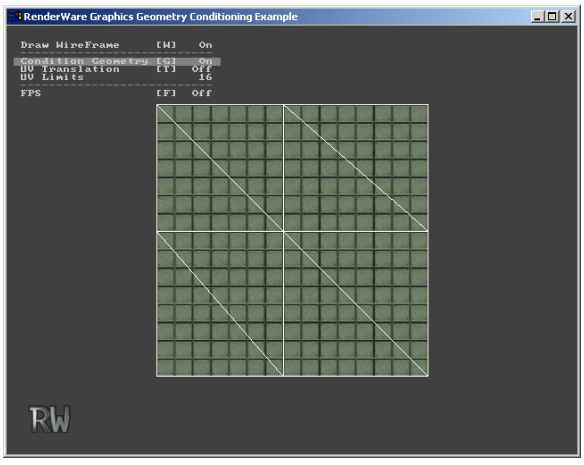
Not all scenes will benefit from geometry conditioning while some might benefit from only a small subset of geometry conditioning functions. But many scenes can benefit enormously from being conditioned.

The geometry conditioning toolkit is standalone, and takes a simple format of vertices and polygons. It is also called from **RtWorldImport** and the exporters if required.

In the remainder of this document, we shall see an overview of geometry conditioning; how to use geometry conditioning with **RtWorldImport**; how to use **RtGCond** and **RtWing** from the API; and finally how to write a custom geometry conditioner using the toolkit and the geometry conditioning pipeline mechanisms it hosts.

29.1.1 Examples

There is an example, found in **examples/gcond**, that illustrates some of the issues in this Chapter, namely polygon welding and UV-translation:



GCond Example

29.1.2 Other Documentation

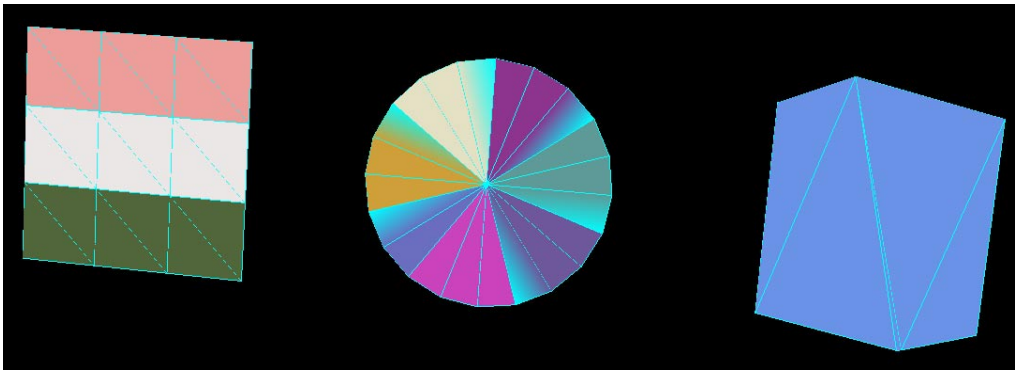
- See the API reference for details of the code for **RtGCond, RtWing** and **RtImport**.

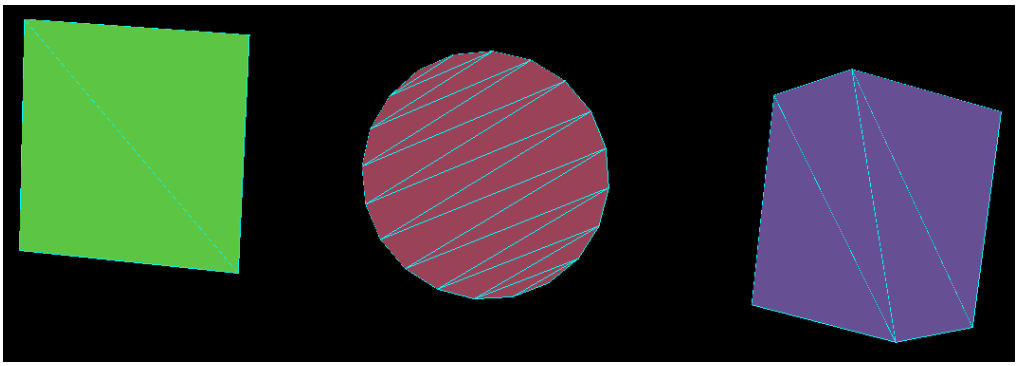
- This chapter assumes you are familiar with the workings of **RtImport**. Details can be found in the User Guide chapters on *World and Static Models*.

- There are further details regarding the optimization of geometry in the *Optimizing Static Geometry* white-paper.

29.2 Overview

To begin with, let's consider the follow three primitives. They are color coded according to how they are tri-stripped. The original art work is shown in the first figure, whilst post-geometry conditioning is shown in the second:





- The 3x3 patch comprises 18 triangles and three tri-strips in the original artwork; it has been rearranged to comprise just two triangles with one tri-strip.
- The circle approximation originally designed as a tri-fan does not tri-strip efficiently; it has been rearranged such that only one tri-strip is necessary.
- The irregular shape has a sliver in the original artwork that could lead to poor pre-lighting and texturing; it has been rearranged without the sliver.

This example shows three topological rearrangements to illustrate the potential of geometry conditioning. Using **`RtGCond`** and **`RtWing`**, these operations can be performed automatically. (Note it should still be the desire of the artist to build efficient scenes, as the geometry conditioner is limited in its intelligence. For useful information on optimizing scenes, refer to the *Optimizing Static Geometry* white-paper.)

29.3 API Details

In this section, we shall look at the most common use of the geometry conditioners: **RtWorldImport**. This uses geometry conditioning whenever the **RtWorldImportParameters** has **conditionGeometry** set true. In this case, we must provide some functionality before calling **RtWorldImportCreateWorld()**. (See the User Guide chapter, *Worlds and Static Models*.)

It works using the default geometry conditioning pipelines that are supported by **RtGCond**. Here, a pipeline is simply a collection of ordered calls to geometry conditioning functions, and a number of pipelines can be created, attached, and applied to a set of geometry.

RtGCond supplies two of these:
RtGCondFixAndFilterGeometryPipeline(), and
RtGCondDecimateAndWeldGeometryPipeline().

The first function is a collection of low level filters that tidy up the initial geometry, they are such things as vertex welding, sliver removal and UV-limiting.

The second function's principal purpose is to weld triangles together to form fewer, larger polygons. Subsequently, it re-triangulates each of these, with a view to efficient tri-stripping. Note, this only welds together polygons that are coplanar and conform to certain parameters discussed next – it is not suitable for LOD, for example.

29.3.1 Setting up a Geometry Conditioning Pipeline

We'll now see how we can set up the pipelines and look at the parameters they take.

First, we need to set up our own pipeline. This is simple since it's just a collection of the provided pipelines mentioned above:

```
void
GeometryConditioningPipeline(RtGCondGeometryList *geometryList)
{
    RtGCondFixAndFilterGeometryPipeline(geometryList);
    RtGCondDecimateAndWeldGeometryPipeline(geometryList);
}
```

We do not need to concern ourselves with the structure of **RtGCondGeometryList**, in this section, since **RtImport** deals with that, but it is simply a collection of vertices and polygons.

We can then set the pipeline by a call to **RtGCondSetGeometryConditioningPipeline()**, with our declared geometry conditioning pipeline function.

29.3.2 Setting up Geometry Conditioning Parameters

The pipelines comprise a series of filters that require a number of parameters. These are supported via **RtGCondParameters** that groups them together. These parameters are available to a wide range of functions and are not just limited to these pipelines. In fact, they may be used by custom written pipelines too.

The parameters are:

PARAMETER	DESCRIPTION
flags	Geometry attribute flags that determine whether the geometry has pre-lit colors, textures and normals.
weldThreshold	The orthogonal distance below which (or equal to) a pair of vertices are considered to occupy the same space.
angularThreshold	The angular difference (in degrees) below which (or equal to) a pair of vertices are considered to share the same angle.
uvThreshold	The UV difference (in 1-D textel-space) below which (or equal to) a pair of vertices are considered to share the same UV value.
preLitThreshold	The pre-lit difference (in 1-D RGB-space) below which (or equal to) a pair of vertices are considered to share the same pre-lit value.
PolygonAreaThres-hold	The area below which a polygon is considered to have zero area.
uvLimitMin	The maximum allowable value for any UV coordinate.
uvLimitMax	The minimum allowable value for any UV coordinate.
sortPolygons	TRUE if polygons are to be sorted by their centroid
textureMode[rwMAX TEXTURECOORDS]	For any element set to **rwTEXTUREADDRESSWRAP**, UV coordinates are aligned prior to welding – this helps welding tremendously.
polyNormalsThres-hold	Fractional range which polygons normal area have to fall within to be welded
polyUVsThreshold	Fractional range which polygons uv area have to fall within to be welded
polyPreLitsThres-hold	Fractional range which polygons prelit area have to fall within to be welded

convexPartition-ingMode	The approach to convex partitioning, one of: **rtWINGEDGEPARTITIONFAN** (in preparation for tri-fanning) **rtWINGEDGEPARTITIONTACK** (in preparation for tri-stripping, default and highly recommended) **rtWINGEDGEPARTITIONEAR** (to maximize the size of the triangles most central to the primitive – sometimes useful with hierarchical culling)
decimationMode	The approach to edge-decimation/polygon welding, one of: **rtWINGEDGEDECIMATIONFEW** (minimizes the number of polygons, but some may be long and thin) **rtWINGEDGEDECIMATIONSMALL** (tries to avoid long thin polygons if possible, at the cost of less reduction in number)
decimationPasses	This controls how thoroughly polygon welding searches for and welds polygons. Since the algorithm has a multi-pass approach, this sets the number of passes.

With all *threshold* values, zero represents the most strict matching criteria, whilst a value of minus one disables the test.

We need to initialize the parameters using **RtGCondParametersInit()**, change any that need changing, then set them using **RtGCondParametersSet()** (In addition, we can recall the currently set parameters by a call to, **RtGCondParametersGet()**):

```
void
SetGCondParameters()
{
    RtGCondParameters GCParams;

    RtGCondParametersInit(&GCParams);
    GCParams.flags = rtGCONDNORMALS | rtGCONDTEXTURES | rtGCONDPRELIT;
    GCParams.weldThreshold = 0.001f;
    GCParams.angularThreshold = 1.0f;
    GCParams.uvThreshold = 0.5f/128.0f;
    GCParams.preLitThreshold = 2.0f/256.0f;
    GCParams.areaThreshold = 0.0f;
    GCParams.uvLimitMin = -16.0f;
    GCParams.uvLimitMax = 16.0f;
    GCParams.textureMode[0] = rwTEXTUREADDRESSWRAP; /* 1 texture */
    GCParams.polyNormalsThreshold = 0.01f;
    GCParams.polyUVsThreshold = 0.01f;
    GCParams.polyPreLitsThreshold = 0.01f;
    GCParams.decimationMode = rtWINGEDGEDECIMATIONFEW;
    GCParams.convexPartitioningMode = rtWINGEDGEPARTITIONTACK;
    GCParams.decimationPasses = 5;
```

```
    RtGCondParametersSet(&gcParams);
}
```

29.3.3 UserData CallBacks

Since the pipelines and many of the functions modify the vertices and polygon, there are a series of callbacks that can be provided to maintain the user data associated with each primitive.

`RtGCondSetUserdataCallBacks()` takes callbacks for vertex cloning, vertex interpolation, polygon subdivision, vertex removal, and polygon removal.

29.4 Advanced API Details

RtWorldImport takes care of most of the geometry conditioning set up for us. If we want to condition our geometry without it, we must consider some of the other API functions.

29.4.1 The Basics

We've already seen how to group together pipelines. However, to action the pipeline on our geometry list, we need to call **RtGCondApplyGeometryConditioningPipeline()**. In addition, we can recall the currently set pipeline with a call to **RtGCondGetGeometryConditioningPipeline()**.

29.4.2 Allocating Data

The **RtGCondGeometryList** contains all the information about the geometry and can be allocated with three functions: **RtGCondAllocateVertices()**, **RtGCondAllocatePolygons()**, **RtGCondAllocateIndices()**. The first allocates space for a contiguous list of vertices, the second for polygons, and the third for the indices of each polygon that index the vertices. In addition, **RtGCondReallocateIndices(),RtGCondReallocateVertices()** and **RtGCondReallocatePolygons()** is provided. Subsequently, these can be freed using **RtGCondFreeIndices()**, **RtGCondFreeVertices()** and **RtGCondFreePolygons()**.

29.4.3 Custom Pipelines

Sometimes, we will want to condition our geometry, but we might want to do it significantly differently to the way the provided pipelines work. For example, we might want to weld vertices and remove slivers, but do no other work. To do this, we can simply create our own pipeline – a linear collection of function calls.

For convenience, each function that can be an integral part of a geometry conditioning pipeline, has the "PipelineNode" suffix.

In our first example, we'll look at **RtGCond** alone. Later, we'll look at **RtWing** too.

Vertex welding

A pipeline takes the form:

```
void PipelineName(RtGCondGeometryList *geometryList);
```

The body of the pipeline that welds vertices (removing slivers en-route) and limits UV-coordinates might look something like this:

```
    /* Get the conditioning parameters, set elsewhere */
    RtGCondParameters* params = RtGCondParametersGet();

    /* Weld together virtually matching vertices (two passes) */
    RtGCondWeldVerticesPipelineNode(geometryList,
                            GCParams->weldThreshold,
                            -1.0f, -1.0f, -1.0f,
                            TRUE, FALSE, FALSE, FALSE);
    RtGCondWeldVerticesPipelineNode(geometryList,
                            0.0f,
                            GCParams->angularThreshold,
                            GCParams->uvThreshold,
                            GCParams->preLitThreshold,
                            FALSE, TRUE, FALSE, TRUE);

    /* Remove slivers and redundant polygons */
    RtGCondRemoveSliversPipelineNode (geometryList);
    RtGCondRemoveIdenticalPolygonsPipelineNode (geometryList);

    if (params->areaThreshold > 0.0f)
    {
        /* Remove anything that wasn't classified as a sliver
           but is still undesired */
        RtGCondCullZeroAreaPolygonsPipelineNode (geometryList,
                                    params->areaThreshold);
    }

    /* Make sure no vertex is used more than once,
       since this is a prerequisite to UV-limiting */
    RtGCondUnshareVerticesPipelineNode (geometryList);

    /* Limit the UVs */
    RtGCondLimitUVsPipelineNode (geometryList, params->uvLimitMin,
                                  params->uvLimitMax);

    /* For efficient tri-stripping, make sure polygons share
       vertices - (was undone by calling RtGCondUnshareVertices) */
    RtGCondWeldVerticesPipelineNode (geometryList, 0.0f, 0.0f,
                                0.0f, 0.0f, TRUE, FALSE, FALSE);

    /* Make sure the work done is validated for RenderWare */
    RtGCondUnshareVerticesOnMaterialBoundariesPipelineNode(
                                                geometryList);
    RtGCondSortVerticesOnMaterialPipelineNode(geometryList);
}
```

So a pipeline is just a collection of function calls that take a **RtGCondGeometryList** pointer and a set of control parameters.

Note, **RtGCondWeldVerticesPipelineNode()** is a very versatile function, and we have called it twice at the beginning. The first pass moves vertices within the weldThreshold together regardless of their other attributes – this is to avoid introducing holes in the scene. The second pass then welds together those vertices that have been grouped and that match the given criteria.

Note, in the example, the order of function calls is important. Vertex welding (**RtGCondWeldVerticesPipelineNode()**), before sliver removal (**RtGCondRemoveSliversPipelineNode()**), will identify and remove more slivers than if the other way around. For example, in the first figure, the primitive with a sliver is only removed if the vertices that form the smallest side are mapped onto each other, and that is only done by vertex welding.

Note also the prerequisite for UV-limiting (**RtGCondLimitUVsPipelineNode()**) – that no vertex is shared by more than one polygon (**RtGCondUnshareVerticesPipelineNode()**) – if the prerequisite isn't considered, then UV warping might, in some cases, be noticed.

The vertices are then once again welded, with zero tolerances, to make sure tri-stripping is efficient; any initial sharing was of course undone earlier as the UV-limiting prerequisite.

Finally, the vertices are sorted according to texture.

Polygon Welding

Now let's look at a pipeline that makes use of **RtWing**. Here we see a pipeline the welds coplanar polygon faces:

```
void PipelineName(RtGCondGeometryList *geometryList)
{
    RtGCondParameters* params = RtGCondParametersGet();
    RtWings wings; /* The winged-edge data structure */

    /* Polygon welding remaps UV values--this is a pre-requisite */
    RtGCondUnshareVerticesPipelineNode(geometryList);

    /* Create winged edge data structure */
    RtWingCreate(&wings, geometryList);

    /* Weld polygons */
    RtWingEdgeDecimation(&wings, geometryList);

    /* Triangulate the resulting welded polygons */
    RtWingConvexPartitioning(&wings, geometryList,
                             params->convexPartitioningMode);
```

```
    /* Destroy the wings, leaving just the geometryList in tact */
    RtWingDestroy(&wings);

    /* For efficient tri-stripping */
    RtGCondWeldVerticesPipelineNode (geometryList, 0.0f, 0.0f,
                                0.0f, 0.0f, TRUE, FALSE, FALSE);

    /* Make sure the work done is validated for RenderWare */
    RtGCondUnshareVerticesOnMaterialBoundariesPipelineNode(
                                                    geometryList);
    RtGCondSortVerticesOnMaterialPipelineNode(geometryList);
}
```

Here, again we have unshared the vertices as a prerequisite to polygon welding – to improve welding results. This step also helps edge decimation which aligns UV coordinates contiguously if **textureMode** is set appropriately.

The call to **RtWingCreate()** sets up all winged edge data structure references. The structure, formally, is a modified half-edge data structure, which allows the representation of non-manifold surfaces. It is also augmented by neighbor tags which determine, if there is a neighbor, whether it is continuous (in UVs, Normals, etc.) or creased.

Edge decimation is then performed with **RtWingEdgeDecimation()** on the data, maintaining the internal data (geometry list) as it proceeds. Ultimately, the geometry list, which comprises a number of polygons, is triangulated using **RtWingConvexPartitioning()** in a manner to promote good tri-stripping. Finally, the winged-edge data structure is no longer needed and destroyed with a call to **RtWingDestroy()**. In addition, the vertices are rewelded to enhance tri-stripping further.

29.4.4 Utilities and tools

In addition to the pipeline functions, there are a set of lower-level tools (both in **RtGCond** and **RtWing**) to help you write your own geometry conditioning functions, whether they be standalone or part of your own pipeline. As well as the functions we've already seen in the examples above, there are a number of additional functions that can be used as part of a pipeline.

RtGCond tools

Let's say we just wanted a pure vertex filtering pipeline: we can, in addition to what we've already seen quantize vertices and their UVs with **RtGCondSnapPipelineNode()** and **RtGCondSnapUVsPipelineNode()**.

If we wanted to work on just polygons, we still need to consider vertices along the way. For example, unused vertices should be removed:

RtGCondRemoveSliversPipelineNode() is a useful function. While it might appear that **RtGCondCullZeroAreaPolygonsPipelineNode()** would resolve these, some slivers cause difficulties for the effective calculation of area, and therefore should be called as one of the first filters in a pipeline.

RtGCondRemoveIdenticalPolygonsPipelineNode() is for poor artwork – and useful as an extra fail safe. It is useful if there are polygons in the scene that occupy the exact same space. It avoids subsequent problems with UV coordinates and polygon welding, and of course stops Z-fighting. NB: It cannot identify overlapping polygons which have different vertex positions, so care needs to be taken in the original artwork.

Any function that removes polygons, and therefore potentially makes some vertices redundant, makes a call to **RtGCondRemapVerticesPipelineNode()** to remove them. NOTE: If you write your own function, this is a post-requisite of a filter.

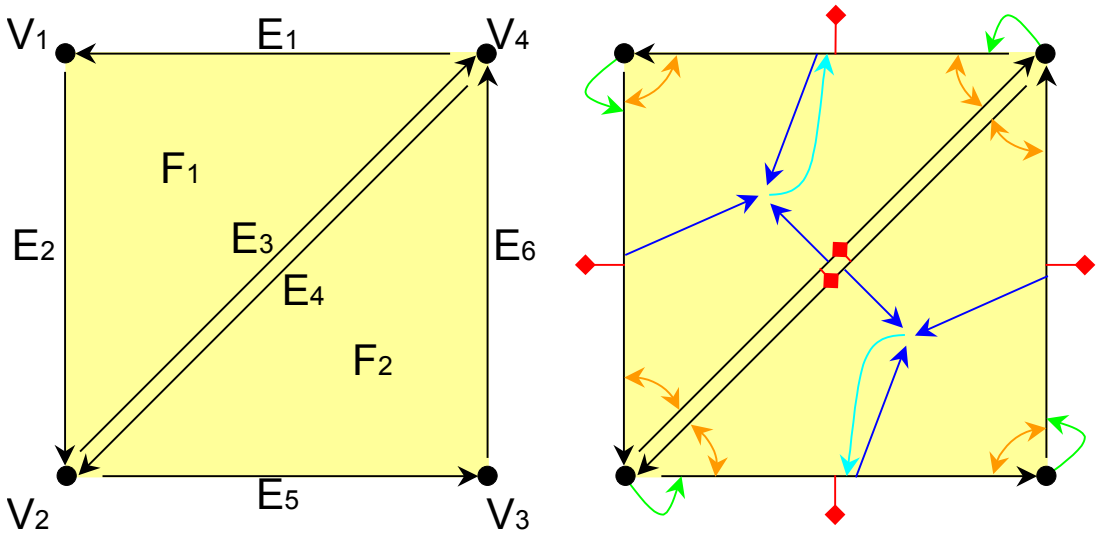
To aid writing your own functions, some utility functions are provided: **RtGCondAreaOfPolygon()**, **RtGCondNormalize()**, **RtGCondLength()**, **RtGCondColinearVertices()** and **RtGCondVectorsEqual()**.

RtWing tools

In addition to those already described, there are a number of other tools provided by **RtWing**. To understand how they work, we'll look at the modified Winged Edge-data structure. More formally defined, the data structure is a half-edge data structure, which is modified to work under non-manifold environments: each edge has a pointer to the previous edge as well as the next one, and neighbor pointers are allowed to be NULL for terminal edges.

In the figures below, we show how the data structure would look for a simple quad, in yellow, that has been triangulated. The first image we use for referencing primitives later; in the second image, the pointers and relationships between the primitive are illustrated: Black circles are the vertices. Black arrows are directed edges. Red arrows are pointers to neighboring edges, or NULL if there is none. Orange arrows are the doubly-linked previous/next pointers that cyclically define the primitive. Green arrows are a reference from a vertex to one edge emanating from it. Dark-blue arrows are pointers from an edge to a face it bounds. Light-blue arrows are pointers from a face to one of its edges.

In addition to this, we hold a neighbor relationship tag to say whether the neighbor to an edge is NULL, continuous (as in the example below), or is a crease – which would be the case if the quad below was folded along the shared boundary so that each face had its own plane. (Creased would also be the case if the faces differ in surface normals, UV coordinates and pre-lit colors.)



There is a set of callbacks for traversing and querying this data structure, namely:

- **`RtWingFaceForAllEdges()`**: With this function we could discover all vertices that define the face. If the face were F1, it would reveal edges E1, E2 and E3, and each of those edges point to the next vertex in the primitive.

- **`RtWingFaceForAllFaces()`**: For F1 this would only reveal F2, since that's all it's connected to in the example, but this is useful for traversing primitives such as sphere approximations, and can be used for mesh decimation.

- **`RtWingVertexForAllEdges()`**: For V1 this would reveal E2 and E1. It can be used for vertex decimation.

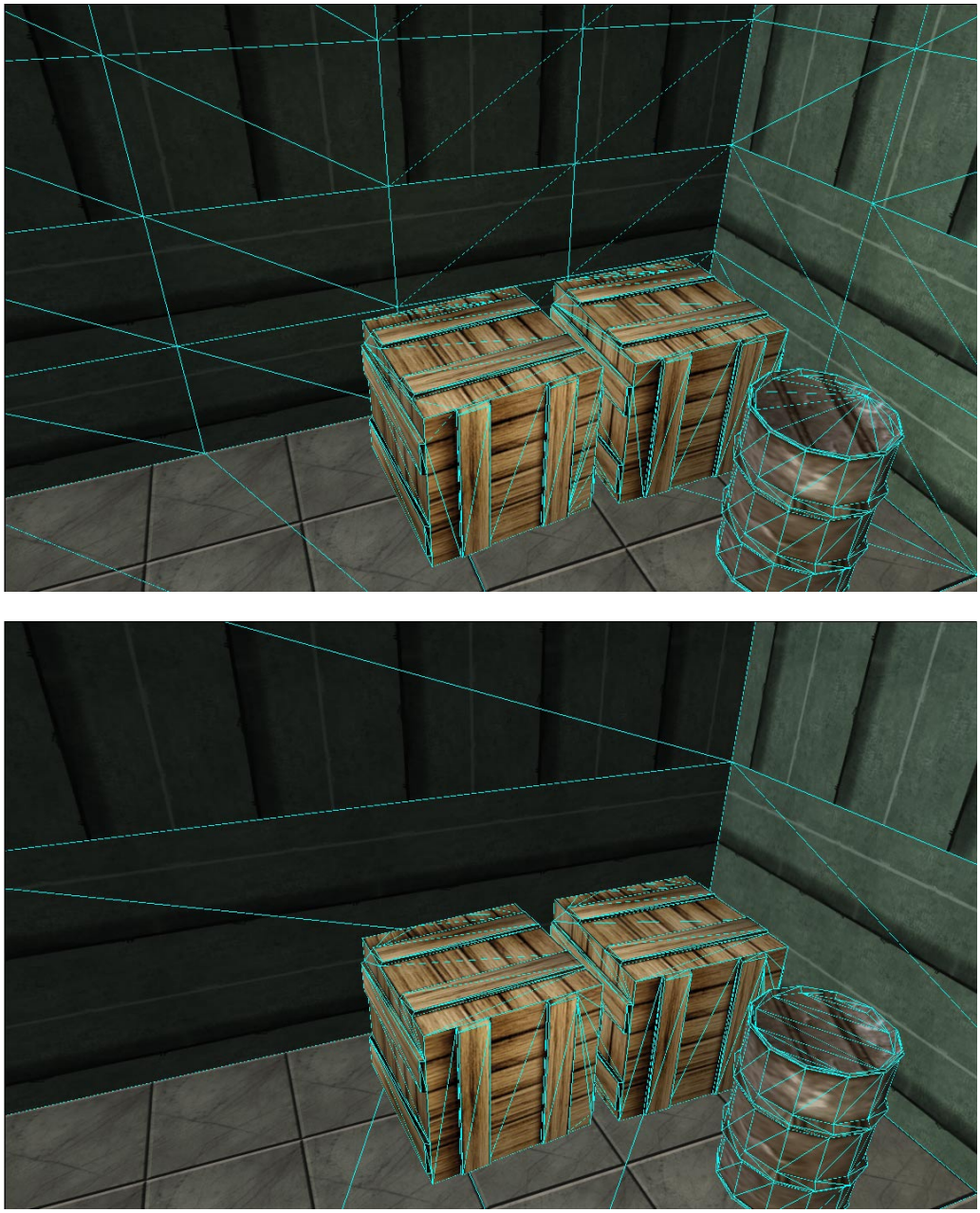
- **`RtWingVertexForAllFaces()`**: For V1 this would reveal F1 only. We could use this to color all polygons that use the vertex.

Since the winged edged data structure simply sits on top of **`RtGCond`** geometry, the two representations need to be kept in synchrony; **`RtWingUpdateInternalRepresentation()`** is proved to maintain the validity of the internal (**`RtGCondGeometryList`**) geometry and the winged-edge (**`RtWings`**) representation.

Finally, **`RtWingPartitionPolygon()`** is provided. It retriangulates any non-triangular faces.

29.5 Summary

The following screen shot, before and after geometry conditioning illustrates some of the issues discussed in this chapter:



Geometry conditioning, if used appropriately, can be a valuable tool. It can remove anomalies that may have visible effects, reorganize polygons and vertices to aid more optimal tri-stripping, and it can reduce the number of polygons in the scene significantly whilst keeping the aesthetic reaction to a minimum.

Index

Index

Page numbers in bold face indicate the most important reference to the subject, where multiple references exist. The page numbers shown below refer to Volume III of the User Guide.

3

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W