Making 3D GAMES with Reality Factory

3D Games Without Programming

Dan Valeo

Making 3D Games with Reality Factory

Dan Valeo

Bookshock Publications

Making 3D Games with Reality Factory

by Dan Valeo

Published by:
Bookshock Publications
Toronto Ontario

Copyright © 2003, Bookshock Publications

ISBN 0-9687185-4-X

All rights reserved. The purchaser of this electronic publication may store one copy in electronic format and one printed copy. Any more copies of this book stored in any means, is a violation of the purchaser agreement and international copyright laws. No copies of this book may be distributed in any format. This title is not freeware or shareware. This publication is not for resale. The author and the publisher does not resume any responsibility for use or misuse of any information provided within this book.

Contents

Introduction to Reality Factory • 3
Level Planning • 13
Game Planning • 21
Using the World Editor • 27
Building a Level - Space Escape • 35
Lighting the Level • 53
Adding Models and Doors • 59
Creating Actor Files • 73
Bringing Actors into RF • 81
Adding First Person Weapons • 89
Creating Enemy Pawns • 99
Introduction to Pawn Scripting • 107
Writing Pawn Scripts • 117
Space Escape - Stage 5 • 153
Heads Up Display • 163
Adding a Puzzle • 171
The RF Menu • 177
Space Escape - Finishing Touches • 189
Packaging a Demo • 195
Advanced Level Design • 203

Introduction to Reality Factory

Game design in one form or another, is a hobby shared by millions of people world wide. Some people even have the benefit of creating games as a career. Whatever the purpose though, these people all use a set of selected tools to help them accomplish this. In many instances, one of these tools is referred to as a Game Engine. This is the core of the game and its function is to establish a system, that can create a game from a simpler level of programming and utilize specific media for the game. Reality Factory is a generic game engine, that is easily configured with little or no programming at all.

Reality Factory was devise by two individuals named Ed Averil and Terry Morgan. Ed Averil has since then moved on to other things and the engine is now currently managed by Terry Morgan, Ralph Deane and Mike Wuetherick. Their efforts have allowed Reality Factory to develop into a very sophisticated game engine, that is freely available to almost anyone world wide. The original purpose of Reality Factory, was to give non-programmers the ability to very quickly create a game demo, that can be presented to game publishers. It serves this purpose very well. Since its first release though, RF has become a very powerful engine, that is capable of creating a complete game that can be published entirely on its own.

The Reality Factory game engine is capable of creating 3D role playing and action / adventure games. It can create games that run in First Person mode, similar to the Quake and Doom style of game play. It can create third person views where the player stands behind the character, isometric views similar to those found in many sim games and a fixed camera system similar to the original Resident Evil view style. Recently, another addition has allowed for the creation of 2D style racing games. RF becomes more and more versatile with every update.

Reality Factory and Genesis 3D

Reality Factory is an open-source game engine. The term open-source is given to the license agreement that RF abides by. It stems from the agreement imposed by the Genesis 3D game engine, from which RF is built upon. The license agreement for Genesis states that the engine can be used freely as long as the Genesis logo is displayed at the start of a game and that any changes to the source code be made public. Because the source code is open to the public, the license is referred to, as an open-source license. To remove the Genesis 3D logo, a license fee must be paid to the current owners of the Genesis3D engine.

Reality Factory has added much to the original Genesis 3D engine including the ability to use larger sized textures and a faster 3D rendering engine. This was contributed to the engine by Wendel Buckner. Thanks to the dedicated contributions of Ralph Deane and a few others, the RF game engine now surpasses the original Genesis 3D engine in many ways.

How RF Works

Reality Factory can be best looked at, as a ready to run game, that can be completely configured by the user. If RF is executed right after it is installed, it will run the demo game that it is configured to run by default. The demos usually change with each major release of the engine. As of this writing, the current release is RF07. The Reality Factory game engine is a stand alone executable, which means that it can be run from any writable folder it is in. It needs no installation or system registry entries. A person can have as many copies of RF on their system as they like, each developing into a different game.

The Reality Factory engine itself can be broken down into four main components. These are:

- The engine itself - RealityFactory.exe
- The configuration files, located in the INSTALL folder.
- The media files, stored in the MEDIA folder.
- The level that is loaded by the engine.

These components combine themselves to create a game. They are tightly integrated and changes to one will usually effect the others. By making specific alterations to these components, the user can modify the default demo game and make it a completely new game. To distribute the game, all that needs to be done is to redistribute the folder which contains the modified components. On the following page is a diagram that details the integration between the basic components of the engine. This diagram covers only the first few layers of integration. As each specific component is

Reality Factory Web Sites

www.realityfactory.ca

This is the main RF site. It has the official release as well as demos, some tutorials and access to the main RF forum. It should be the first stop for anyone who is interested in using RF.

www.rfactory.org

This site is managed by Ralph Deane and is used as a place to quickly find the newest updates before they become an official release.

RFDN
rfdn.realityfactory.ca

The Reality Factory Developers Network, is a site for professional developers, offering a variety of media libraries, advanced tutorials and various articles regarding advanced techniques suited for RF. It is a members only site but is very affordable and well worth a member-ship. The RFDN is a vast collection of RF resources, that makes for an indispensable site to game developers.

Basic Structure of an RF Game

examined throughout this book, they will be detailed more closely. With Reality Factory, any changes to one of the components in the diagram above, can have an effect on another component that it is directly connected to. For example. If the Player Actor is replaced with another actor, then this will require that changes be made to the PlayerSetup.ini configuration file. This file determines the animations within the model used as the player. This works both ways.

Installing Reality Factory

The RF07 installation file is named RF070.exe. It is as easy as double clicking it and letting it install to the default directory named realityfactory070. Download the RF07C update and just unzip it to the realityfactory070 directory. It will overwrite the necessary files which need updating. Before

running RF for the first time, open up the Reality Factory folder and examine the directory structure. This is the default system which RF uses to locate the necessary media and configuration files. It is not set in stone and can be defined within the main RealityFactory.ini file. Here is the definition from the main ini file:

- LevelDirectory=media\levels
- BitmapDirectory=media\bitmaps
- ActorDirectory=media\actors
- AudioDirectory=media\audio
- AudioStreamDirectory=media\audio
- VideoDirectory=media\video
- MIDIDirectory=media\midi

These definitions determine the directories where Reality Factory will locate the configuration and media resources. The folders which can not be defined, are those under the media folder that are not listed above and the Scripts folder. These are set and can not be changed.

In order to create a new game, the files within these folders are modified or replaced. The tools that are required to do so, also come with Reality Factory. They are stored within the Tools folder. As each tool is required, it will be examined in more detail. Below is a short overview of each.

The Reality Factory Tools

The utilities that come with RF can be broken down into two sections, the World Editors and the Tools. World Editor 2 is examined in a later chapter. For now a brief look at

the five most frequently used Tools will suffice.

- **RF INI Editor** - The INI Editor is a graphical interface to the options defined in the RealityFactory.ini file. Through this, the user can define the game level, the player actor and the menu configuration file. It also allows for the selection of options that will be useful during the level debugging process.
- **Actor Studio** - Models which are to be used in Reality Factory are referred to as Actors. RF requires that they be in the .ACT file format. The Actor Studio can build act files from other formats. A later chapter is dedicated to the creation of actor files for RF.
- **Actor Viewer** - This is a simple interface to the Genesis engine which allows the user to view act files and their animations. A newer release named Actor Viewer++, is capable of creating animations by blending two other animations. The newer version is not a part of the official RF package, but can be downloaded on-line.
- **Font Maker** - The Font Maker tool will generate fonts that can be used in RF, from the True Type fonts that are installed in the computers OS. The resulting bitmaps can then be altered in a third party graphics program, to add effects to the text such as an embossed look or a pattern overlay.
- **TXL File Editor** - Reality Factory levels use a unique texture format with the extension .TXL. These are similar to the WAD files which were made popular with Doom. TXL files are not distributed with the final game and are only used by the World Editors, to compile the BSP file which makes up the level. To create a TXL file for use with Reality Factory, all that is required is to drag and drop the

texture images into the TXL file editor window. The TXL file is then saved in the media\levels folder.

Running Video Setup

Before running Reality Factory, it is important to run the Video Setup Program that it comes with, in order to set the video driver and the resolution. The 32 bit D3D driver may not work on all computer systems. The 16bit normal driver, can be used with most systems. The Video Setup program will determine if it can run, before it is set. By default the resolution is set to a fullscreen 800 x 600. It is important that when a new game is designed for Reality Factory, its final resolution be determined before hand. This is because many of the components must be designed with the final resolution in mind. When the final game is distributed, it is important to also include the Video Setup, so as to allow players to select between 16 bit or 32 bit drivers.

Running Reality Factory

Once RF has been installed, the default demo level can be tested out. First the player will be presented with an interactive menu. Check a few areas of the menu. A fade effect will appear as the screens change. This is controlled in the menu.ini file. The entry defined as; fadetime=750, determines the duration of this effect. In this case it is made to last for 750 miliseconds.

There are many features and control options that are available with Reality Factory. Before starting the demo, be sure to check the Options category in the menu and then see

Dealing with RF Updates

Updates to Reality Factory are constant and very frequent. It is important to keep up to date with the fixes and technological enhancements that are constantly being applied to RF. Updates come in ZIP format and to use them, just unzip the file into the RF root directory. If prompted to overwrite an existing file, then select yes. The ChangeLog file will list the changes made to an official update. As of this writing the latest update is RF07B.ZIP.

the section named Controls. This will display a scroll window with the keys available in Reality Factory. These controls are configured with this menu. Any changes to the default setup, is stored in a keyboard.ini file. This file is not stored in the install directory but is instead located in the root directory.

By default, the audio levels are raised to half level and may need adjusting to suit the particular likings of the player. RF has options for both the sfx and the music volume. The Audio menu is found through the Options menu.

RF also has a Difficulty Menu and a Character Selection menu. These are displayed after the Start button has been selected. By default however, the options to display these are not checked, so the demo will not display them. These options can be set through the RF INI Editor.

Once the game has been started, a loading screen will be displayed and then the level will play once it has been loaded. The level should start in first person view and a HUD displaying the weapons ammo, should be overlayed on the screen. To play the demo, pick up the ammo packs and switch to a weapon that fires. The default weapon is a melee weapon. Weapons are selected with the 0-9 keys. Fire upon the flying monster and be careful to dodge the flames it fires back. Once the monster has been defeated, the player enters the teleporter to the next room. To quit a level, hit the ESC key. This will return the game to the starting menu.

The RF Documentation

Before beginning any game development with RF, it is

imperative that the documentation that comes with the main RF installation, is read over thoroughly. The ‘docs’ explains the use of the entities, the pawn scripting commands, methods of creating weapons and effects and much, much more. The side bars found throughout the book, will constantly be referring to the off-line documentation. During any development with RF, the docs should be open and ready for quick access. They should be used as a reference for every modification made to the config files and when adding an entity to a level.

RF also comes with a file that documents the changes made to the engine with each release. This file is named changelog.doc and is found in the RF root directory. Before adding an update, this file can be examined within the zip of the update, to see what changes have been made before any overwrites take place.

Trouble Shooting RF

If RF ever fails to start or crashes once it has started, an error will be logged to a file in the root directory named, RealityFactory.log. If there are no errors during operation, then an examination of the RealityFactory.log file will read something along the lines of:

```
File open attempt failed on type '9', file '.\keyboard.ini'
File open attempt failed on type '9', file '.\savedgames.rgf'
Game started
Game exiting
```

The first line states that the keyboard.ini file could not

be found. This is normal since the keys are not re-configured. The second line states that the savegame file could not be found. This is also normal since no games have been saved. So the game starts and logs this event. No errors were reported during game play and the final line states that RF has shut down cleanly and successfully. If an error occurs during runtime, then the error log might be read as follows:

```
File open attempt failed on type '9', file '.\keyboard.ini'
File open attempt failed on type '6', file
'media\bitmaps\menu\credit.bmp'
Missing Bitmap menu\credit.bmp or Missing Path
```

In this instance, the log file states that a file named credit.bmp, could not be found in the specified directory, media\bitmaps\menu. This type of error is the most common a person will run into. Be sure to check the RealityFactory.log file when ever an error occurs. If nothing has been logged to the file, then a critical system error has occurred and must be corrected another way.

Level Planning

Game developers want to put as much imagination into their creations as they can. But a designer must conform with the limitations of the engine that they are using. By considering these during the design process, a developer will be able to plan the level to compensate for these limitations. This chapter will discuss what RF is capable of and how the design process should consider this.

The Environment

The first consideration for a designer is if the engine can support the environment outlined for the game. RF can handle large indoor levels very efficiently. A level that keeps the player going from ‘room to room’, is easily handled by the engine, because such a small area of the level is rendered at any point during the game. (One room at a time.) Indoor levels can obtain the appearance that they are within a large outdoor environment, by creating windows and ceilings that display a SkyBox. More about this in a later chapter.

RF also allows for the creation of large outdoor levels, due to a feature which allows for a set clipping distance. When combined with a Fog effect, the two components interact to create a system that prevents the engine from wasting time rendering objects in the distance, while giving

the effect that they are slowly coming into view. The chapter on advanced level building discusses methods for creating large outdoor levels.

For an indoor level, it is important to consider the following factor. The default scale for the RF Ernie player actor requires a minimum world height of 128 texels and will be able to crouch and jump over, objects that are 64 texels from the ground. The scale of the player actor will ultimately determine the size of all other objects used for the level. Important design sizes include the dimensions of doors and stair case openings.

For outdoor levels, it is important to consider the maximum viewing distance of the player. This is controlled by the Fogging and Clip Plane features and determines the amount of brushes that will be rendered by the engine at any one time. The shorter the viewing distance, the more efficient the rendering of the level will be.

The Player Actor

Almost everything in the level will revolve around the player actor in one way or another. Obviously, this is because the player actor is the interface between the game player and the level. One will have to consider many things but the most important is the final view of the game. Will it be a first person shooter, a third person view, an isometric view or a fixed camera system. The simplest to develop is a first person shooter, since it requires the least modification to the main playersetup.ini file and the least animations. The game outlined throughout this book is a first person shooter. There

are many other considerations with the player actor including:

- **Actor Size** - The player must fit in the level. The ultimate scaling of the player and the other actors in the game, will determine how seemingly large the level is.
- **Actor Bone Names** - These will be referred to in many instances and should be written down for quick reference.
- **Actor Animations** - A first person shooter requires no more then six animations applied to the actor, while a third person view will require more then sixteen animations minimum. First person shooters also only reveal the weapon in view, making them very easy to model for.

This image is of a door face in the game Sons of Mars. The brush has a bottom width of 128, a top width of 64 and a height of 128. In the corner of the image is the texture that is applied to the face. It was made to match the dimensions of the door brush.

The Texture Library

All textures used in RF must be a power of 2. This means that the dimensions of the texture must be a number that can divide itself down to 2. For example, 2, 4, 8, 16, 32, 64, 128, 256 and 512 are all valid dimensions. The images do not have to be square. 128 x 64 is a valid texture size. Since textures must abide by this rule, it is logical to build the level with brushes that also use the power of two rule. For instance, if a house was going to be modeled from a 128 x 128 x 256 sized brush, then a texture that is 128 x 128 can be applied perfectly. The image in the side bar is of a door face in the game Sons of Mars. The brush has a bottom width of 128, a top width of 64 and a height of 128. In the corner of the image is the texture that is applied to the face. It was made to match the dimensions of the door brush.

The actual look of the textures is very important as well. They will have the greatest effect on the mood of the

level. It's the paint on the walls, so to speak. In order to keep things consistent from level to level, each should be built using the same set of Base Textures. A base texture, is an image that is repeatedly used throughout the level. Any structural brushes such as walls, ceilings and floors all use base textures. As more detail is required, these base textures are modified to create new textures that are meant for specific positions within the level. For example, a door texture is made from the wall base texture or a room with a burnt wall, should also have its textures made from the wall base texture.

The base textures are so important that they should be made prior to any level construction or made as needed during the construction of the level. The best method is to plan and create the base textures and then to go and design the basic structure of the level. As detail is added to the level, the base textures can be modified and added to the library as needed.

Professional game design teams leave base textures to the texture artists. They develop all of the internal game play using only a single, brightly colored texture that offers good contrast against the play elements. However, as a small indie design team, it is better to use base textures as the level is built. It will help to convey the design atmosphere to the texture artist, which can help to reduce the necessary concept art. It also allows for a game play demo to be quickly released for testing by an indie game community. The texturing process can take much longer then the actual level building process and being able to release a demo for testing before the final textures are added, is an excellent means of finding viable feedback to the game play features.

When selecting the base textures, be sure that they convey the right mood and theme of the level. Use textures that tile well and do not have too much detail. RF can use 8 bit and 24 bit bitmaps or 24 and 32 bit TGA files with a transparency alpha channel. The texture library can consist of any combination of these, but by far the best choice is to use 32 bit TGA files or 24 bit bitmaps.

What to Read

Be sure to read the following articles from the RF manual.

- RealityFactory Overview
- 24 Bit Textures
- Bitmaps with Alpha Mask
- Bitmapped Fonts
- Video Setup

Required Game Features

The number one question designers should ask themselves is, can RF do what the level is designed to do? In many instances the answer is no, but this is only due to the grand imagination of designers. The key is to design the level, so as to work with what RF can do. For instance, at this point in time, RF has no support for the 3D driving genre. However it does allow for the capability to create a 2D driving scenario. An example of this is the game Speed City. It was designed in this fashion, to compensate for the lack of 3D vehicle support. However, the original game idea was never changed. Whether it ended up being 2D or 3D, the actual outlined game play was kept the same.

The designers must ascertain the special features or game events that will be encountered throughout the game and then ask themselves if RF can handle it. If RF does not directly support the required feature, then a work around will have to be considered or the feature will have to be added by modifying the code. In order to create Speed City, the code was modified to disable the mouse rotation controls. This has since then, been added to RF by the developers. It is very important to consider this before any real game development

starts. Many developers find themselves dropping a project, because they find that the engine could not handle their design criteria. Things to be considered are the actions of the player, the puzzles, the actions of the enemies and the required graphic effects. Before any actual construction begins, the designer must have a clear outline of how they are going to implement the main features into the game.

Designing a Level

After all of the above mentioned considerations have been carefully looked at, the level design process can begin. A level starts on graph paper. Graph paper is a game designers best friend. The units on a page of graph paper, are used to represent units in the game level. For example, one square on graph paper can represent a 128 x 128 texel area in the game world. It takes a small amount of time to design the level on graph paper and that will translate into a much more efficient construction process.

Start by considering the total size of the level and how many texels long and wide it will be. Have a defined starting point and the points that end the level. Draw each large room or structural wall on the graph paper. Remember to use the specified units of the graph paper, as a guide to the size of the brushes that will be used to build the level. Try to add as much level concept as possible. Mark where the puzzles will be, where enemies should be positioned and the paths that enemies will traverse. Mark off doors and try to find areas in the level that can be sectioned off with them. More about this in a later chapter.

With the level laid out on paper, the designer can now define the necessary dimensions and scale. By using the default actor scale, the base unit for the level should be a 128 x 128 texel area. If it ends up that the level may be too large for the engine to handle efficiently, then the level and all of its components will have to be scaled down. In order to double the efficiency, a level first planned to be 4096 x 4096, can be scaled by 0.5 to have dimensions of 2048 x 2048. The actor, all textures and the enemies will all have to be scaled by 0.5 as well. This will require modification to components within the level and the RF configuration files.

After a level or levels have been put on paper, it still is not time for any construction to begin. What is required now is a complete outline of the game. This will determine what media will be required to build the levels, allowing the developer to assemble the necessary media library, to start building the game with.

Where to Find Media

There are many sites on the Internet dedicated to distributing public-domain and royalty free textures. RFDN has a huge collection of complete texture libraries which come in many formats. RFDN also has many RF ready actors and fabricated objects, which can be used to quickly add detail to levels. The RFDN site is found at: rfdn.realityfactory.ca.

Starting the Media Library

The first necessary media component is the TXL file of base textures. This should be assembled prior to building the level. As brushes are added to the level, they will require a texture to be applied and it is important to have these textures ready for use before starting any level construction. This is about all that is required, to begin building the main structure of a level. It can be previewed using the default RF settings and media can be added to the level as required. However, if the game is a third person game and the original RF actor is not used, then it is best to first model the player actor as well. This will allow the level builder to fine tune the dimensions

before any actual structure is built.

TXL files are built using the TXL File Editor, which is very straight forward and easy to use. Just run the program and drag and drop all properly formatted media into the TXL File Editor list or viewing window. Save the TXL file in the media\levels directory.

This chapter focused on the necessary requirements prior to building a level. In the next chapter, the focus will broaden to examine the necessary media that is required to build an entire game and the stages at which these elements are required.

Game Planning

Designing a game can be broken down into two major steps. The first part is the idea for the game itself. Outlining the premise, the story and the game play features must be done first and be clearly documented. The second part is to take this idea and to create a game with it, in accordance to the demands of the game engine. This chapter will focuses on the second part. The game that is created through the course of this book is a simple First Person Shooter with a few puzzle elements. Outlined in this chapter, is a procedure for creating a FPS demo with Reality Factory.

Stage One

Stage one of the game building process involves determining whether or not RF can reproduce the features required for the game play. The game idea is then altered to accommodate any limitations and a method for applying each feature is outlined. This step helps to realize the feasibility of the game. Take into consideration, everything that was outlined in the previous chapter.

Stage Two

Stage two is the collecting of the minimum media

Stage One Planning the game idea and premise.	Before a game can be made, a game premise must be established first. The game must be thoroughly planned as well as the levels that are to be built. The next process is to turn the ideas on paper into a 3D game.
Stage Two Start the minimum media library.	First build the base textures and TXL library and any other media that will be required for the basic structure of the game. If the game is a third person view, then this includes the player actor.
Stage Three Build the basic level structure and add basic lighting.	Build the basic structure of each level and set the basic lighting. Add as many game play elements that do not use pawns. Create any doors that will be used and any other world models as well.
Stage Four Begin to build the actor media library and add weapon actors to the game.	With the level structure built and the base textures applied, any pawns and enemy actors that are intended for the game should be collected and animated. This includes the player weapon actors.
Stage Five Develop the pawn scripts and add the pawns to the game.	Pawn scripts are developed in a separate small level that is quick to load. Once the scripts are working, the pawns are added to the levels and the scripts are refined to work with the specified game play.
Stage Six Build the Effects Library. Add audio and effects to the game.	At this point, there should be a playable level with interactive pawns. Now the audio is added to the game and the visual effects graphics are enhanced and modified.
Stage Seven Create the media needed to connect levels and connect each level to the next.	The levels are now brought together into a single demo game. Create any splash screens and cinematics that will be used for the demo. Adjust the ChangeLevel entities to link each level.
Stage Eight Create a new menu for the demo.	All that is required now to create a complete demo is a new menu for the game. Prepare the menu graphics and then create a new menu.ini file.
Stage Nine Package the demo for beta testing.	Package the demo and minimize the necessary media. Play through the demo at least once to be sure that everything is included. Change the Icon and the RF exe name and then package a demo for beta testing.

library required to build the basic structure of the level. This is the TXL library of base textures. For this demo, the Basic Media Library can be downloaded from the Bookshock web page. For a unique demo, the texture images should all be placed into a single directory for easy storage and then made into a TXL file using the RF TXL File Editor. Save the TXL file in the RF, media\levels folder. When level construction begins, the first step is to load this TXL file.

Professional game development teams, usually use a single, brightly colored texture, to create the basic level structure. This one texture level, is then handed over to a team of texture artists, who add the final look to the game. A small team of indie developers however, will be better suited to create a library of base textures. There are many advantages to this, that will save a small development team a great deal of time down the road. It also allows for a beta testing demo to be released much more quickly and for a more detailed feedback from the testers.

Stage Three

Stage three is where the level construction begins. First the basic walls, floors and ceilings are created, then any doors are added. The level is taken as far as it can be, adding as much game play that does not involve “player enemies.” The objective is to create a level that can be walked through and examine without being hindered by enemies. When the basic structure has been added, the lighting can be applied the level. Again, the lighting does not have to be very fine tuned at this point, just allowing enough to direct the player path.

Stage Four

Stage four is where level construction stops for a moment and the Actor Library is assembled. This includes all enemy pawns and weapon actors. Now, this stage could have been been performed by a modeler while stage three was in progress, but for a “one person team,” it can be taken care of after the basic world building stage. The library should include all of the animated enemies, the animated weapons, the projectiles for the weapons, any Attribute actors and any Static Entity actors. They should all be made into act files and placed in the RF media\actors folder.

Stage Five

Stage five is actually four separate parts to it. Each will interact closely, so this stage of development is all done within a smaller test level. The four steps to this stage include, writing the pawn scripts, adding the pawns to the level, adding the player weapons and modifying the HUD. This may seem like a lot of work for one step, but the only time consuming work is in the scripting of the pawns. Adding the weapons and modifying the HUD is a quick and simple process in RF. Both are defined in later chapters dedicated to them. Once again, these steps are performed in a smaller and much simpler “test level,” that can easily be loaded and compiled. This way, there isn’t the need to load a complex level, just to make subtle changes and also helps to make developing the pawn script, a much more efficient process.

Stage Six

Stage six is devoted to level esthetics and effects. At this stage, a playable demo level should be ready for sound and visual effects to be added to it. Again, a media library is assembled containing the sound effects, the explosion bitmaps and the final sprites that will be used in the level. Any wall decals which are to be added as level detail is also added at this point. The final result is a playable level with enough visual and audio effects to call a completed and ready for beta testing. However, a game is bit more then a playable level and now these completed levels must be combined into a complete demo game.

Stage Seven

Stage seven involves bringing the levels together and creating any splash screens and cinematics that will be played between levels. Each level for the game demo has the ChangeLevel entities altered to go from one level to the other and display the appropriate media in between. Now the demo has come together and should be playable from start to finish.

Stage Eight

Stage eight is the second last stage of development and involves the creating of a custom game menu. Once again, a media library for the menu graphics is assembled and the menu.ini file is modified to display these custom images as a new menu. This is one of the quicker stages of

development, but should have some consideration as to the quality of the graphics. A nice looking menu screen can help to pass on the game mood and atmosphere. The menu can display such things as, how to play, the game controls and credits. It is important to not skimp, when it comes to building a menu. The menu buttons should be clear to the user and the controls should also be easy to locate. An example of a well designed menu is found n the RF demo game Sons of Mars and an example of a bad menu design, can be seen with the RF demo game, Crypt Spawn.

Stage Nine

Stage nine, the packaging and distribution stage, is the final step to developing a game demo. At this point, the game is brought down to the minimum amount of media needed to run it and the RealityFactory.exe file is renamed and the icon is replaced with a new one. The minimized package is now assembled into an installation file, using one of the popular installation builder programs available. The demo is now ready for the final test run and it can then be distributed for feedback testing.

Using the World Editor

Of the many tools that come with RF, the tool that will see the most use is one of the World Editors. Three such world editors come with RF, but the focus of the book will be World Editor 2. In the opinion of the author, World Editor 2 is the most stable and reliable of three.

3DT and BSP

World Editors create what is know as a BSP file. BSP stands for Binary Space Partitioning and refers to a method of creating 3D geometries, by dividing up a larger area into smaller areas within. A BSP file is the actual level that is loaded by RF. It contains all the geometries and Entities that are recognized by the engine. The BSP format for RF is particular to RF and can not be used with any other engine.

When a world is created using a World Editor, the file is saved under the 3DT format. This is a text file, which contains all of the world information. What the World Editor does is take this 3DT file and compile it into a BSP file. All 3DT and BSP files, are stored in the RF media\levels folder.

RF also allows for MAP files to be imported and used in a level. These can be used to create terrain geometries that can be placed in a level to act like hills. More about using

MAP filesas terrains in a later chapter.

World Editor 2

World Editor 2 was programmed by Michael R Brumm and is distributed as Shareware. If you use World Editor 2 to create a commercial game that will make profit, then it is your legal obligation to send $20 USD to Michael R Brumm. See www.michealbrumm.com/projects.html for information.

The World Editor 2 interface is very different from the other two editors. It is designed to be used most efficiently, by using a mouse/keyboard combination of commands. Before examining these however, start World Editor 2 and take a look at the 4 viewport windows. Below is a diagram which outlines these windows and their relevance to the X, Y and Z vectors of the level being created.

World Editors use three, 2D view ports so that a brush can be manipulated or rotated along any of the three axis. A brush will move or rotate, in a direction that depends on what viewport it is being modified in. The camera viewport in the bottom right corner, is where all the texturing is done. In the camera window, brush faces are selected by clicking on them and the textures are then applied and aligned.

The Grid

The three modelling viewports all display a grid pattern in them. By default, this is made up of 128 x 128 texel areas. Within this grid is a smaller sized grid, which is the Snap Grid. The size if this grid is defined in the Edit menu, under the Snap Settings option. By default, this is sized to 8 x 8 texels and should be kept this way unless a smaller Snap Grid is required. When Snap is set to ON, any brushes made will snap to fit against this grid. This is the number one most critical feature required by a BSP world editor. Without using Snap, the level would end up being filled with ‘leaks’ that will prevent compiling.

Navigating the Viewports

A viewport is activated by selecting a brush within the viewport. The view within the viewport itself, can be moved around by holding down the CTRL key and then, using the left mouse button, click - dragging the view within the viewport. The Camera Viewport, is a bit more intricate. There are basically two means of navigating the Camera View. The first is through the Camera Window using the following control scheme.

- **Forward :** Left Mouse Button + CTRL Key + move mouse forward.
- **Backward :** Left Mouse Button + CTRL Key + move mouse back.
- **Turn Left :** Left Mouse Button + CTRL Key + move mouse left.
- **Turn Right :** Left Mouse Button + CTRL Key + move mouse right.
- **Look Up :** Right Mouse Button + CTRL Key + move mouse forward.
- **Look Down :** Right Mouse Button + CTRL Key + move mouse back.
- **Straight Up :** Both Left and Right Mouse Button + CTRL Key + move mouse forward.
- **Straight Down :** Both Left and Right Mouse Button + CTRL Key + move mouse back.
- **Move Left :** Both Left and Right Mouse Button + CTRL Key + move mouse left.
- **Move Right :** Both Left and Right Mouse Button + CTRL Key + move mouse right.

These controls, navigate the camera through the Camera Viewport. In most instances, this usually runs smoothly, however when levels become very large or detailed, it will be quicker to use the modeling viewports to manually select and move the Camera Entity. The Camera Entity is always visible in the viewports and is displayed as a green arrow. This can be selected and moved to any position in the viewport. When the camera is positioned into place, the view is then updated. This is the second method used for navigating the Camera Viewport. When a level becomes difficult to navigate in the conventional manner, then this

method should be used instead.

The Tool Bar

Below is a diagram which outlines the Tool Bar found in World Editor 2. The most commonly used buttons are outlined below.

- **Save, Cut Copy and Paste** - These are self explanatory but are only applied to already created brushes.
- **Lock Y, Y, Z Movement** - When any of these are selected, the selected brush will not be able to move along the locked axis.
- **Build BSP** - This refreshes the camera window with whatever new brushes may have been added to the level. This button should always be selected before performing any texturing of the level.

- **Snap** - This button toggles the Grid Snap on or off.
- **Compile** - This button opens the Compile Level dialog box and is explained further thought out the following chapters. When a level is compiled, a BSP is made from the 3DT.
- **Move** - When this button is selected, the selected brush can be moved. When the right mouse button is held down as opposed to the left, the brush will rotate. The keyboard shortcut is CTRL-R.
- **Scale** - This option allows the brush to be scaled in two axis. World Editor 2 can only scale one brush at a time. The keyboard shortcut is CTRL-W.
- **Shear** - An example of shearing a brush is outlined in the next chapter. The Shear shortcut is CTRL-Q.
- **Brush, Face and Entity Properties Dialog** - These open the properties box for either a brush, a brush face or an entity. Below are screenshots of each dialog box. The use of each is outlined in the next chapters.

These features are also available through the menu, along with many others.

The Tab Bar

The Tab Bar contains all of the interface features that used to build the level. The headings that will be examined in this chapter include, Template, Texture and Sky.

- **Template** - The Template tab is where most of the world building will occur from. It contains six buttons which create templates for differently shaped brushes. When one of these buttons are selected, a dialog box appears which asks for parameters that define the dimensions of the brush. When OK is selected in the dialog box, a template of the brush will appear in the level. This brush has not been created yet and the level editor is now in Template Mode. To create the brush and enter brush mode, hit INSERT. (In RfEdit and RfEditPro, the key which inserts a brush is ENTER.) This will create the brush, with the current texture selected in the Texture Tab, applied to the faces. The Template Tab also has the interface which adds Entities to the level. This is the first pull down menu and contains a list of every entity that is available in RF. To add an Entity to a level, select the entity from the menu and then select the button beside the list. A small blue x will appear at position 0,0,0. This is the template for the entity. The entity is added to the level by hitting INSERT.
- **Sky Tab** - All levels can be surrounded by a Sky Box. This is a combination of images which surrounds a level in order to display a scenic background. Each check box

activates the specific sky face in the level and the pulldown menu defines the image. It is a list of all textures in the TXL library being used for the level. In the next chapter, the Sky Box will be used to display images of outer space. The player will be given the impression that they are aboard a space ship.

- **Textures Tab** - The Textures tab is displayed in the side bar. It is a list of all the textures within the TXL library. Before a brush is added to a level, the texture which will be applied to it, should be selected from the Textures tab. Textures can also be added to the TXL library as needed, by using the Add button. This opens a file dialog box set to search for bitmaps. The Apply button, is used to apply a texture to a selected face of a brush.

This chapter served as a brief introduction to the World Editor 2 tool that comes with RF. In the next chapter, this tool will serve as an example to create a demo game named Space Escape.

Building a Level - Space Escape

Building a level with Reality Factory is a fast and efficient task. Small levels are easily built with little complication and large levels can be handled safely. This chapter describes the first stage to building a level. The stage involves adding enough brushes, to create a basic skeleton for the level. These brushes include the outer Sky Box, the interior floor, the ceiling and the main interior walls.

World Editor Basics

No matter which world editor is chosen to build the level, there are common points which must be considered, in order to efficiently create worlds. These are considerations that will help the designer prevent errors that may prevent the level from compiling.

- **Snap** - The most critical feature for any world building program is the ability to snap brushes into alignment with the grid. Without this feature, a level would be filled with numerous points where the brushes do not touch each other face to face. This can cause unwanted light leaks, extra faces to be drawn or worse level leaks. Before any brush is added to the world or manipulated in any way, be sure to know what the snap setting is and that it is toggled on. The default snap setting is 8 texels. Always start a

level with this setting.

- **Selected Brushes** - There are many ways to select a brush or entity in World Editor 2 and this makes it easy to accidently select more then one brush or entity. This can lead to undesirable results when manipulating the selected brushes. Before moving a brush, be sure that no other brushes are selected.
- **Copy and Paste** - World Editor 2 has the ability to copy multiple brushes and entities. This is very useful and allows for a quick and means of building the main structure of a level. However, it is very important that the Paste action never be performed with the keyboard using CNTRL-V, or it can easily create multiple brushes in the same place. Always use the menu or tab bar, to paste a brush into the level.
- **Wireframe Mode** - When building the level, be sure that the texture window is in Wireframe Mode. Only use Texture Mode when texturing the level. The camera mode is set under the Camera menu.
- **Moving Multiple Brushes** - If multiple brushes are selected, then snap can effect there placement when repositioning them. To safely move multiple brushes, turn

off Snap and use the *Move To* feature found under the *Modify* menu. A dialog box that appears will allow the brush to be moved using an exact or relative value. Be sure to turn on Snap when done.

- **Selecting Brush Faces** - Many people say that they have difficulty selecting brush faces in the camera window, using World Editor 2. Brush faces can be selected by first selecting the brush in one of the modeling windows and then from the *Edit* menu choose, *Select - Faces - All in Selected Brushes*. The faces can then be scrolled through using CNTRL-4 and CNTRL-3. If a face must be selected in the camera window, then first be sure to select, *Camera - Refresh*. This will update the camera window with any newly created brushes.

Starting the World Editor

For this example, a level for the demo game Space Escape will be created. This game can be downloaded from the Bookshock or Reality Factory web site. It is a very simple level that is an example for readers of this book. Take a look at the level to get a visual idea of what it should look like. In order to recreate the level, the Media Pack for this book must be downloaded.

Stage two of developing a game involves obtaining of the basic media. From the Media Package, extract the escape.txl file and place it in the RF media\levels directory. This takes care of stage two. Now start World Editor 2 and first save the file under the name escape.3dt. This should always be the first step. Next, before adding any brushes, select *Properties*, from the *File* menu. Under the Texture

Library heading, select the Browse button and choose the escape.txl file in the media\levels folder. Select, File - Save. The level is now ready to start having brushes added to it.

Adding Brushes

Brushes are added to a level using the Templates Tab. There are six basic brush shapes that can be added to the level by the World Editor. They are, Cube, Cylinder, Cone, Sphere, Stairs and Arch. These brushes can be made solid or hollow. To start the Space Escape demo level, a large hollow cube will be used to define the borders. This hollow cube will act as a *Sky Box* that displays a star pattern. To add the hollow box, select the cube shaped button under the Template Tab. A dialog box will appear asking for the dimensions of the cube. Match the settings to those shown in the screen shot below.

These settings create a hollow cube with walls that are 16 texels, thick resulting in an interior length of 1296 texels, a width that fits 512 texels and a height that can contain 144 texels. By adding an 8 texel high floor and ceiling, the interior height comes to 128 texels. This will be the height of the walls

Figure - 1

This is a top view of the level for the Space Escape Demo. The squares are a 128 x 128 section of the Grid. The gray area represents the floor and ceiling, while the light grey area is the surrounding outer brush. The thick black lines are the 8 texel wide walls that make up the structure of the level. All measurements shown are in texels.

in the level.

Adding the Sky

Before continuing with building the level, it would make sense to stop and flag the faces of the outer hollow brush as Sky faces. To do this, select the brush in one of the editor windows and then under the *Edit* menu got to, *Select - Faces - All in Selected Brushes*. The brush should go from a blue color to a purple color. The purple color represents a selected face.

Click the Open Face Properties button or from the *View* menu select, *Face Properties*. Click the check box marked Sky as shown below.

Now close the dialog box and select the Sky Tab from the right side of the World Editor. The Sky Tab has a check

box for six sides of a box. Check in all the boxes and then from the corresponding drop down menu, set the textures to match the following list. Be sure to also set the Rotation Speed to 0 and the Scale to 1.

- Left - space1
- Right - space3
- Top - space1
- Bottom - space1
- Front - space2
- Back - space1

These settings set up the images which are used to build the Sky Box. As the level progresses, other faces will also be flagged as sky, in order to simulate windows. Now that the outer hollow brush faces have been flagged as Sky and the Sky Box images defined, the floor and ceiling can be placed within the hollow cube.

Adding the Floor

The floor is made up of three brushes, that are each 8

texels high. Once again select the Template Tab and select the Cube Brush button. Set the values to match those in the screenshot and select OK. This creates a solid brush that is 512 long, 512 texels wide and 8 texels high.

This brush must be positioned to snap right against the bottom of the outer hollow brush. Next, two more 8 texel high brushes are added to the level. The first one has the X dimensions to be 256 and the Z dimensions to be 512. The second brush has the same dimensions but reversed. See Figure-1 for a reference to their positions. These three brushes make up the floors for the Storage and Air Lock Rooms, the Hallway and the Control Room.

These brushes must now be textured with the floor texture. Click on one of the three brushes in a modelling window and then while holding down the SHIFT key, click on the other two brushes. This is how multiple brushes are

selected individually. If the outer hollow brush is accidently selected, then while still holding down the SHIFT key, select the outer brush again. This will deselect it. With all three brushes selected, go to the *Edit* menu and choose, *Select - Faces - All in Selected Brushes*. The brushes should go from blue to purple. Select the Textures Tab in the World Editor and select the texture named FloorBase_02. Click the Apply button. This will apply this texture to all of these faces. Now click the open Face Properties button and set the texture scale settings to 0.5 for both the X and Y scale. See the screenshot on the previous page. This makes the image much crisper at close distances. Now to copy the floor, in order to make a ceiling.

If the three floor brushes are still not selected, then select them again using the SHIFT key. Under the *Edit* menu, select *Copy*. Now under the *Modify* menu select, *Move To*. In the resulting dialog box, select the Relative option. Then in the Y box, enter 128 and then add a - as a prefix, so that it ends up being -128. Select OK to move the selected brushes to the ceiling. To replace the floor brushes, select *Paste* from the *Edit* menu. The outer hollow box should now contain an 8 texel high floor and an 8 texel high ceiling. This then allows for a 128 texel high brush, to fit in between as a wall.

Adding Walls

The walls from the level will be made from a single 128 x 128 x 8 brush, that will be copied and scaled as needed. Examine Figure-1 for a reference to the position and size of the brushes. Start by selecting the texture named OutWall_11 and then click the Cube brush button from the

Template menu. Create a brush that has a X size of 8, a Z size of 128 and a Y size of 128. Position it on the map and scale it to be one of the first side walls. Make sure that it snaps right against the inner face of the hollow cube.

Under the *Edit* menu select, *Select - Faces - All in Selected Brushes* and open the Face Properties dialog box.

This step will align the texture to properly fit on the brush. Examine the screenshot of the Face properties box and the wall. Notice that the check box marked Texture Lock is selected and that the Y Offset is set to 64. Texture Lock is applied so that the brush can be moved without having to do too much re-texturing. The wall should look like the face shown in the screenshot.

Compare the texture image to the face of the brush. it seems that the face has been applied to the brush upside down. To correct this, select the button marked Flip, beside the Y offset setting in the Face Properties dialog. The texture will then be applied right side up. Now take a look at the 8 texel wide sides of the brush. These textures are still upside down. To select these faces, Select all the faces in the brush and the use CTRL-4 and CTRL-3 to scroll through each face, until the side of the brush is highlighted in purple. For each of the two 8 texel sides of the brush, hit the Flip button again. Now, all of the faces should have the textures properly applied.

These walls are then copied and pasted to create duplicates, which are then positioned and scaled. Examine Figure-1 closely to see exactly where the brushes go and the correct size. Be sure to use the grid markings as a reference and remember to not use the keyboard, for any pasting of a copied brush. The only wall that is not 8 texels thick is the center wall that divides the Store Room and the Air Lock. This wall is scaled to be 16 texels thick. Before adding any brushes, be sure that the Snap Settings are on and set to 8 texels. Add the walls, that make up the Store Room and the Air Lock. The next step will be to build the Hallway area.

Adding Sheet Brushes

The Hallway section of the ship will be made up of 6 brushes. Four will be corner pillars, that are simply 64 x 64 x 128 brushes, textured with the OutWall_09 texture. These are positioned in each corner as shown. The two walls which make up the side of the Hallway though, are Sheet brushes. A Sheet brush, is a brush that will only render one face and make it visible from both sides. Sheet brushes are well suited to be textured with a transparent texture, like the one in this example. There are a few pointers to remember however when adding a Sheet brush.

- When a sheet brush is created, the face that is always made the sheet is the brushes top face. This face is highlighted in yellow.
- The only face that can be textured, is the sheet face that is highlighted in yellow. Never select multiple faces of a sheet brush and never edit any other face in any way. This can cause rendering glitches, which can display more then one face.
- Keep sheet brushes at least 4 texels thick, unless they are going to be used as a surface decal. If a sheet brush is going to be used as a wall, then it is a good idea to keep it at least 8 texels thick.

Add the four corner pillars, by creating a single brush that is 64 x 64 x 128 and then texture lock all faces with an adjusted wall texture. Copy and paste three more of these brushes into the remaining corners. To add the first Sheet brush, open the Cube Template dialog and match the settings shown in the following screenshot. Note that the check box

marked Sheet, is checked. Select OK and then from the Texture Tab, choose the texture named OutWall_11. This is a 32 bit TGA texture with an alpha map. Hit the Insert key to add the brush from the newly created template.

The brush will appear in the world with the Sheet face pointing upwards. It must be rotated 90 degrees, to make the Sheet face resemble a wall. Select the Sheet brush in the Front XY modelling window (bottom viewport) and then choose *Rotate* from the *Modify* menu. In the resulting dialog box select, Right and 90 degrees. Click OK to rotate the brush 90 degrees in a clockwise direction. Position the sheet brush on the right side of the Hallway as shown in Figure-2. Before copying the brush to create the right side of the Hallway, select all of the faces in the brush and then use CNTRL-4 to scroll to through until only the Sheet face is selected. Click the open Face Properties button. The X and Y offsets will both have to be set to 64. The face of the wall should look like the screen shot on the following page. To create the other wall, copy the brush and then rotate it 180 degrees in the Top XZ modelling window. Position the new brush on the right side of the hallway and the select *Paste* from the *Edit* menu, to restore the brush for the left side.

Figure -2

Adding a PlayerStart Entity

At this point, the level can be compiled and previewed. Before doing so however, the level needs two entities added to it. From the Template Tab, use the first pulldown menu, to scroll down to find PlayerStart. See the side bar screenshot. Click the Add Entity button, located beside the pulldown menu. A blue entity template will appear at position 0,0,0. Move the entity to the bottom left corner of the level and position it right against the floor. Now do the same with a PlayerSetup Entity. This entity is virtual, so it can be positioned anywhere inside the hollow cube where there is room. These two entities are required to start a level. Without these entities, the level will not compile.

Compiling the Level

Although there are still many brushes to add to the basic structure of the level, the BSP can be compiled and

previewed. Under the *Tools* menu, select *Compile*. Match the settings in the resulting dialog box to those in the screenshot. Be sure to check the Preview option at the bottom. When the level compiles, which can take anywhere from a few seconds to a few minutes, the user is prompted to preview the level. Selecting Yes, will load and run the level. Examine the level so far, specifically the Hallway. The Sky Box should be visible through the side panels.

Building the Control Room

Take a look at Figure-2 again, this time looking at the Control Room. Five brushes have been added to make up the interior. Two side walls, two corner brushes and a front brush

which is a sheet brush. Before examining how the corner brushes were made, add two walls to the sides of the control room. Copy and paste them anywhere from the level and then scale them to fit. Be sure that Snap is on and that the window is zoomed in enough to see the Snap Grid lines.

The two brushes in either corner are angled inwards. Theses brushes were made by creating a triangular square brush and then Shearing it into shape. Below is a step by step examination of how to create the two corner brushes.

Top View	Create a Cube brush with the following dimensions. • Top X - 128 • Bottom X - 128 • Top Z - 32 • Bottom Z - 64 • Y - 120
Top View	Select the newly created brush in the Front XY window and rotate it 90 degrees to the right.
	Using the Top XZ viewport, position the brush in the left hand corner of the Control Room.
	Select the brush and then from the *Modify* menu select, *Shear*. Position the cursor to the right side of the brush and then shear it upwards as shown in the illustration. Be careful not to overshoot the Snap or the brush may shear a bit too much. If this happens, just shear it back to where it should go.

With the first corner brush created it can then be copied and moved into the right hand corner. The brush is then rotated 180 degrees in the Front XY window and the old brush is then pasted back into place. What remains is 256 texels, between the two corner brushes. This space will be filled with a Sheet Brush, similar to those in the Hallway. Copy one of the Sheet brushes in the Hallway and move it to the front face of the Control Room. Be sure that the yellow side of the brush (the Sheet Face) is facing inwards, away from the outer hollow cube. Scale the brush to fit between the 256 texel area. Do not forget to paste the old Hallway Sheet brush back into place.

The final step, is to alter the texture scale of the Sheet brush in the Control Room. Select all the faces in the sheet brush and then scroll through to the Sheet Face or select the sheet face in the Camera Viewport. Open the Face Properties dialog and then set the texture X Scale to 2. This will stretch the texture to fit the 256 texel brush and make it look like one window instead of two. Adjust the X Offset, so that the texture is properly aligned. Below is a screenshot, of what the sheet

face should look like.

Compile the level and examine the structure. The first part of the basic level structure is now complete. In the next chapter, colored lighting will be added to the level, in order to experiment with the RF light entity.

Lighting the Level

Lighting a small level such as this one is a very quick and simple task. Before going over adding lights to the Space Escape demo level, a brief discussion of using lights with RF is in order. There are many types of lights that can be used in RF, but the only one that should be used for normal lighting is the is the Light Entity. This is the simplest and most basic light entity. It is designed to be a static entity, that does not change during run time. Other light entities such as the Dynamic Light entity are much more versatile, but should be used very sparingly, because they can quickly use up system resources and slow the game down.

Level Lighting

There are three basic factors which define the lighting of a level. These are the Light Scale of the level, the Default Light Level and the intensity of the light entities. Below is a breakdown of each factor and how they relate to each other.

- **Level Light Scale** - The Light Scale of a level is defined under the Properties dialog which is accessed under the File menu. The default light scale setting for a level is 2. A smaller setting creates tighter shadows, but will decrease frame rate. This is because the light scale of a level, has a direct impact upon the number of polygons that are

drawn. A higher light scale setting will help to increase the frame rate, but it will also reduce the sharpness of the level shadows.

- **Default Light Level** - The Default Light Level is defined when the level is compiled, in the Compile Dialog box. This is the amount of light which will be applied to each face in the level. The default setting is 128, which allows for the level to be viewed without any lighting. When lights are applied to a level, the default setting should be brought down to 32 or 64, depending on the level atmosphere. It isn't wise to bring the light level down to 0, because the player will not be able to use the screen brightness, to adjust the lighting to a level of their liking.
- **Light Intensity** - The amount of light that is created by a light entity, has to be large enough to cover the area that it is meant to light. However, if the light is too intense, then light leaks will be created. This effect shows up. as light coming through doors and walls. In order to prevent light leaks, the intensity of the lighting should not be too much greater then the area it is meant to cover.

Lighting Space Escape

The diagram on the following page shows the Space Escape level with the lights added to it. There are a total of 11 light entities in the level represented by blue x markers. The squares above them, represent sheet brushes which will be added to the level first.

Start by selecting the texture named LiteBase_01 and then create a Sheet Brush with the following dimensions, 32 X by 32 Z by 4 Y. Insert it into the level and then select it.

Figure - 3

RGB: 167 37 18
Intensity: 256
RGB: 244 244 0
Intensity: 256
RGB: 0 217 0
Intensity: 190
RGB: 0 0 255
Intensity: 256

Open the Brush Properties dialog box and select the check box marked, Detail. This flags the brushes as a Detail Brush and helps to make the level compile and run much more efficiently. Any brush in a level that is not able to block the players view, should be flagged as a Detail Brush.

With the brush flagged as detail, position it about eight texels form the ceiling and then select *Snap Settings* from the *Edit* menu. Set the snap grid setting to 1 texel and select OK. Use CNTRL + Right Mouse Button + Mouse Forward, to zoom in on the Front XY viewport, until the 1 texel snap grid is visible. Position the first brush 1 texel away from the ceiling. See the diagram in the side bar. With the first brush in place, set the Snap grid back to 8 texels and then copy and paste 10 more brushes into position, using the Figure-3 as a reference. The next step, will add the light entities to the level. These will be positioned 8 texels away from the ceiling.

Adding the light entities to the level, will take as little as 5 minutes. To add the first entity, use the pulldown list of entities from the Template Tab, to scroll down to the entry named light. This is the basic light entity. Select the add entity button beside the pulldown list and then hit INSERT to add the entity. Position the light entity 8 texels from the ceiling as shown in the side bar. Each light entity will be placed directly under the center of each of the square detail brushes. Position the first light under any one of the sheet brushes and then copy and paste 10 more under the rest. Be sure that the Snap is on and set to 8 texels while doing this. The next step, is to change the color and intensity of each light entity.

Figure-3 lists the RGB color settings and intensity for each group of lights in the level. To modify the light entity

properties, select the light entity and then open the Entity Properties dialog box. The properties of the selected light entity, will appear in the dialog box. Double click on the color entry and a Select Color dialog box will appear. Manually enter the values into the R G B settings of the dialog and select OK.
Then double click the intensity entry of the properties and manually change the *light* property. Do this for every light in the level, using Figure-3 as a reference to the entity properties.

At this point all of the lights should be added and the properties set. The level can now be compiled and examined. Select *Compile* from the *Tools* menu and in the Default Light Level entry enter, 32 32 32. Be sure that there are no hidden characters and that the value is only 32 32 32. Click OK to compile and preview the level. The Store Room and Air Lock should have a blue lighting, the passage to the hall should have green lighting, the Hallway should have yellow lighting and the control room should have red colored light.

Adding lighting to a level is as easy as that. Add the first light and then copy and paste the rest into position. Fast and efficient. With the basic lighting added to the level, the next step will be to add doors. This is the focus of the next chapter.

Adding Doors and Models

World levels are mostly made up of static brushes and entities. However, it is possible to create interactive brushes and brushes that move. These brushes are referred to as World Models. This chapter examines how to create world models and use them as doors, area triggers and elevators.

Any brush or group of brushes can be made a model. When multiple brushes are made a model, they then act like a single brush. This is termed a world model brush. Any brush can be made a model. To flag a brush as a model all that is required is to select the Model Tab and then, with a brush selected, click the Model Button. A dialog appears asking for the name of the model. The brush is now a world model and can have a number of different entities control it.

A model does nothing without an entity to control it. The most common entities used with a model includes the Door Entity, the Trigger Entity, the Moving Platform Entity and the DestroyableModel Entity. By associating one of these entities to a world model, the player will be able to interact with it in a manner defined by the function of the entity.

Before going through the steps required to add the doors to the Space Escape demo, start a new level and save it under the name modeltest.3dt. Select *Properties* from the

File menu and set the TXL library to be, escape.txl. Create a hollow cube that is 544 x 544 x 544. With 16 texel thick walls, the interior becomes 512 x 512 x 512. Add a PlayerStart and a PlayerSetup entity and use this as the test level.

Adding a Door

The first example of using a model will be a door. This is the most common use of models for indoor levels. Before going on to the example, it is important to how a door is animated. Below is an outline of the door model keyframe positions, that makes up the animation. The X is the pivot point of the model and can be defined as required. For a door that swings open, it is important to set the pivot point to

K : 0.00	This is the start fame for the animation. In this case the door is closed and waiting for contact to open it.
K : 1.00	The next keyframe still has the door closed for 1 second after it has been activated. This is to allow for the player to get out of the way, when the door opens towards their direction.
K : 2.00	The keyframe at 2 seconds, is the door in the open position. This means that the door will take 1 second to open all the way.
K : 5.00	The door then needs to stay open for a given time so that the player can walk through it. In this case, the door is kept in the open position until the 5 second keyframe. This means the door stays open for 3 seconds.
K : 7.00	The last action is to gently close the door so as to return it to the start position. This completes the full animation sequence for the door.

represent the position of the hinges. To create the door model, start by adding a door shaped brush to the level. The dimensions can be 64 X - 8 Z - 128 Y. Select the brush and position it against the floor.

With the brush still selected, select Model from the Models Tab. A dialog box will ask for the name of the model. Name the model, door1 and select OK. Before adding the animation keyframes, select the Set Origin button. A light blue X will appear at the center of the model. This marks the pivot point and should be positioned in the bottom right corner of the brush, as shown in the previous diagram. When Set Origin is selected, the button changes to Done. Select this button when the pivot point is in position. Now the keyframes can be added to the model, in order to animate it. Below are the steps required to animate the keyframes.

- **Key 1.00** - Start by selecting the button marked Insert, at the bottom of the Models Tab. The button will change to Done. Do not move the brush and just select the Done button. In the dialog that opens enter 1 and select OK. This creates the first keyframe of the animation.
- **Key 2.00** - Be sue that keyframe 1.00 is selected and then click the Insert button again. This time, select the Move button from the Tool bar and right click the Mouse to rotate the door into the open position. It will rotate on the axis set by the pivot point. When the brush is in position, select Done and enter 2 as the time.
- **Key 5.00** - Select Insert and then Done without moving the brush and then add 5 as the key time.
- **Key 7.00** - For the final key frame, select Insert and then rotate the brush back to the original position. Click Done

What to Read

From the RF docs, be sure to read the following articles.

- Door
- MovingPlatform
- Trigger
- DestroyableModel
- LogicGate
- Message

and add a key time value of 7.

With the model animated, all that is required to interact with it is to associate it with a Door Entity. Use the pulldown menu under the Template Tab to scroll down to Door and then add the Door Entity to the level. It can be placed anywhere, but it is good practice to put it near the model it is associated with. Select the added Door Entity and click the Entity Properties button. Modify the following entries for the Door Entity.

- **Model** - door1
- **bReverse** - true
- **PlayerOnly** - true

These settings associate the entity with a model named door1 and only allows the player to activate it. The bReverse entry makes the door reverse the animation, if it hits the player while closing. Close the Entity Properties dialog and compile the level. When the player makes contact with the brush, the model will animate. Adding a door to a level is as simple as that. Next the same method will be used to create an elevator platform.

Moving Platforms

Add to the test level another brush with dimensions 128 X - 128 Z - 4 Y and position it against the floor. Select the brush and make it a model. Name the model, platform1. This model will have 4 keyframes added to it similar to the way the door was animated. Key time 1.00 has the brush still against the floor. This is the delay, which allows the player to step on

and off the platform. The next keyframe has the brush set to the elevated position. To position the brush, lock the X and Z axis and then using the Front XY viewport, move the brush 128 texels upwards. This is held in place for 5 seconds and then the brush is returned to its starting position. See the side bar for an example.

The entity used to activate the models animation, will be the MovingPlatform entity. Use the template tab to add a MovingPlatform Entity to the level and set the properties to the following.

- **Model** - door1
- **PlayerOnly** - true

Compile and preview the level. When the player steps on the platform, it should pause for 1 second and then lift the player. The platform pauses at the top and then returns the player to the ground. The player must step off and then on the platform, to activate it again. This is the most basic use of the Door and MovingPlatform Entities.

Models as Triggers

The Trigger Entity is one of the most versatile entities found in RF. It acts as a virtual switch that can be controlled by a model and be used to activate other events in the level. The next example uses a model associated with a Trigger Entity, to act as a Lock Release for the door model.

Start by adding a new brush to the level that is a 32 x 32 x 32 cube and position it 8 texels above the floor. Make it a

model and name it, trig1. Add only a single keyframe, that positions the brush down through the floor of the level and set the key time to 2. This creates a model that simply falls through the floor when animated.

Use the Template Tab to add a Trigger Entity to the level and modify the following properties.

- **Model** - trig1
- **szEntityName** - doortrig1
- **PlayerOnly** - true
- **TimeOn** - 60

These properties associate the trigger with the trig1 model and then names the trigger doortrig1. This name can be used by other entities in the level, to associate themselves to this one. The TimeOn property, sets the trigger to be on for sixty seconds. The player has this much time to reach the door. With the Trigger Entity properties set, select the first Door Entity that was added previously and change the following property.

- **TriggerName** - doortrig1

This associates the Door Entity with the Trigger Entity, in a manner that makes the door only open, if the Trigger has first been activated. Compile and test the level. Walk towards the door and try to open it. The door should not open. Walk over to the cube model and make contact with it. The cube model will fall through the floor and the trigger will be activated. Go back to the door and try to open it again. The door is now unlocked and will open.

Callback Messages

Practically, a locked door should return some type of message that informs the player that the door is locked. All that is required to do this is to add two more entities to the level, a Logic Gate and a Message Entity. Use the template tab to add a Logic Gate and a Message Entity to the level and position them near the door model. Before setting the properties of these two entities, edit the szEntityName property of the door entity as follows.

- **szEntityName** - door1

This names the Door Entity door1, so it can act as a type of trigger for other entities. in this case, it will be a callback trigger for the Logic Gate. A Callback is the term given to a momentary trigger action ,that is only detectable by a Logic Gate type 6. This is the callback setting for the Logic Gate Entity. When the entity detects a callback, it will act like a stable trigger that can activate other entities. In this case, the Logic Gate will trigger the Message entity. Below is a diagram which illustrates the association between the entities.

Select the Message Entity and modify the properties to have the following settings.

- **DisplayType** - Static
- **TextName** - LockedDoor
- **TriggerName** - gate1

The property DisplayType refers to a heading which is defined in the message.ini file. Use NotePad to open the message.ini file (not message.txt) and be sure that there is a heading titled Static. The corresponding data defines the time that the message is displayed and the position on the screen. The TextName property refers to a heading named LockedDoor, found in the message.txt file. Edit the message.txt file (not message.ini) and add the following entry.

[LockedDoor]
The Door is Locked

The message entity uses the TextName property to refer to this entry and display the message, *The Door is Locked*. When the trigger is activated and the door is unlocked, the message will not be displayed. Compile and preview the level. This time, when the player tries to open the door, the message will appear in the top center of the screen.

Double Doors

Adding double doors is easily done, using the NextToTrigger property of the Door Entity. To create a set of double doors, create two doors and animate them to act like

double doors opening and closing. Add a Door Entity to each door model. To make them act like double doors, set the NextToTrigger property of one door, to match the model of the opposite door. Below are the property settings for two Door Entities, that work together to make two door models act like double doors.

- **Model** - **door1**
- **bReverse** - true
- **PlayerOnly** - true
- **NextToTrigger** - **door2**

- **Model** - **door2**
- **bReverse** - true
- **PlayerOnly** - true
- **NextToTrigger** - **door1**

One Way Doors

The same method that was used to create a locked door, can be used to create a one way door. A one way door is activated by another trigger, that is only found on one side of the door. To create a one way door, change the bOneShot property of the Trigger Entity to false and delete the second keyframe of the cube model. Position the cube model in front of the door, about 32 texels away. Select all the faces in the cube model and open the Face Properties dialog. Select the check box marked, Transparent and set the value to 0. This creates an invisible brush. With the cube brush still selected, open the Brush Properties dialog and click the check box marked, Empty. The brush can now be used as an invisible area trigger that the player can walk through.

In order to make the door be activated by the trigger

and not the player, modify the following property in the Door Entity.

- **bNoCollide** - true

This prevents the player from opening the door and only allows it to be activated by a separate trigger. In this case, the trigger is the invisible cube brush. Compile and preview the level. This time, the door will open only when the player passes through the area where the invisible cube brush is positioned.

Adding Doors to Space Escape

2 Texels of space on either side of the door brush. This appears as one small grid space.

Two types of doors are found in the Space Escape demo. Single sliding doors and double sliding doors. These will be added to the level by making one door model and then cloning that model as required. Three single doors and three double doors will be added to the level, for a total of nine models.

Figure-4 is the layout of the level with the models and door entities added. Use this as a reference, when positioning the doors. To start, set the Snap Grid to 2, select the texture named door01 and create a brush with the dimensions 64 X - 4 Z - 128 - Y. Position the brush, as the door to the first room where the player starts. Zoom in until the snap grid is visible and precisely position the brush so that it is 2 texels (1 small grid space) away from the edge of each wall. See the sidebar illustration for an example. When the door has been positioned, it can then be animated to slide open and closed. See figure-4, for the position at each keyframe. Before

Figure-4

animating the door though, it would be prudent to align the textures. In regards to this, one should never apply the Texture Lock to a model that is animated. The model will move in the level, but the texture will remain in place. See the screenshots in the side bar for what the doors should look like. The Y Offsets will have to be set to 64 and the X Offsets will have to be adjusted as the door models are added to the level. Obviously, this is because each door will have a different X position and texture lock can not be applied to the original.

With the brush textured and selected, click the Model button from the Models tab. When prompted for the model name type door01. Animate the model as shown on Figure-4 and add a Door Entity to the level only changing the model property to door01. Compile the level and test out the door. It should slide open as soon as the player comes in contact with it and then close after 4 seconds.

Before cloning the model to create other doors, add another model door, but animate it to open in the opposite direction. Name this model door02 and add a Door Entity to control it. The rest of the doors can now be quickly cloned as needed. This is a quick and efficient process. Simply select the model to be cloned from the Models window and click the Clone button. A dialog box will ask for the name of the model. Continue to name them sequentially door03, door04, etc., cloning the appropriate model, for the required direction it needs to open towards. Hit OK and a clone of the model will appear in the level. Position these where required. This works because the new position of the clone, does not need any rotating. If it did, it would change the axis of the animation. When all the door models have been added, copy and paste

Door Entities for each. For the double doors, be sure to set the NextToTrigger property, to the door model that is used in conjunction and vice versa. When all of the doors have been added, compile and test the level. Make sure that all the doors in the level open smoothly.

At this point, stage 3 of the design process is complete. All the structure has been built and the level is ready to have pawns and weapons added to it. The next chapters will discuss adding weapons, creating pawns and writing pawn scripts. Everything that is needed, to get through stage 4 of the design process, which is the construction of the actor media. Then this media will be added to the level, to complete stage 5.

For the next steps in the design process, the World Editors will not always be needed. To make things a bit more efficient, run the RF INI Editor and set the Starting Level to, escape.bsp. This makes the Space Escape Demo, the default level when RF is executed. To make things easier, it will be a good idea to Drag & Drop a RF icon into the Windows Task Bar.

Creating Actor Files

3D models made using third party software, are used with Reality Factory in a number of ways. They can be imported as MAP or 3DT files and be used as actual game geometry, or they can be added as static entities that add to the scenery. Most commonly, 3D models are added to a game as either an Attribute, a Static Entity or a Pawn. In these instances, the model must be added to the game as an Actor file. This chapter will discuss how 3D models are converted into ACT files.

Before continuing with the chapter, it is important to take a look at the necessary, third party software that will be required for the conversions. All of these tools are not required. The software that will be required, depends on the method of conversion that is chosen. These methods will be discussed later in the chapter.

- **Anim8or** - This is a freeware 3D modelling tool that is incredibly easy to use. A complete amateur can sit down with the manual and learn how to make 3d meshes from simple geometries. It can export objects to 3DS format, but can not export animations. It is not at all a requirement to make RF actors, but is very easy to use and good for complete beginners to start learning about basic 3D modelling.

- **GMAX** - This is the scaled down and freeware version of 3D Studio Max, which is specifically designed for game builders. It is complex but extremely powerful. It is well worth taking the time to learn how to use it. In regards to making RF act files, it is one of the few programs which can be used entirely on it's own, without requiring any other software, except the plug-in scripts.
- **Milkshape3D** - Although Milkshape 3D is not freeware, ($20) it is also can create models and animations for use with RF, without needing any other third party tools. Milkshape 3D can make models, animations and export them to a format that is ready for Actor Studio. Using Milkshape is one of the most reliable, fastest and versatile means of building models for RF. Milkshape3D is also very easy to use and there are numerous tutorials dedicated to using it.
- **LithUnwrap** - Now commercially sold under the name Ultimate Unwrap, this software is still available as freeware. It is a very powerful tool that should definitely be in any modelers toolbox. It can be used for conversions of formats, skinning models and polygon optimization to name only the most common features. It is available at the RF site as well as a few other places.
- **TrueSpace 3.2** - Although Caligari Truespace is a commercial product, version 3.2 is available as a free download. It can be used to make 3D meshes which can then be used with Truegene to quickly create RF actors. Using this combination is fast and efficient means of creating RF actors.
- **TrueGene** - This tool is used to convert Truespace COB files into ACT files. It is available at a number of web pages including the main RF site.

- **3DS23DT** - This is a tool which converts 3DS models into 3DT files.
- **GMAX Scripts** - The latest NFO and KEY exporter scripts for GMAX is required to create animations and meshes using only GMAX or 3D Studio MAX.

Parts of an Actor File

The contents of an act file can be viewed using the RF tool, VFS Explorer. Under examination it seems that the file is a compilation of MOT files and a BDY file. The BDY or Body file, which can also be examined with VFS Explorer, is what contains the materials of the actor, the mesh and the skeleton of the act file. The MOT or Motion files, contain the skeleton animation data. RF Actor Studio takes these formats and combines them into an act file. It can also convert NFO and KEY files, which are made by MAX and GMAX. These are the equivalent to BDY and MOT files.

Conversion Methods

There are basically three methods to create ACT files for RF. They use a combination of the software mentioned above and the RF Actor Studio software. This next section will detail these three methods and examine the strengths and limitations of each.

Using Milkshape 3D

As mentioned previously, Milkshape is the single tool that can be used all by itself, requiring nothing else except the RF Actor Studio. The formats that are created with Milkshape

are the BDY format for meshes, and the MOT format for animations. This is handled through the Export options within Milkshape. These formats, are then imported into Actor Studio. Because Milkshape can not import 3D Studio files, it will be necessary to convert the 3DS mesh to the Milkshape MS3D format by using LithUnwrap. Using Milkshape is the only guarantied approach to getting an animated actor into RF.

The basic process is to model the mesh and then create the skeleton for it. The vertices of the mesh are then assigned to the specific joints of the skeleton. Every vertex must be assigned to a joint or the export process will fail. The mesh must then be assigned a material, which must be an 8 bit or 24 bit bitmap. Other wise, the texture will not be recognized by the Milkshape export tool and the material will not be added to the BDY file. Once the mesh has had it's bones assigned and the materials mapped to it, it can be converted to a BDY file by selecting *Export - Genesis3D BDY*. This will ask for a name to save the file as and then create the BDY file.

Animations should not all be added to the same model. The original MS3D file should be saved with a new name and a single animation is used for each new file. Otherwise, when the animation is exported, it will be an animation of all of them combined. Once the file has been animated, the animation is exported to MOT by selecting, *Export - Genesis3D MOT*. Be sure that the Milkshape Anim button, is not selected while exporting the MOT file.

Using Truespace 1.0 and Truegene

This is the only method that does not require the RF Actor Studio. It entails using Truegene to convert a Truespace

COB file into an ACT file. It is as simple as that. It works similar to the RF Actor Studio in the sense that a ‘body file’ must be selected and animation files are then added to it. The COB files are converted to BDY and MOT files and then an actor file is assembled from them. Truespace can also import 3DS files along with their animation data. This makes using this method, to be a viable means of converting commercially purchased models into ACT files for RF. COB and 3DS files, are two of the most commonly found formats, used for both freeware and commercial game models. Some models will require some ‘post processing’ to be exported properly, so it is important to test the models, before any commercial models that may be converted in this manner are purchased.

Using GMAX with NFO and KEY Exporters

This method is the only method that uses completely freeware tools. It also can produce the best results in many instances. The procedure requires GMAX and the NFO and KEY exporter scripts. It is important to read the instructions that come with the exporter scripts. The docs state, that the meshes must be set as editable using the Modifiers. This is an important point to keep in mind. The scripts are used, by running them with the GMAX Script Listener Window open. When the script finishes running, the generated text is copied from the window and then saved as either an NFO or KEY file. These formats are then used by ACTOR studio, to create an ACT file.

Once the proper modifiers have been applied to the model, the export process entails opening the Script Listener Window and executing the NFO and KEY export scripts. As each script is executed, the script listener window will be populated with the necessary data. This data is then copied to

a text file and saved with the extension NFO. Results of the KEY script, are saved with the extension KEY. In many instances, the file will be larger then what is capable of being handled by NotePad, so it is best to copy the contents of the Script Listener window into WordPad. One thing to keep in mind is that the Script Listener window, can not copy its entire contents for a single copy and paste. It must be copied in 'chunks' and added to WordPad piece by piece. Once again, once the data has been copied over, the file is saved with the NFO or KEY extension, as determined by which script was executed.

Using the RF Actor Studio

RF Actor Studio can create animated or static actors, from BDY and MOT formats or NFO and KEY formats. There are a few requirements that must be considered before using Actor Studio. First off, the textures used by the model must have dimensions that are a power of 2, and must be no larger then 256 x 256 pixels. Secondly, the names of the animations that will be used by RF, are the names of the actual MOT files used in the actors, and not the names given to them when compiled. This will be looked at more carefully during this example. Below is a step by step example of using the RF Actor Studio.

Step 1 - Prepare the BDY and MOT file using Milkshape3D or prepare the NFO and KEY files using GMAX. Be sure that the texture dimensions are no bigger then 256 x 256 and are a power of 2.

Step 2 - Place of the media into its own working directory. This will let you organize your models and make changes to them easily. For this example, make a new folder and name it Model_One. Place the both the BDY / MOT or NFO / KEY files and the textures, into this folder.

Step 3 - Run Actor Studio and select *Project - New*. Browse to the folder that contains the media and save the new project under the name model1.apj. Check the box marked Use Working Directory and select OK. By default, the resulting ACT file will be named the same as the project. The ACT file name can be changed under the Tab labeled Target.

Step 4 - Click the Tab labeled Body. At this point the program will require the path to the BDY or NFO file which will be used. Select which format which is being used and browse for the file. The option to use a MAX file will not be examined here. What it does is launch 3D Studio MAX and then automatically run the NFO and KEY exporters.

Step 5 - Select the Motions Tab. This is where the animations will be added to the ACT file. This portion of the project will require the KEY or MOT files that makes up the animation data. To add an animation, follow these steps:

- Select the format you will be adding in the format frame.
- Select the Add button.
- Enter the name of the animation in the box prompted and hit OK. The name that is entered here is not used by RF. It instead uses the name of the actual MOT or KEY file, without the extension.
- To add the file, first select the newly added entry, then

select the button labeled Browse, to find the animation.

- Repeat these steps for each animation file.

Step 6 - Now that the body and animation files have been specified, the actor can be compiled. Select the Build button in the bottom right corner. This will compile the actor. If all goes well, Actor Studio will return with the response saying, 'Build Successful.' Other wise it will return, 'Build Failed.'

Troubleshooting the Build

If the build failed then double check the following:

- All media is in the working directory.
- Use Working Directory is checked.
- The textures are not bigger then 256 x 256 and are a power of 2. ie.. 32 x 32, 128 x 128.
- Be sure that he NFO and KEY files are not corrupt or incorrectly made.
- The materials were in the same directory as the BDY file when exported from Milkshape.

Viewing the Actor

To view the completed actor is a pretty straight forward process. Just run Actor Viewer, select your video driver (usually D3D) and then open the ACT file. The animations can be selected from the animation list. You can also view the animations running at different speeds. If an actor fails to load in the RF Actor Viewer then it will most likely fail to load in RF.

Bringing Actors into RF

The last chapter examined the different ways that an actor file can be made for RF. This chapter will examine the different entities that use actor files. Be sure to place your actor files in the media/actors directory. Actor files are used for Pawns, Static Entities and Attribute entities. Each will be examined more closely with an example.

Adding Actor Entities

When one of the above mentioned entities are to be placed in the world using one of the World Editors, it's position will be the bottom point of the actor. In order for the actor to not have to 'fall into place', the entity should be position exactly at ground level to where it will stand. However, it is very important to not have the entity within a brush, because it may cause strange effects to take place with the brush and the entity.

StaticEntityProxy

The StaticEntityProxy is used to bring an actor file into the BSP and animate it or interact with it if necessary. The entity is capable of triggering other entities and sounds, as well as being able to cause damage to the player and at the

same time be damaged by the player. A StaticEntityProxy can be pushed around by the player, animate as necessary and also remove itself when destroyed. It is one of the most versatile entities RF has to offer and also one of the easiest to add to a level. The only required entry is szActorFile. This entry requires the file name of the actor file in the media\actors directory. For example, if an actor file named plant.act is to be used, then the szActorFile entry would be, plant.act. Once again, this file must be located in the media\actors directory. If a directory named plants was added to the media\actors directory, then the szActorFile entry would need to be, plants\plant1.act. This will look for a file named plant1.act, in the media\actors\plants directory.

The next thing to consider is the bounding box size of the actor. This is handled using the BoxSize entry, in the entity properties. The box size is the 'real' area of interaction with the player. The player can not enter this area and will trigger any specified actions when contacting it. If this is not set, then the bounding box will be as large as the actor.

Finally there is the ScaleFactor entry. This is also important to consider, because it scales the actor outwards from the center. This is obvious enough, but it is important to remember that the placement in the world editor, must change in accordance to the scale difference. The scale and bounding box, are the two variables which must be adjusted for the actor to properly interact within the world. Bounding box dimensions can be viewed in game, by selecting the DEBUG option from the RF demo menu and then checking the Show Entity Bounding Box option. This will display the bounding box of all entity actors in the game.

Attribute Entity

The Attribute Entity has a very specific purpose in RF and that is to allow for a model that when contacted by the player, will increase an attribute value. For example, a model of a medical box that increases the players health. The Attribute entity is very versatile. It can re-spawn itself after a specified time and can have its presence in the level, controlled by Logic Gate or Trigger entity. It can act like a trigger and even create a Callback when it is contacted with. This feature will be used in the example here, to display a message when the attribute is picked up. Animation is not a feature of this entity as of version 070, but it may be added by RF developers in a future version.

An Attribute entity can be placed into a world by simply specifying the szActorFile entry, but in order for the player to interact with it, the player.ini file will require the appropriate attribute definition. Also, the attribute amount can not be already at its maximum or the player will not be able to 'pick it up'. For this example, the attribute used will be health. Edit the player.ini file in NotePad and locate the entry labeled [health]. Set the initial amount to be 50. It should look like this.

[health]
initial = 50
low = 0
high = 100

Save the file and exit NotePad. This will ensure that the player will be able to pick up the attribute for this example.

Next is a step by step example of adding an attribute to a level and then displaying a message when it is picked up.

Step 1 - Start up a new level in RfEdit or World Editor 2. Create a hollow cube to be the world boundary, then add a PlayerStart entity and PlayerSetup entity. Select an actor file to be used as the attribute. By default the szActor file uses skull.act. This file did not come with the RF07 media package. If you do not have an actor file to use, then just choose the Ernie player actor.

Step 2 - Add an Attribute entity to the level and place the name of the actor file in the szActorFile entry. In the AttributeName entry type, health. In the AttributeAmount entry, type 10. What this does is set the entity to add 10 health points to the player when it is collided with. Position the Attribute entity 8 texels above the ground in the level.

Step 3 - At this point, when the level is compiled, the player will be able to walk over the attribute and pick it up. It will disappear and the players health will increase by 10 points. But it needs something more to spruce things up a bit. A message display, will be added to inform the player that they picked up the attribute. The Attribute entity uses a Callback, when the player collides with it. This Callback can be recognized by a LogicGate entity and used to trigger a message or other event. To the Attribute entity properties, add the name Attrib1 in the szEntityName entry. Now add a LogicGate entity and a Message entity to the level.

Step 4 - Edit the LogicGate properties to set the szEntity name to LG1 and the TriggerName entry to Attrib1. Set the

Type to be 6. This will define the logic gate as a callback trigger. Now edit the Message entity and set the TriggerName entry to LG1. Then define the DisplayType entry as Static and the Message entry as GotHealth. The value Static, refers to the type of message as defined in message.ini. It is important to check and see that this is correct. Prior to RF07, the message type was defined as Normal.

Step 5 - All that is left to do is define the message GotHealth in the message.txt file found within the install folder. Open the message.txt (not message.ini) file using a text editor and add the following:

[GotHealth]
Picked Up Health

Now compile and preview the level. If all went well, then when the player walks over the attribute, it will disappear and display the message, 'Picked Up Health.' Ten health points should also be added to the players health level. To make the attribute reappear, set its ReSpawn entry to true and then set the Delay entry to the number of seconds between spawning. This will re-spawn the attribute and allow the player to continually pick it up, until no more health can be added to the players health levels.

Adding Pawns

The Pawn entity is the most versatile entity found within RF. This is because it is controllable through the RF Scripting method. This is thoroughly explained, in the

chapters on pawn scripting. Pawns are the only entities that do not simply use an szActorFile entry, but instead require a pre-defined Pawn Type. These are defined within the pawn.ini file. Below is the pawn definition found in the default pawn.ini that comes with RF07. It defines the flying monster seen in the default demo. Take a quick look at the definition.

[Flock]
actorname = pawn\flock.act
actorrotation = 0 180 0
actorscale = 1
fillcolor = 255 255 255
ambientcolor = 255 255 255
subjecttogravity = false
boundingboxanimation = Idle

This defines a pawn type named Flock. Examining the entries, it seems that the actor file used is named flock.act and the scale is set to 1, the original size. This actor file is located in the media\actors\pawn directory. The pawn is not subject to gravity, so it will not fall within the level. The least important entry is boundingboxanimation. The bounding box size of the pawn, can be defined in the pawn script using the BoxWidth() command. This entry can be left out when a new pawn is defined.

With the pawn properly defined in the pawn.ini file, it is added to a level using the Pawn entity. Add a pawn entity to the simple level built from the attribute example and edit the following three properties.

- **ActorType** - This requires the name of the pawn definition

in the pawn.ini file. For this example, type Flock in this entry.

- **ScriptName** - This is the name of the script file which will be used by the pawn. The entry is case sensitive and requires the file extension as well. If the pawn entity does not find the script, then the pawn will appear in the level and play its default animation. For this example, type null.s within the entry. It will not find the script and only display the pawn in the level.
- **SpawnOrder** - When an actual script is used, the entity will look for the first Order to execute. This is described in detail in the scripting chapter. Common practice is to just use Spawn as this entry and name the first script order Spawn as well. This entry is also case sensitive.

After altering these entries, compile the level and preview it. The flock actor should appear where the entity was positioned. It should simply be playing its idle animation. This example only showed how to get the actual pawn type within the level. Very little can go wrong, but if the pawn does not show up in the level, then check the following.

- Can RF find the specified actor file? If RF can not then it will crash and log an error saying that it could not find the file, specifying the path where it looked for it.
- Is the PawnType entry correct? This entry is case sensitive and must be spelled exactly as it is in the pawn.ini file.
- Is there an entry in the SpawnTrigger field? If so, then remove it. The pawn will only spawn if the Trigger in this entry goes high.
- Is the pawn positioned inside a brush or at a height that is

not visible by the player?

More About Pawns

As it was just stated, pawns are the most versatile entity in RF. Below is a list of tasks that pawns can perform and be used for.

- Player Enemies - A shooting combat a.i. for a pawn is built right into the RF scripting system, making it very easy to add enemies that will attack the player.
- Player Helpers and Guides - Pawns can also act to help the player and even attack other enemy pawns.
- Conversation Screens - Pawns can generate conversations that offer logical choices to the player and the react accordingly.
- Moving Cameras - When combined with a Fixed Camera Entity, a pawn can be used to create in game movies and cut scenes.
- Attribute Modifiers - Pawns can modify player attributes and use this feature to create timers and score keeping systems.
- Event Manager - A pawn can be used to control events throughout an entire level and do so at specified times.

Adding First Person Weapons

The RF weapons manager is simple and very versatile, allowing almost any weapon model to be quickly and easily added to any RF game. An already modelled weapon can be brought into RF in less then 30 minutes. This chapter goes through the process of creating a First Person weapons model into RF. The model used in the example, is a commercial model of an MP5 submachine gun, but this method can be applied to any similar model. Tools required for this example include, Milkshape and Lithunwrap.

Converting the Model

The first step is to create a suitable BDY file, that can be used by RF Actor Studio, to create the act file. Note, that these same methods can be applied using GMAX / Max and the NFO exporters. The MP5 model was originally in 3DS format. It was opened with LithUnwrap and optimized using the Optimize Model option found under the Tools menu. It was then saved as an MS3D file. The file is then opened in Milkshape and the model is centered at 0,0,0. This is a very important step, which makes the final positioning process much more efficient. It ensures that the model starts at a position, that has no offset from the player position. Also be sure to scale the model to a size that is appropriate for the engine. Some commercial models are quite small and need to

be scaled up just a bit, in order to be properly worked on in Milkshape.

Next step is to add a root joint and a joint that will be the launch point for the projectile. Assign all of the models vertices to the root joint. Below is a screen shot of the MP5 model with the joints added to it.

Note that the root joint is as 0,0,0 and the two other joints are set at either end of the weapon. The second joint to the right, is protruding just past the edge of the barrel. Write down the name of this joint, as it will be referred to later on, when defining the projectile launch point for the weapon. Be sure that all vertices are assigned to the root joint and then export the model as a BDY file.

First Person Weapon Animations

As it was stated in a previous chapter, a first person view requires only six animations for each weapon. However, the same animation can be used for many actions, so it is only necessary to create a minimum of two animations per weapon. These animations are Arm and Idle. Here is the full list of animations, used by a first person weapon.

- **Idle** - For this example, the idle animation will be a single keyframe of the weapon position at 0,0,0. This same animation will be used for Shoot, Walk and Empty.
- **Arm** - The arming animation, is what is displayed when the player selects this weapon. It can be made from a minimum of two keyframes. The first keyframe having the weapon in a holster position and the second keyframe having the weapon in the Idle position. (0,0,0)
- **Shoot** - The same single keyframe animation that was used for the Idle animation, is used for Shoot, Walk and Empty. This has the effect of keeping the weapon still at all times. This is the simplest method for adding a weapon, but the effect can be enhanced by adding a muzzle flash, ejecting shells and the correct sfx.
- **Walk** - The same single keyframe animation used for Idle can be used for Walk.
- **Reload** - RF has two options for reloading the weapon. Two animations can be defined, one for when the weapon automatically reloads and another for when the player selects the reload button. In this example, the Arm animation is also used to create the Reload animations.
- **Empty** - Again, the Idle animation will be used for empty. A sound effect will let the player know that the weapon is out of ammo.

The MP5 model as an Act file viewed in the RF Actor Studio. When the game is complete, the weapon is re-modelled, removing any polygons that are not seen by the player during the course of the game. This is done to help maximize the rendering efficiency.

When exporting these animations, be sure to export an original MOT for each animation. For example, the single keyframe animation of the weapon at 0,0,0 is exported as Idle, Shoot, Walk and Empty, while the arming animation is exported as Arm and Reload.

With the BDY and MOT files exported from Milkshape,

the model can then be assembled using the RF Actor Studio. Below is a screenshot of the RF Actor Studio at the Motions Tab. Note the small number of animations that are required. Keep in mind that the names specified, are not the names of the animations that will be used by the RF ini files. RF uses the name of the actual MOT file, that is contained within the MOT file itself. Before going on to specifying the weapon definitions, be sure to know the names of the animations and the name of the joint at the tip of the barrel. If it was the second joint added with Milkshape, then by default it will be named, joint2.

With the actor compiled, copy it to the media\actors directory. Now that a modeled weapon is available as an act file with the necessary motions, it can be defined as a weapon and used within a game.

Adding Weapon Definitions

Act files which are to be used in RF as weapons, are defined in the weapon.ini file. Double click the weapon.ini file in the RF install folder to view it in NotePad. Scroll down to the section that has the heading, [9mmAR]. This is the weapon definition for the 9mm pistol used in the RF demo.

Highlight all of the definition data, then copy and paste it at the end of the file. Modify the newly pasted entry as follows. The parts in bold are the altered entries.

[MP5]
type = weapon
slot = 5
firerate = 0.05
catagory = projectile
projectile = pistolbullet
attribute = health
ammunition = pistolbullet
ammopershot = 1
shotpermagazine = 30
loosemag = true
worksunderwater = true
attacksound = weapon\bulletfire.wav
reloadsound = weapon\reload3.wav
emptysound = weapon\click.wav
muzzleflash = PMuzzleFlash
;
; 1st person
;
viewactor = MP5.act
viewfillcolor = 128 128 128
viewambientcolor = 128 128 128
viewrotation = 0 0 0
viewoffset = 0 0 0
viewscale = 1
viewlaunchoffset = 0 0 0
viewanimspeed = 1
viewarmanim = arm
viewidleanim = idle
viewattackanim = shoot
viewwalkanim = walk
viewreloadanim = arm
viewkeyreloadanim = arm
viewattackemptyanim = shoot
viewlaunchbone = joint2
crosshair = crosshairs\pistol.bmp
crosshairalpha = crosshairs\a_pistol.bmp
crosshairfixed = true
recoilamount = 0
recoildecay = 0
;
; 3rd person

```
;
playeractor = mp5.act
playerfillcolor = 128 128 128
playerambientcolor = 128 128 128
playerrotation = 0 180 0
playerscale = 1
playerlaunchoffset = 0 0 10
playerbone = joint2
```

By altering these entries, a new projectile weapon has been defined for use within RF. Examining the changes, it seems that the new weapon is named MP5. The offset and position data has been reset to 0 and will be added in the next step. The animation names have also been changed, to match those for the new model. Weapon slot 5, is defined as the slot location. This weapon can be selected by pressing the 6 key. The viewlaunchbone setting defines the projectile launch point and is the name of the joint that protrudes from the barrel.

The ammunition used by the weapon is defined to be pistolbullet. This is a projectile definition, also defined in the weapon.ini file. For simplicity sake, it will be kept for the MP5. Before being able load the MP5 in RF though, one more modification must be made to the player.ini file. The player.ini file defines the attributes of the player. An MP5 attribute will have to be added and the initial value set to 1. Add the following to player.ini file.

```
[MP5]
initial=1
low=0
high=1
```

This will start the player off, with an MP5 already

accessible to them. The weapon can be selected by pressing the 6 key. With the definitions in place, the final step is to properly position the weapon in relation to the player. This is performed within an RF demo level.

Positioning the Weapon

In order to properly position a weapon in RF, a system of keys is used to orient the actor. While the actor is being rotated and positioned, it displays this information at the bottom right of the screen. This information is written down and then entered into the weapon.ini file. If the demo level described in this book has not been prepared, then quickly run the World Editor and create a small world from the default hollow cube. Add a playerstart entity, a playersetup entity and then save / compile it as a test level. This level will be used to position the weapon.

First run the RF INI Editor Tool and select the check box marked, Weapon Positioning On. This will activate the weapon positioning keys, which is the number pad of the keyboard. Once again, if the demo level has not been prepared, then set the Start Level option in the RF INI Editor, to run the test level that has been quickly assembled for this. Save the ini file and exit the editor.

The next step happens within the game itself, so run Reality Factory and go to the Options Menu. Select the Debug menu and check the box marked, Debug Info. Now go to the Controls Menu, select Advanced and check the box marked Crosshair. This will display all the data that is needed, to properly position a weapon and turn on the game crosshair.

Select the weapon by pressing 6. In most instances the weapon will be in a position that is out of the players view. It is offset and rotated, by using the number pad of the keyboard. The data in the bottom left of the screen, is the weapons current scale and position. The diagram above shows the keys used to move and rotate the weapons position. The 0 key, is a shift that toggles rotation. For example, to move the weapon up, the 4 key is used. To rotate the weapon counter clockwise, 0 is held down while the 1 key is pressed.

In most cases, the weapon will have to have the Y rotation set to 90. This is because the weapons is sometimes accidentally modelled side ways, in the front viewing window of the modeling program used to make it. Rotate the weapon 90 degrees if necessary and start to reduce the scale using the 9 and 6 keys. Continue to position the weapon until it suits the game view. On the next page is a screenshot of the MP5

in a game. Notice that the Ghost Ring Site of the weapon, is lined up with the location of the crosshair. This lets the player use the actual crosshair of the weapon. When the weapon has been positioned, write down the offset and rotation data.

To make the changes permanent, open the weapon.ini file and add the position data to the following lines. Here is the position data for the MP5.

viewrotation = 4.5 -88 -2
viewoffset = -1.0 -4.4 12.5
viewscale = 0.35

Save the weapon.ini file and run RF again. This time when the weapon is selected, it will be in the position that is was set to. When the weapon fires off 30 rounds it will automatically reload and play the arm animation when it does. There should also be a slight recoil every time it fires. The amount of the recoil will have to do with the scale of the weapon and may need some adjusting to look right.

The process that was outlined, is similar for all first person weapons. Melee weapons will require two animations

for attacking and RF will switch between them randomly. For the most part however, almost any model can be brought into RF and used as a projectile weapon.

Creating Enemy Pawns

Right along side weapons, enemies are an integral part of many RF games. They are responsible for bringing the level to life, so to speak. This chapter does not directly discuss modelling methods, since that is a preference for the reader. The focus is more towards what animations will be required and what to consider when creating the model.

Animations for enemy pawns can be added in one of two ways. Bones can be assigned to a mesh and then animated, or a mesh can be designed around an already animated skeleton. Which method to use, depends on the tools and the media that is available.

Enemy Animations

Before continuing, it is important to understand what the basic animations required to create an enemy are. Below is a minimum list of animations that should be found within an enemy pawn.

- Idle
- Walk
- Shoot
- Die

This is the absolute minimum that a pawn will need to interact with the player as an enemy. Transitional animations from one animation to another, can be created by the engine using the script command, BlendToAnimation(). However, to add variety to the actions of the enemies in the level, it is a good idea to have multiples of the same animation. For example, a pawn should at least have two or three die animations. The more animations that a pawn has to perform the same task, the more varied the actions of the pawns will be. Of course, the random selection of these animations, must be written into the script. A more advanced set of animations would include:

- Idle 1, 2
- Shoot
- Melee 1, 2
- Turn Left
- Turn Right
- Walk
- Walk and Shoot
- Pain 1, 2
- Die 1, 2, 3

This is a more complete set of animations, that can cover a wider range of events and actions. If multiple pawns are in view at once, then they will have varied actions making them seem more like ‘individuals’. With this set of animations, the pawn is capable of performing the following tasks:

- Stand ready
- Fire at the player while standing still
- Fire at the player while walking towards the player.
- Attack the player with a melee combination.

- Follow a Script Point Path patrolling for the player.
- Display that it has been injured.

When creating these animations, remember that each must be exported as a unique file and not as a large single file of combined animations.

Using Milkshape

Many meshes are available on the Internet as royalty free downloads, for individuals to use in the creation of their games. Milkshape3D is very easy to use and with a few other programs it can easily create an animated model from a ready mesh. However, because of a few issues revolving the animation system it uses, there are a few considerations that must be addressed.

Many models available on the Internet are available in 3DS format. However, Milkshape does not import 3DS models. This is overcome by using the freeware program LithUnwrap. A 3DS file can be opened with LithUnwrap and then saved in MS3D format. Be sure that is saved to the same folder, that contains the materials.

With the file now in Milkshape, it requires the skeleton to be added and the vertices assigned to each joint. The first points to consider is, the position of the root bone above the ground plane. The feet of the mesh must be at Y=0 and it should be moved into position, before any joints are added to it. It is also important to remember that all vertices must be assigned, for a BDY file to be successfully exported. This can be taken care of by selecting the root joint and then assigning

every vertex to this joint, before any other vertex assigning. This will ensure that no vertices are missed.

Once the skeleton has been assigned to the mesh, the file should be saved and the BDY file exported. This MS3D file will be opened, each time a new animation needs to be created. The next step, is to create the MOT animations. Each animation should be saved as a unique file. Save the original MS3D file under another name and begin to create the animation. When a new animation is started, it should be from the MS3D of the non-animated skeleton. Each animation is exported as a unique MOT file.

Milkshape Seamless Animations

When creating animations with Milkshape, the most important factor to consider is the duration of the animation. Milkshape can not create an animation that is less then 1 second. One keyframe in Milkshape, is one second long. This means that in many instances, an animation will have to be made out to last many seconds and then scaled down using the Milkshape Scale Animation tool. It is important to consider this when scripting the pawn, since it is the animation duration that will control many factors in the script. More about this in the chapter on pawn scripting.

Milkshape also can not create a completely seamless animation. When a MOT is exported from Milkshape, an extra keyframe is placed between the end of the last frame and the first frame. This is keyframe 0 and can not be edited with the Milkshape animation system. In order to create a seamless animation using Milkshape, the animation must be manually

edited to start at frame 0 and not frame 1. The following is an example of how to accomplish this, using WordPad and the Scale Animation Tool.

Assume that a model has been animated to walk, using Milkshape. It has 6 keyframes, with number 1 and 6 being exactly the same, to simulate a seamless motion. At this point, the exported animation will last 6 seconds and have a 1 second ‘pause’ between the motion. In order to achieve the right effect in the game though, the animation must be made to last only 1 second and the pause between the start and end of the motion, must be removed. First the animation is scaled down to 1 frame with the Scale Animation tool. Then it is exported as a Milkshape3D ASCII file, to create an editable file with the extension, txt. The ASCII file is opened and edited using WordPad, in order to remove the unwanted frame between the end and start frames.

Start by searching for the word, **Bones:** . CTRL-F will open up the Find Text dialog box. The cursor should end up at a point in the file, that then displays the text similar to the following:

Bones: 18
"joint1"
""
24 0.771498 54.517002 -10.333334 0.000000 0.000000
0.000000
6
0.166667 0.000000 0.000000 0.000000
0.333333 0.000000 0.000000 0.000000
0.500000 0.000000 6.000000 -1.500000
0.666667 0.000000 0.000000 0.000000
0.833333 0.000000 8.000000 0.000000
1.000000 0.000000 0.000000 0.000000
6
0.166667 0.000000 0.000000 0.000000

0.333333 0.000000 0.000000 0.000000
0.500000 0.000000 0.000000 0.000000
0.666667 0.000000 0.000000 0.000000
0.833333 0.000000 0.000000 0.000000
1.000000 0.000000 0.000000 0.000000

This information is present for each bone in the model. It is motion data for each keyframe. The numbers that are in bold print, represent the start times for each keyframe. The data above shows that the animation sets the start of the first keyframe at 0.166667 seconds and lasts for 1.0 second. It must be modified to start at 0.00000 in order to remove the 0.16 second time lag, before the start of the first keyframe. Below is the same data, now modified to be seamless. This was performed for each section of data that represents each bone. Be very careful when editing the ASCII file, since any subtle changes in the wrong spot can effect the entire model.

Bones: 18
"joint1"
""
24 0.771498 54.517002 -10.333334 0.000000 0.000000
0.000000
6
0.000000 0.000000 0.000000 0.000000
0.200000 0.000000 0.000000 0.000000
0.400000 0.000000 6.000000 -1.500000
0.600000 0.000000 0.000000 0.000000
0.800000 0.000000 8.000000 0.000000
1.000000 0.000000 0.000000 0.000000
6
0.000000 0.000000 0.000000 0.000000
0.200000 0.000000 0.000000 0.000000
0.400000 0.000000 0.000000 0.000000
0.600000 0.000000 0.000000 0.000000
0.800000 0.000000 0.000000 0.000000
1.000000 0.000000 0.000000 0.000000

The modified data now starts the animation off at 0.00 seconds, instead of after 0.16 seconds. This file is saved and imported back into Milkshape by selecting, *Import* -

Milkshape3D ASCII. Play back the animation and examine the difference. The animation is now seamless. Seeing how easy it was to correct the data on 6 keyframes, this number is a good choice for animations in Milkshape, that are meant to be seamless. This new animation is then exported as a MOT file.

Using a Pre-Made Skeleton

The above was a brief examination of the steps involved with animating a mesh in Milkshape. Now the focus will turn towards using a pre-made skeleton, that can be extracted from an already animated act file, such as the player actor. Milkshape3D is capable of decompiling an already compiled Genesis Act file. By doing so, the skeleton and the resulting animations, can be used to create a new actor without the need to create new animations. In this example, the default Ernie actor will be decompiled and the skeleton will be used to outline a new mesh.

First begin by creating a new temporary working folder and copy the Ernie actor to there. Run Milkshape and select, *Tools - Decompile Genesis3D Act*. Locate the Ernie actor in the temporary folder when prompted and select OK. Ernie will be decompiled in the folder, creating a BDY file and a large number of MOT files. Each file will have the Prefix Ernie_. The file with the skeleton is the BDY file. Import this file into Milkshape by selecting, *Import - Genesis3D BDY*.

The Ernie mesh and skeleton will be displayed in the viewing ports. To get to the skeleton, select the faces of the mesh in Milkshape and delete them so that only the skeleton

remains. Next select all the joints of the skeleton and use the Clear button, found under the Joints Tab in Milkshape. Each joint must be individually chosen through the Joint Tab, for the joints to be cleared of any assigned vertices that no longer exist.

What remains is a skeleton that a mesh can be modelled around. The basic process involves taking a screen shot of the skeleton and sketching the mesh outline around it. This drawing is then used as the backdrop in Milkshape to create the mesh. Again the vertices are then assigned to the joints and a new BDY file is exported. An excellent example of this, is the CS Girl Tutorial, found in chm format at the RFDN.

Once a new BDY file has been created, the BDY and MOT files can be compiled using the RF Actor Studio. One thing to keep in mind however, is that although the MOT files were given the Ernie_ prefix when the file was decompiled, within each MOT file itself, the name is still as it was without the Ernie_ prefix. This means that when referring to the animation name in RF, the name is still the original animation name and not the new one that starts with Ernie_. This is because the actual animation name, is stored within the MOT file itself.

Introduction to Pawn Scipting

One of the most powerful features found in Reality Factory, is the ability to add a scriptable actor entity. In RF, these entities are named Pawns. The pawn entity is described in the chapter titled Bringing Actors Into RF. This chapter introduces the reader to the Pawn Scripting Language and prepares them to understand the next chapter, which describes specific script code.

The scripting language in RF, is based on a C style script handler called Simkin. It will help to read the Simkin documentation, specifically the description of the included operators. Aside form the operations which are allowed in Simkin, RF also has a list of unique commands and functions, that specifically control aspects of the pawn that is running the script. One thing to keep in mind before beginning to write scripts, is that the scripting system is not like many of the common languages. It is actually quite unique as to how it works and must be examined carefully before starting to write scripts.

How to Use Scripts

Pawn scripts are placed in the RF Scripts folder, found in the root directory. Scripts have the extension .s, but this is not a mandatory feature. Pawns use the script with the entry

in the pawn entity named ScriptName. This entry contains the name of the script, that the pawn will use. The pawn entity, also requires an entry that specifies the first Order to be executed by the pawn. This is specified in the entry named SpawnOrder.

What is an Order

A pawn script can be regarded as a collection of functions that each contain a set of specific script commands. Each one of these functions, is called an Order. Below is an example of an order.

```
Spawn[()
{
NewOrder("Start");
}]
```

At this point, all that is important to consider, is that an order starts with the name of the order and the identifier that an order is started. In this case, the order is named Spawn. All commands within an order, must be kept within curly brackets. An open bracket specifies that start of the command list and a close bracket specifies the end of the list. The final square bracket, marks the end of the order. In this particular example, a command named NewOrder is used. This command is telling the script, to start another order named Start.

All scripts, start and end with a curly bracket as well. An open bracket is the start of the script and a closed bracket marks the end. Every script file must have the first line as {

(open bracket) and the last line as, } (close bracket).

High and Low Levels

Pawn scripts have two levels of operation, a low level and a high level. Commands that are considered Low Level, will not work when the script is running at a High Level and vice versa. However, Functions can be called at any level.

The main difference between HL and LL is that a LL order constantly loops and the looping interval can be specified at the start of the order. HL order do not loop, unless they contain a command that specifies the order to restart and they do not run at specified intervals.

By default, scripts start running in High Level mode. This means, that the first order can only contain High Level commands or functions. In order to switch between Low Level and High Level, a command that instructs the script handler to switch levels is used. Level switching commands are always accompanied by the name of an order to switch to. To examine this further, look over the simple script below, made up of only two orders.

```
{
Spawn[()
{
LowLevel("RunLow");
}]

RunLow[()
{
self.ThinkTime=0.1;
```

}]

}

The curly brackets in bold, mark the start and end of the script. The two orders within the script are named Spawn and RunLow. Below is a breakdown of the script.

{	The first line in every script is an open curly bracket. This marks the start of the script.
Spawn[()	The first order on the script is named Spawn. This is the name of the order in the pawn entity entry, SpawnOrder.
{	All commands within an order are enclosed within curly brackets.
LowLevel("RunLow");	The command instructs the script handler, to switch to Low Level processing and go to the order named RunLow.
}]	All orders end with a curly bracket and a square bracket.
RunLow[()	This is the name of the second order in the script. This order will contain only Low Level Commands.
{	Again, a curly bracket marks the start of the order.
self.ThinkTime=0.1;	This command sets the looping interval for this order, to run every 0.1 seconds or 10 times a second.
}]	The order is closed with a curly bracket and a square bracket.
}	All scripts are closed with a final close curly bracket.

One thing that should be pointed out is that the script handler does not automatically change from HL to LL and vice versa. It must be instructed to do so and given a new order that contains the appropriate level commands. If the

script handler is instructed to switch levels, it will only understand commands for the specific level.

High Level Events vs Low Level Looping

HL processing does not loop an order automatically. For a HL order to restart itself, it must use the command RestartOrder() or use the command NewOrder(), to switch to another HL order. If these commands are not found by the end of the order, then the script handler idles and waits for an event to occur. Events are set at a high level and the parameters that activate the events, are constantly monitored by the script handler. When the specified event occurs, the order which the event calls is executed. Events can include a trigger going high, a flag changing state, the pawn losing attribute, an enemy being spotted or the end of a pre-defined time period.

What is very important to know about HL event handling and the relation to LL handling, is that when the script handler is instructed to start processing LL commands, all monitoring of HL event parameters will cease. No events that were set to run in a previous HL order, will be called if the script is instructed to run at a low level. However, as soon as the order switches back to HL command processing, the events will take place.

Switching Between Orders and Level

There are a few considerations regarding switching between levels and orders, that are important for the script

writer to understand. The first consideration, is that when switching from a High Level to a Low Level order, all HL command processing will cease. Now this was stated previously, but what was not said is that the change will occur instantly and because the HL commands take place at their own speed as required, any commands that did not finish processing will not be called. This can lead to undesirable results. With this in mind, it is important to allow the switching from HL to LL, to occur only after all HL commands in the order have been processed. More about this in the next chapter.

How to properly switch from LL to HL, also needs a bit of discussion. The command to go from a low to high is, HighLevel(). This command accompanied by an order name, will switch to that order, at the end of the current LL order that is running. The problem, is that as soon as the script interpreter encounters the command HighLevel(), it automatically switches to HL processing. However, the LL order will still keep running until it is done. This will cause the script to return an error, stating that it has encountered an unspecified command. In order to prevent this, the LL command, *return 0* is used. This command tells the script interpreter, to stop the current script. Below is an example of a LL order that switches to HL.

```
RunLow[()
{
self.ThinkTime=0.1;
HighLevel("Spawn");
return 0;
}]
```

Switching between HL orders is as simple as using the NewOrder() command. There is not much to worry about, except to keep in mind that a call to an event, will always over-ride the currently running HL order.

Low level switching, is accomplished by setting the self.think variable. This variable instructs which LL order, should be run at the end of the current LL order. Below is an example of a LL switch to another LL order, named RunLowTwo.

```
RunLow[()
{
self.ThinkTime=0.1;
self.think="RunLowTwo";
return 0;
}]
```

Functions and Global Variables

Some functions and variables can be called at any time during a script, regardless of the level that it is running at. An example of this, is the random() function. It can be used at any time to return a random number, that is between two specified numbers. An example of a variable that can be accessed at an level, is the self.DifficultyLevel variable. This value is the current difficulty level as chosen by the user at the start of the game. An easy setting has a value of 1, a normal setting is 2 and a hard setting is 3. This can be used to give a varying degree of game play to the level enemies. Note that this option is selected using the RF INI Editor to check the box marked, Use Difficulty. Next is an example of a

pawn script that when run, will switch to a LL order and display a random number between 1 and 10, once every second.

```
{
Spawn[()
{
Console(true);
LowLevel("ShowRandom");
}]

ShowRandom[()
{
self.ThinkTime=1;
debug(random(1,10));
}]
}
```

The HL command Console(true), instructs the script handler to display the current order and reports errors in the top left of the screen during the game. The LL command debug(), is used to display the value of a variable underneath the console display. In this case, the value is a random number between 1 and 10, generated by the random() function.

User Defined Variables

Variables can also be defined by the script writer and used throughout any order in the script. A variable is declared at the start of a script before any orders. Below is the same example script showed previously, but with a numeric value

and a string value defined.

```
{
NumVal [10]
Stringval [ShowRandom]

Spawn[()
{
Console(true);
LowLevel(Stringval);
}]

ShowRandom[()
{
self.ThinkTime=1;
debug(Numval);
}]
}
```

In this example, two variables are defined at the top of the script. A numeric integer named Numval, is set to 10 and a string variable named Stringval, is set to contain the string ShowRandom. Note that the string variable is not in quotations and that neither variables have a semi-colon at the end of the line. The string variable, is then used to replace the name of the order to switch to, using the LowLevel() command. Notice, that there are no quotes around the variable name. The numeric value, is displayed on screen using the debug() command.

Common Script Errors

When Console(true) is set as the first line of the Spawn Order, it will display any script errors in the top left corner of the screen. However, there will be times when no error is displayed, but what is expected to happen does not. Other times, there will be no display shown at all. The errors are caused by mistakes which the interpreter can not process or does not seem like a mistake in the script. An example of this is the following script.

```
{
Spawn[()
Console(true);
}]
}
```

Notice that there is no open curly bracket, to mark the start of the order command sequence for the order. This type of error confuses the script handler. Another common mistake is to leave out quotations around strings. The script handler then treats it as a variable name and not as a string. Below is an example.

FireProjectile(FireBall,joint12,0,-2,0,health);

This should read:

FireProjectile("FireBall","joint12",0,-2,0,"health");

The script handler would regard FireBall, joint12 and health as the names of variables and not actual string names. The console would report an error saying, unidentifed variable.

Writing Pawn Scripts

This chapter discusses methods of controlling a pawn, to act as an enemy against the player. However, there are many other uses for pawns, many of which are examined in chapters that discuss performing specific tasks in RF. For now, the focus will be on writing a script that can turn the Robot actor form the media pack, into a player enemy.

Adding the Robot Pawn

The first task is of course, to add the Robot pawn definitions to the pawn.ini file. Use NotePad to edit the file and add the following entry.

```
[Robot]
actorname =android1.act
actorrotation = 0 180 0
actorscale = 0.6
fillcolor = 128 128 128
ambientcolor = 128 128 128
subjecttogravity = true
boundingboxanimation = idle
shadowsize = 64
```

Then copy the file named android.act to the RF media\actors folder. This actor can now be used as a pawn in

Pawn Entity

- **PawnType** - Robot
- **ScriptName** - robot.s
- **SpawnOrder** - Spawn

a level.

For these examples, create a new world with one of the world editors and start it off with a 1024 x 1024 x 256 hollow cube. Add a playerstart entity, a playersetup entity and a pawn entity. Set the entries in the pawn entity to match those in the side bar. Place the pawn at one end of the room and the player at the other end. Have them facing each other and position them as close to the floor as possible, without actually being inside it.

Starting the Script

Run NotePad and save a new file as robot.s, in the RF Scripts folder. Be sure that the extension is .s and not .txt, by selecting All Files from the file type setting or by typing the name in quotations. First add an open curly bracket, a few carriage returns and then a close curly bracket. The orders for the script will be added between these two brackets. Now spawn order is added. Save the script as follows.

```
{
Spawn[()
{
Console(true);
}]
}
```

Now this script does nothing but instruct the interpreter to display the console, but it is a start. The next commands, will set the necessary parameters to define the pawn as a player enemy and define the amount of attribute given to the enemy.

Creating an Enemy

Add the following commands to the order named Spawn and then examine the explanation of each below. These commands instruct the pawn, to regard the player as an enemy. They only need to be called once during the script.

```
{
Spawn[()
{
Console(true);
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
}]
}
```

The pawn will now regard the player to be hostile, but not any other pawns that it sees. When the player comes into

SetFOV(360);	This command set the Field Of View for the pawn, to be 360 degrees. The pawn can see all around.
HostilePlayer(true);	This tells the pawn to consider the player to be an enemy and when seen, target it as one.
HostileSame(false);	The pawn is instructed to not consider any other pawns in the same Group, to be an enemy.
HostileDifferent(false);	The pawn is instructed to not consider any other pawns in a different Group, to be an enemy.
SetGroup("Enemy");	This command sets the name if this pawn Group. In this case the group is named, Enemy.

the field of view of the pawn, the pawn will register the location of the player, as a location of an enemy.

Adding Pawn Attributes

Pawns can be ‘killed’ by the player, only if given an Attribute that the player weapon can damage. In order to define an attribute for the pawn, the command AttributeOrder is used. Add the following to the script.

```
{
Spawn[()
{
Console(true);
AttributeOrder("health",20,"Die");
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
}]

Die[()
{
Remove(true);
}]
}
```

At this point, the test level can be compiled and the robot pawn should appear where positioned. There should be a display in the top right corner of the screen, saying something along the lines of, **Pawn001 Spawn SetGroup.** If the pawn is attacked by a weapon that removes the attribute

named health, then after 20 points have been removed, the order named Die will be called. The order Die, contains the command Remove(true). This command removes the pawn from the level and stops processing that particular pawns script. The order die will only be called, if the script interpreter is running at a High Level. If the script is running at a low level, then the order will not be called when the 20 health points are removed. A means of detecting a pawn death while running at Low Level, is examined later in the chapter.

Finding a Target

Before any interacting with a targeted enemy can be performed with script commands, an enemy must be targeted by the pawn. This means that the player must come within the set FOV of the pawn, before it can be interacted with. The commands used to locate an enemy, include the HL commands, FindTargetOrder() and AddDistOrder(). This example will examine the command FindTargetOrder(). Add the following to the script.

```
{
Spawn[()
{
Console(true);
AttributeOrder("health",20,"Die");
FindTargetOrder(200,"Alert","health");
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
}]
```

```
Alert[()
{
AnimateStop("alert",0,"");
}]

Die[()
{
Remove(true);
}]
}
```

Here is a breakdown of the FindTargetOrder() command.

FindTargetOrder(200,"Alert","health");

- **200** - This value sets the distance texels that the pawn can see. In this case, the player must come within 200 texels to be 'seen' by the pawn.
- **Alert** - This is the name of the Order that will be called when the target is found. It will only be called while the script is running at a High Level. The order will only be called once and then the FindTargetOrder() command must be used again, in order to re-target the player. Tthis is only required, if the player goes out of the pawns defined FOV.
- **health** - This is the name of the attribute that is targeted by the pawn. The player must have an attribute named health, defined in the player.ini file. The pawn will only target an enemy, that has this attribute specified. Note that these entries are case sensitive.

Examining the breakdown, it seems that the command used in the script will call an Order named Alert, when an enemy with the attribute health, comes within 200 texels inside the pawns FOV.

The new Order named Alert, plays a pawn animation using the AnimateStop() command. The three parameters used by the command specify that the pawn is to play an animation named alert and hold it at the last frame. When the animation is complete, the script handler is to wait 0 seconds before going on to the next command when. The empty quotations, defines no sound file to be played during the action.

Save the file and compile the test level. Now, when the player enters within a 200 texel area of the pawn, the pawn will play the alert animation. This raises the weapon from the resting position, to the aiming position. The player is now targeted by the pawn and can be interacted with as an enemy.

Firing a Projectile

At this point, the pawn can detect the presence of the player and react accordingly. Now it is time to give the pawn the ability to attack the player. This will be accomplished using the High Level command, FireProjectile(). Modify the Alert Order and add the following order named Attack, to the script as well.

```
Alert[()
{
```

```
AnimateStop("alert",0,"");
NewOrder("Attack");
}]

Attack[()
{
FireProjectile("pistolbullet","HandR",0,0,0,"health","");
Delay("stattack",1,"");
RestartOrder();
}]
```

The new command added to the Alert Order simply instructs the script to switch to a new order named Attack.

FireProjectile()	The entry pistolbullet defines the projectile to be fired. This is a projectile defined in the weapon.ini file. HandR is the name of the joint in the robot actor from which the projectile is fired from. The values 0,0,0 are an X,Y,Z offset for the point from which the projectile will fire. The empty quotations would define a sound that will be played when the projectile is fired. In this case, no sound is specified. Note that there are a total of three different versions of this command. This example uses the High Level method. A later example will examine the Low Level methods.
Delay()	This command, obviously enough, generates a delay within the script. The parameters specify the animation to play, the duration of the delay and the empty quotations define a sound file to be played. In this case, no sound file. Audio will be added later on. The purpose of this command within the script is to do two things. One it plays the attack animation for the correct duration and two, it allows for a firing delay between shots. When the delay is over, the script runs the next command, RestartOrder() which simply tells the script to restart the order and fire again.

At this point, when the script is compiled, the robot actor should be standing idle and when the player approaches it, the alert animation starts and the robot begins to fire at the wall. The wall being fired upon, should display the bullet explosion for the pistol bullet projectile once a second as the robot fires. If this does not happen, check the script to make sure that there are not any typos, that may be causing an error.

Making a Turret

All that is required to make the robot act as a turret which can fire upon the player when approached, is to make the robot face the player before it fires. A single command, is all that is required to do this. Modify the Attack Order to add the following command.

```
Attack[()
{
RotateToPlayer("wattack", 160, false, "");
FireProjectile("pistolbullet","HandR",0,0,0,"health","");
Delay("stattack",1,"");
RestartOrder();
}]
```

RotateToPlayer is a very useful command that turns the Robot into a dangerous player enemy. By adding this command, the robot will turn to face the player before firing at it. Here is a breakdown of the parameters.

- **wkattack** - This is the name of the animation to be played while the rotation is occurring. In this case the animation is

named wkattack.

- **160** - This value defines the speed of the rotation in texels per second. The higher the value, the faster the rotation.
- **false** - If this value was set to true, then the robot would also rotate the pitch to point at the player. However, because the robot and the player are both standing on the same plane, there is no need to change the pitch of the robot as well.
- **“”** - Once again the empty quotations is where a sound file would be defined. No sound will be used for the rotation.

Compile the level and test out the script. The pawn should be a bit more of a challenge now, actually being able to kill the player. At all times it will turn to face the player and fire a projectile. To make it a bit more versatile, a means of detecting the player range, will allow the robot to only fire when the player is within a specific range. Add the following command to the Alert Order and then add another order named LostTarget.

```
Alert[()
{
PlayerDistOrder(-256,"LostTarget");
AnimateStop(“alert”,0,””);
NewOrder(“Attack”);
}]

LostTarget[()
{
BlendToAnimation("idle", 1, false,"");
FindTargetOrder(200,"Alert","health");
```

}]

Before examining a breakdown of the commands, it is important to understand what this does in general. This addition to the script, will have the pawn stop firing when the player goes out of range and then return to its idle animation. It then restarts the checking for a target. Below is a breakdown of the parameters for the commands, PlayerDistOrder() and BlendToAnimation().

- **-256** - This parameter for PlayerDistOrder, defines the distance that will trigger the order to be called. In this instance, it is a negative value. (-256) This means that the order will only be called when the player moves out of the specified distance. In this example, the maximum shooting distance is 256 texels from the center of the pawn.
- **LostTarget** - This is the name of the order that will be called when the player moves out of the 256 texel radius defined in the first parameter.

- **idle** - The first parameter to the BlendToAnimation command, defines the final animation that the current keyframe is blended to. The blending of the two animations is handled buy RF and can be used to generate transitional animations.
- **1** - This defines the amount of time in seconds that the blend will last for. In this instance, the blended animation will last for 1 second.
- **true** - When this parameter is set to true, the animation defined in the first parameter, will play completely, before going on to the next command. For this example, the pawn will play the idle animation after the blend.

Compile the level and test the additions made to the script. Now the pawn will only fire at the player, while it is within 256 texels it. When the player goes out of this range, the attack animation is blended back to the idle animation and FindTargetOrder is called again, so that the whole process can restart itself.

Moving Pawns

A stationary turret is one thing, but to make the pawn come to life, it needs the gift of motion. There are basically three ways to move a pawn, using either Script Point paths, High Level free movement commands or Low Level free movement commands. For this example, first a High Level free movement command will be used and then this will be tied in to a Script Point method. Add the following line to the order named Attack.

```
Attack[()
{
RotateToPlayer(“wattack”, 160, false, ””);
FireProjectile(“pistolbullet”,”HandR”,0,0,0,”health”,””);
// Delay(“stattack”,1,””);
MoveForward("wkattack",50,50,"");
RestartOrder();
}]
```

First, notice the two forward slashes placed in front of the Delay command. This defines the line to be a comment and is ignored by the script interpreter. It will be used again later, but for now it will be replaced by the command MoveForward(). Here is a breakdown of the parameters.

wkattack - This is the animation to loop while the pawn is traversing the specified distance. In this case it uses an animation named wkattack.
50 - The first value specifies the speed, in texels per second. This is how fast the pawn will travel. This is important to be in relation with the next value.
50 - This specifies the distance that the pawn will travel. It is set to 50 because the speed is also set to 50. This is to ensure that the movement will last for 1 second. This is also the length of the animation used by the pawn.
“” - Once again the empty quotations is where a sound file would be defined if used.

What happens now, is the pawn will walk towards the player for one second, while it fires one shot. It then rotates to face the player and performs the same task again, until it finally runs into the player. What is required is a means of testing the players distance from the pawn, so that the pawn will stop from running into the player. Add the following to the attack order.

```
Attack[()
{
RotateToPlayer("wkattack", 160, false, "");
FireProjectile("pistolbullet","HandR",0,0,0,"health","");
        if (self.player_range>96)
        {
        MoveForward("wkattack",50,50,"");
        }
        else
        {
        Delay("stattack",1,"");
        }
RestartOrder();
}]
```

The order has now been changed to test the value of the global variable, self.player_range. This variable holds the distance in texels, from the pawn to the player. The condition is tested using the **if** command. The syntax for **if**, is shown below.

if (*condition* = *true*)
{
Command Statements
}

Condition=true, represents a value to test. In this case, the condition to test is if the variable self.player_range, is greater then 96 texels. If it is greater then 96 texels, then the condition is returned as true and the commands within the curly brackets are executed. In this example, the condition test is expanded using the **else** command. Below is the syntax for using **else**.

if (*condition* = *true*)
{
Command Statements
}
else
{
Secondary Command Statements
}

The **else** command can only be used with the **if** command. Together, they make for a system that can test for a particular condition and run one of two command sets, depending on the result. Examining the modification made to

the Attack Order, it seems that the **if** / **else** command checks to see if the player is more then 96 texels away and if so, walks 50 texels toward the player. If the player is within 96 texels of the pawn, then the pawn will stand and fire, without walking towards the player.

Using Script Points

Script Points, are markers that reference a location in the level. They are used to direct the pawn and define motion objectives. The most common application using Script Points, is to guide a pawn with a camera attached to it and is used to create in-game movies. However, pawns can also be teleported to a script point, making them capable of being re-used within a level. In the next example, a Script Point is used to start the pawn at its original position before it has found a target. This is scripted to occur, when the pawn is destroyed by the player. Add a Script Point to the test level and position it at the same height of the pawn entity. Offset it in any direction by 8 texels. Change the szEntityName entry for the Script Point to, **point01**. Then add the following to the order named Die and remove the command, Remove(true).

```
Die[()
{
PlayerDistOrder(0,"");
FadeOut(0,2);
TeleportToPoint(“point01”);
FadeIn(255,5);
AttributeOrder("health",20,"Die");
NewOrder(“LostTarget”);
}]
```

Here, three new commands are introduced. FadeIn, FadeOut and TeleportToPoint. These commands do exactly what their names imply. All that is required to use them is an understanding of the parameters. For TeleportToPoint, the parameter is simply the name of the Script Point. For FadeIn and FadeOut, the first parameter is the time it takes in seconds, to fade to a specified transparency value. The second parameter defines a transparency value that is between 0 and 255. Completely transparent is a value of 0 and 255 is completely solid. In this example, the FadeOut command has the pawn become completely transparent in 2 seconds and FadeIn has the pawn go from transparent to solid in 5 seconds. In between these actions, the pawn is tele-ported to the point named, point01. Compile the test level and do battle with the pawn. Every time it is destroyed, it will fade out and tele-port itself back to the start position. It then returns to the idle animation and waits for the player to come into view again.

The most important command in the new death order though, is the first line, PlayerDistOrder(0,""). This turns off the distance checking that was set in the Alert Order. If this command was omitted, then as soon as the pawn is tele-ported away, the distance checking would call the order LostTarget. This would make the Die order jump to the LostTarget order and not run the last three commands. The result would be a pawn that is tele-ported to point01, but remains invisible and can not be destroyed. The last three commands, are there to fade the pawn back to a solid transparency and then reset the health attribute assigned to it. With the distance checking properly disabled, the script will run the Lost Target order, only after the fading in has finished and the attribute is reset.

When testing the script, one may notice that when the pawn is tele-ported back to point01, it is still facing the direction it was when it was destroyed. The pawn needs to be rotated, to face the same direction as the Script Point is in the world editor. Add two more commands to the Die order.

```
Die[()
{
PlayerDistOrder(0,"");
FadeOut(0,2);
NewPoint("point01");
RotateToAlign("idle",500,false,"");
TeleportToPoint(“point01”);
FadeIn(255,5);
AttributeOrder("health",20,"Die");
NewOrder(“LostTarget”);
}]
```

The first command NewPoint(), is used to define the reference point for the pawn. In this case, it references point01. When a pawn has to interact with a Script Point, the pawn needs to have that Script Point defined as the, Current Script Point. This is because many of the commands that interact with a script point, do not use a Script Point name as a parameter. RotateToAlign, is an example of one of those commands. The parameters for RotateToAlign are as follows.

- **idle** - This is the animation to play while the rotation is occurring. In this instance, the pawn is set to play the idle animation.
- **500** - This is the rotation speed in texels per second. Because the command is called while the pawn is

invisible, it is done very quickly to just get the pawn into position as fast as possible.

- **false** - If this command was set to true, then the pawn would rotate to align itself with the pitch of the Script Point as well. In this case, it is not necessary so the parameter is set to false.
- **“”** - Once again, this is the parameter that defines a sound file. No sound is defined for this command.

Now when the level is tested, before the pawn is teleported to the Script Point, it will align itself to match the direction that the Script Point is facing.

Enhancing the Death Effect

Right now, all the pawn does when destroyed is fade out. The next enhancement adds an Explosion Effect and an animation before the fade occurs. Add the following to the Die order.

```
Die[()
{
PlayerDistOrder(0,"");
AddExplosion("GrenadeExplosion","Root",0,0,0);
        if(random(1,10)>5)
        {
        AnimateStop("die1",0,"");
        }
        else
        {
        AnimateStop("die2",0,"");
        }
```

```
FadeOut(2,0);
NewPoint("point01");
RotateToAlign("idle",500,false,"");
TeleportToPoint("point01");
FadeIn(5,255);
AttributeOrder("health",20,"Die");
NewOrder("LostTarget");
}]
```

Before examining the AddExplosion command, take a look at the system to animate the pawn. The if command is again used to check if a random number between 1 and 10, is greater then 5. If it is, then the first animation, die1 is played. If it is not greater then 5, then the second animation, die2 is played. This adds a bit of variety to the pawns death sequence. In order to add a nice looking effect to the death, the AddExplosion command is used. Below is a breakdown of the parameters.

GrenadeExplosion - This is an explosion defined in the explosion.ini file for RF07 Be sure to open the explosion.ini file and check to see that the heading is present. More about effects in a later chapter. For this example, a large sprite explosion is used, but any of the explosions in the ini file can be used.

Root - This is the name of the joint that the explosion is attached to. This is specific to the model. The android actor file that is found in the media pack for this book, has the Root Joint named, Root.

0,0,0 - These values define an X, Y, Z offset for the explosion. In this instance, there is no offset. These values are in relation to the angle that the pawn is facing.

At this point, the pawn is close to being able to go in a level and be used as an enemy. Right now though, it can only work in a test level that has no geometry. To make it more functional, it will need a means of avoiding obstacles.

Low Level Combat

Before continuing to examine a Low Level attack routine make a few adjustments to a couple of orders in the script. Modify the Spawn, the LostTarget and the Alert order.

```
Spawn[()
{
Console(true);
AttributeOrder("health",20,"Die");
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
NewOrder("LostTarget");
}]

Alert[()
{
AnimateStop("alert",0,"");
LowLevel("Attack");
}

LostTarget[()
{
PlayAnimation("idle", false,"");
```

```
FindTargetOrder(512,"Alert","health");
}]
```

What these changes have done is make the script a bit more efficient, increases the target distance and removes the PlayerDistOrder() command. This gets the script ready to use a Low Level Combat routine. It switches to Low Level from the Alert order. Re-write the Attack order to read as follows. Below the new order is a breakdown of the commands.

```
Attack[()
{
self.ThinkTime=0.05;
self.yaw_speed=160;
UpdateTarget();
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
}
```

self.ThinkTime=0.05;	Because the order runs at a Low Level, the order will constantly loop. The looping interval is specified to run the order every 0.05 seconds. That comes to 20 times a second. For the order to operate smoothly, the game will need to run at a minimum of 20 fps.
self.yaw_speed=160;	self.yaw_speed is a variable which can be set by the script and controls the Y axis rotation speed of the pawn. This variable must be set before any rotation can occur. Although in this instance it is called every time at the start of the order, it really only needs to be called once during the script.

UpdateTarget();	When an enemy is targeted by a pawn, it is the location of the enemy that is kept as a reference to the target. Since the player is constantly moving, this new target position needs to be updated manually using the UpdateTarget() command. The command is called once every time the order loops so this makes the pawns targeting very accurate.
if(self.enemy_vis=true)	The self.enemy_vis variable is a read only value that contains a true or false depending on whether or not the currently targeted enemy is in view of the pawn.
self.ideal_yaw= self.enemy_yaw;	This line contains two variables. The self.ideal_yaw variable sets the Y axis angle in radians, that the pawn should face. This will be discussed further a bit later. The variable self.enemy_yaw is a read only variable, that holds the Y axis rotation in radians, that the pawn must face in order to point at the currently targeted enemy. By setting the self.ideal_yaw variable to hold the self.enemy_yaw variable, the pawn will always have it's ideal rotation to face the player.
ChangeYaw();	The above command does not actually rotate the pawn, it only sets the self.ideal_yaw variable. In order to rotate the pawn, the command ChangeYaw() is used. This command begins to rotate the pawn towards the self.ideal_yaw value. More about this later on.

}]

This low level order returns the pawn back to the level of a turret. All the pawn will do is face the player at all times, a soon as it is targeted.

Low Level Motion

Moving a pawn using low level commands, is very different from how High Level commands move a pawn. A Low Level order, loops at a rate that is defined by the

self.ThinkTime variable. In this case, it is set to run at 20 times a second. This presents a smooth rate of rotation for the pawn, as it comes to its ideal yaw. The slower the ThinkTime for the order, the more ‘choppy’ the rotation would be. Before continuing, modify the order so as to give the pawn the ability to walk. Add the following to the Attack order.

```
Attack[()
{
self.ThinkTime=0.05;
self.yaw_speed=160;
UpdateTarget();
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
        if(self.enemy_range>128)
        {
        SetHoldAtEnd(true);
        Animate("wkattack");
        self.think="Walk";
        return 0;
        }
}
}]
```

Note, that this code is added before the closing bracket of the if(self.enemy_vis) condition tester. This makes it a part of the code that results from a true condition. What this code does, is prepare the pawn to switch to another order, that will execute the walking. Before adding the new order, examine the description of the newly added

commands.

- **if(self.enemy_range>128)** - This conditional test uses the variable, self.enemy_range. This is similar to the self.player_range variable, but instead holds the distance to the currently targeted enemy. In this case, if the distance is greater then 128 texels, then the commands within the brackets are executed.
- **SetHoldAtEnd(true / false)** - This command instructs the script handler to stop playing the animation at the end of it and not loop it. This will be used in the next order, to find out when the animation has come to its end. The command is called with a false value, in every instance before the order exits.
- **Animate("wkattack")** - This is a Low Level command that sets the current animation for the pawn. In this case, the animation is wkattack.
- **self.think="Walk"** - This command instructs the script handler to to go to the order named Walk, immediately at the end of this order.
- **return 0** - This instructs the interpreter to no longer continue with this order and restart.

As an overview, it seems that this bit of script, first tests to see if the visible enemy is further then 128 texels of the pawn and if so, set the animation to hold at the end, start the wkattack animation and then switch to another Low Level order named Walk. Now to add the Walk order. Add the following order to the script.

Walk[()

{

```
self.ThinkTime=0.05;
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
}
else
{
SetHoldAtEnd(false);
HighLevel("LostTarget");
return 0;
}
if(self.animate_at_end=false)
{
walkmove(self.ideal_yaw,50);
}
else
{
SetHoldAtEnd(false);
FireProjectile("pistolbullet","joint12",0,-2,0,"health");
self.think="Attack";
return 0;
}
}]
```

This order can be broken down into two parts. The first, is the conditional test for self.enemy_vis and the second part, is the conditional test or self.animate_at_end. One may ask why the test for self.enemy_vis is checked again, if it was just checked in the last order? The reason for this is that, the self.enemy_vis call, also updates the position of the enemy. If this was not checked, then the pawn would always consider

	the original position, to be the current position of the enemy. If UpdateTarget was used to fire at the enemy, it would do so without rotating the pawn to face the enemy. So the first section of the order is similar to the first section of the Attack order, in the sense that it uses ChangeYaw, to constantly face the enemy. This allows the pawn to always turn with the player as the walking animation occurs. If the conditional test for self.enemy_vis fails, then the order returns itself to the LostTarget order. The next half of the order is a second conditional test, that will determine whether or not the animation has come to it's end. This is how the attack sequence will be timed. Before continuing with a breakdown of the order, it is important to understand how the order is being processed. Since the order is processed while running at a Low Level, it constantly loops at a rate that is defined by the self.ThinkTime setting. This means, that only a portion of the motion will be performed
if(self.animate_at_end =false)	The self.animate_at_end variable, is a read only value that will return true, if the animation has come to its last key-frame. This only works when the SetHoldAtEnd(true) command has been previously called.
walkmove(self.ideal_yaw,50)	The walkmove() command is the basic Low Level method for moving a pawn. It breaks down simply as, move the pawn in the direction defined by the first parameter, a distance that is defined by the second parameter. In this case the pawn is instructed to move 50 texels in the direction defined by self.ideal_yaw. Since the pawn is constantly rotating to face this direction, it will always be forward. What is important to know, is that the pawn will only move the full 50 texels within one second, because this is the length of the order. During one loop of the order, the pawn will only travel 2.5 texels. This is the distance, divided by the looping rate. Later, a conditional test is added to the walkmove command, to avoid obstacles.

FireProjectile("pistolbullet"," joint12",0,-2,0,"health")	This the first of two Low Level versions of the FireProjectile command. It is exactly the same as the High Level version, except there is no definition to play a sound. This command is called at the end of the animation, making the pawn fire at a rate of 1 shot per second.
self.think="Attack"; **return 0;**	Once the animation has ended and one projectile has been fired by the pawn, the looping of this order ceases and the script is redirected to run the order named Attack. The Attack order then checks for an enemy in site and takes the correct action as to either attack again or go to the LostTarget order.

within one loop of the order. What the order needs to do is run for only as long as the animation of the pawn. Then at the end of the animation, the pawn fires a projectile and returns to the Attack order. Since the animation lasts for one second, the order only runs for this long as well.

At this point, when the pawn is alerted to the presence of an enemy, it will attack the enemy by constantly charging and firing. Now the script will be modified, to have the pawn keep a certain distance from the enemy, so as to prevent the pawn from over running the player. Add the following to the Attack order.

```
Attack[()
{
self.ThinkTime=0.05;
self.yaw_speed=160;
UpdateTarget();
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
```

```
        if(self.enemy_range>128)
        {
        SetHoldAtEnd(true);
        Animate("wkattack");
        self.think="Walk";
        return 0;
        }
else
        {
        TC=self.time;
        Animate("stattack");
        self.think="Stand";
        return 0;
        }
}
}]
```

What this does, is add an else condition to the self.enemy_range conditional test. If the enemy range is less then 128 texels, then these commands are processed. The commands are similar to the result of a true condition, but instead of using the animation length to time the Stand order, a variable to keep track of time is used. This is the first line, TC=self.time, which accesses a read only variable that returns the time in seconds that have passed since the start of Low Level processing. Before continuing, add a declaration for the TC variable, between the very first curly bracket of the entire script and the Spawn order. Below is an example.

```
{
TC [0]
Spawn[()
```

```
{
Console(true);
AttributeOrder("health",20,"Die");
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
NewOrder("LostTarget");
}]
```

Now add the Stand order to the script. It is very similar to the Walk order, except for the second conditional test.

```
Stand[()
{
self.ThinkTime=0.05;
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
}
else
{
HighLevel("LostTarget");
return 0;
}
if(self.time>TC+0.5)
{
FireProjectile("pistolbullet","joint12",0,-2,0,"health");
self.think="Attack";
return 0;
```

```
}
}]
```

It seems that the order is exactly the same as the Walk order, except that instead of checking for the end of the animation, a conditional test for the time is evaluated. TC was set to the hold the time just before the order was called. This conditional test, will continue to check and see if 0.5 seconds has elapsed and then run switch to the Attack order, so as to assess the enemy range again. It does this by checking if the self.time variable, is 0.5 seconds greater then TC. Compile the test level and run the order. The pawn will now act in a manner, similar to that when it was a running at a high level. What needs to be added now, is a means of detecting whether or not the pawn has been destroyed during the Low Level processing.

Add the following lines to both the Walk order and the Stand order, right after the line self.ThinkTime=0.05.

```
if(self.health=0)
{
HighLevel("Die");
return 0;
}
```

This is a conditional test for the self.health variable. This is a read only value, that contains the amount of attribute that the pawn still has, after being assigned it by the High Level, AttributeOrder() command. If the value has reached 0, then the script processing is switched to the High Level order named Die. Compile the test level and try the low level

combat routine. Every time the pawn is destroyed, it restarts itself at the Script Point. To make the pawn almost complete, all that must be added is a means of avoiding obstacles.

Low Level Collision Detection

Collision detection at a Low Level is accomplished using the walkmove() command as a conditional test. Add the following to the Walk order. The conditional test for self.health from the last addition has also been made bold.

```
Walk[()
{
self.ThinkTime=0.05;
if(self.health=0)
{
HighLevel("Die");
return 0;
}
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
}
else
{
SetHoldAtEnd(false);
HighLevel("LostTarget");
return 0;
}
if(self.animate_at_end=false)
{
```

```
        if(walkmove(self.ideal_yaw,50)=false)
        {
                if(random(1,10)>5)
                {
                ForceLeft(10);
                }
                else
                {
                ForceRight(10);
                }
        }
}
else
{
SetHoldAtEnd(false);
FireProjectile("pistolbullet","joint12",0,-2,0,"health");
self.think="Attack";
return 0;
}
}]
```

Now walkmove is used as a conditional test, to see if it can travel the distance that is covered, within one loop of the order. If it can, then it returns true. If it can't, then it returns a false. With the result of a false value being returned, a second conditional test is performed. The second test, is for a random number between 1 and 10. If it is less then 5, a ForceRight() command is executed. If it is greater then 5,. then the ForceLeft() command is executed. The parameter for ForceRight and ForceLeft is not a value that represents texels. It instead defines the amount of Force to apply to the pawn. This value usually needs some adjusting, since it

depends on the scale of the pawn.

This level of collision avoidance is about as simple as it gets, but is effective enough to make the pawn go around short objects and avoid stopping at corner. Add a few 32 x 32 x 32 brushes to the test level and watch the pawn bounce its way around them as it travels towards the player. At this point the pawn is ready to be customized for a level. The next chapter will outline adding the robot pawn to the Space Escape level. Here is the complete script.

```
{
TC [0]

Spawn[()
{
Console(true);
AttributeOrder("health",20,"Die");
SetFOV(360);
HostilePlayer(true);
HostileSame(false);
HostileDifferent(false);
SetGroup("Enemy");
NewOrder("LostTarget");
}]

Alert[()
{
AnimateStop("alert",0,"");
LowLevel("Attack");
}]

LostTarget[()
{
PlayAnimation("idle",false,"");
FindTargetOrder(400,"Alert","health");
}]
```

```
Attack[()
{
self.ThinkTime=0.05;
self.yaw_speed=160;
UpdateTarget();
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
        if(self.enemy_range>128)
        {
        SetHoldAtEnd(true);
        Animate("wkattack");
        self.think="Walk";
        return 0;
        }
        else
        {
        TC=self.time;
        Animate("stattack");
        self.think="Stand";
        return 0;
        }
}
}]

Walk[()
{
self.ThinkTime=0.05;
if(self.health=0)
{
HighLevel("Die");
return 0;
}
if(self.enemy_vis=true)
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
```

```
}
else
{
SetHoldAtEnd(false);
HighLevel("LostTarget");
return 0;
}
if(self.animate_at_end=false)
{
	if(walkmove(self.ideal_yaw,50)=false)
	{
		if(random(1,10)>5)
		{
		ForceLeft(10);
		}
		else
		{
		ForceRight(10);
		}
	}
}
else
{
SetHoldAtEnd(false);
FireProjectile("pistolbullet","joint12",0,-2,0,"health");
self.think="Attack";
return 0;
}
}]

Stand[()
{
self.ThinkTime=0.05;
if(self.health=0)
{
HighLevel("Die");
return 0;
}
if(self.enemy_vis=true)
```

```
{
self.ideal_yaw=self.enemy_yaw;
ChangeYaw();
}
else
{
HighLevel("LostTarget");
return 0;
}
if(self.time>TC+1)
{
FireProjectile("pistolbullet","joint12",0,-2,0,"health");
self.think="Attack";
return 0;
}
}]

Die[()
{
AddExplosion("GrenadeExplosion","Root",0,0,0);
        if(random(1,10)>5)
        {
        AnimateStop("die1",0,"");
        }
        else
        {
        AnimateStop("die2",0,"");
        }
FadeOut(2,0);
NewPoint("point01");
RotateToAlign("idle",500,false,"");
TeleportToPoint("point01");
FadeIn(5,255);
AttributeOrder("health",20,"Die");
NewOrder("LostTarget");
}]
}
```

Space Escape - Stage 5

The previous chapters, were devoted to developing the media required for stage 5 of the game design process. This chapter, will discuss stage 5 for the space escape demo. The next steps will add a weapon and the game enemies. Then, after the game play has been tested out, a level puzzle and a HUD will be added.

Adding a Weapon

Start by copying the two actor files named PlasmaRifle.act and bullet1.act, from the media package for this book, to the RF media\actors folder. With the actors in place, RF must now be configured to use this weapon. Run NotePad and open the player.ini file. Add the following entries and then close the file.

[PlasmaRifle]
initial=1
low=0
high=1

[PlasmaBullet]
initial=100
low=0
high=100

This starts off the player with 1 Plasma Rifle and 100 Plasma Bullets. Now these will be defined in the weapon.ini file. Edit the weapon.ini file and add the following entries.

[PlasmaBullet]

type = projectile
actor = bullet1.act
rotation = 0 0 0
scale = 1
gravity = false
bounce = false
speed = 1000
lifetime = 1
boundingbox = 0.1
explosion = CorExpl
actorexplosion = BotExpl
damage = 5
altdamage = 1
explosionradius = 0
explosiondamage = 1
decal = 0

[PlasmaRifle]

type = weapon
slot = 0
firerate = 0.2
catagory = projectile
projectile = PlasmaBullet
attribute = health
altattribute = health
ammunition = PlasmaBullet
ammopershot = 5

viewactor = PlasmaRifle.act
viewfillcolor = 128 128 128
viewambientcolor = 32 32 32
viewrotation = -1.40 90 0
viewoffset = 1.20 -3.70 -2.60
viewscale = 1
viewlaunchbone=joint2
viewanimspeed = 1
viewidleanim = gun2idle
viewattackanim = gun2shoot
viewwalkanim = gun2walk

Now delete all the other entries from the weapon.ini file, so that only these two remain. This will be the final weapon definition for the Space Escape demo. The first entry, defines the projectile for the Plasma Rifle. It is a visible actor, that has two effects assigned to it. These are defined with the entries, explosion and actorexplosion. They will be defined next.

The second entry defines the Plasma Rifle and makes it the default weapon for the level. When the level starts, the player will automatically have the plasma rifle present. This is done by removing the entry named Default and setting the slot entry of the Plasma Rifle to 0. This only covers the first person data for the weapon, since the demo will be fixed to this view.

Before defining the effects for the projectile, edit the playersetup.ini file and find the attributename entry under the Misc heading. Change the following entries.

attributename = PlasmaBullet

recoverytime = 0.1

This feature makes the Plasma bullet entry always restore itself. The effect is a Plasma RIfle that can fire 10 rapid shots and then slowly charge back up to full capacity. Save the file and open the effects.ini file. Add the following.

[YellowLight]

type = light

colormax = 237 230 37

colormin = 237 40 37

radiusmax = 150

radiusmin = 100

intensity = 1

totallife = 0.2

[Corona]

type = corona

fadetime = 0.15

minradius = 0.5

maxradius = 2

maxvisibledistance = 1000

maxradiusdistance = 2

minradiusdistance = 1

color = 237 40 37

totallife = 0.2

[BotSpray]

type = spray

bitmapname = flame03.bmp

alphamapname = a_flame.bmp

angles = 0 90 0
colormax = 0 0 255
colormaxalpha = 255
colormin = 0 0 100
colorminalpha = 200
sourcevariance = 2
destvariance = 2
gravity = 0 0 -45
maxscale = 0.25
minscale = 0.15
maxspeed = 300
minspeed = 100
maxunitlife = 0.3
minunitlife = 0.2
particlecreationrate = 0.01
totallife = 0.3
bounce = false

[BlueCorona]
type = corona
fadetime = 0.15
minradius = 0.5
maxradius = 2
maxvisibledistance = 1000
maxradiusdistance = 2
minradiusdistance = 1
color = 200 200 255
totallife = 0.2

Note that the effect YellowLight, is already defined in RF07 and may not need to be added. These four effect entries will make up the explosion that are defined for the weapon. One effect is a Light type, two are Corona effects

and another is a Spray effect. The only entry that uses media is BotSpray, which requires flame03.bmp and a_flame.bmp. If these are not already located in the RF media folder, then copy them from the media package for the book. Save and exit the effect.ini file and then edit the explosion.ini file. Add the following.

[CorExpl]
effect0 = Corona
delay0 = 0
offset0 = 0 0 0
effect1 = YellowLight
delay1 = 0
offset1 = 0 0 0

[BotExpl]
effect0 = BlueCorona
delay0 = 0
offset0 = 0 0 0
effect1 = BotSpray
delay1 = 0
offset1 = 0 0 0
effect2 = YellowLight
delay2 = 0
offset2 = 0 0 0

These define the explosions, from the predefined effects added to the effect.ini. The weapon is now ready for testing. Compile and preview the level. It should start with the Plasma Rifle ready and loaded. Fire at the walls and get a feel for how the weapon fires and charges. A red colored corona effect, should be seen when a projectile hits any level

geometry. When level actors are hit, a blue colored corona with a spray will occur. In the next step, enemy pawns will be added to the level to create a combat experience.

Adding Enemy Pawns

Before adding any type of enemy that is capable of killing the player, it is necessary to modify the amount of health the player starts with and the defined death animation. Start by editing the player.ini file and find the heading named health. Change it to the following.

[health]
initial=1000
low=0
high=1000

This starts the player off with enough health to clear the level. If it is still not enough health, then bring it up to 100000. The next step, is to remove the death animation defined in the playersetup.ini file. This cancels the RF third person death sequence, which is used to cinematic scenes when the player is killed. Edit the playersetup.ini file and find the following entries.

die = Die Die1
injury = Injury

Use a semi-colon, to comment them out like so.

;die = Die Die1
;injury = Injury

What to Read

Be sure to read the following from the RF off-line docs.

- Weapon Definition
- Predefined Effects
- Predefined Explosions
- Pawn
- Player Data Definition

Save and exit the files. RF is now ready to test out the combat system for the game. All that is required now, is to add the enemies to the level.

To make the pawn available to RF, it must be defined in the pawn.ini file. If this has not been done already then add the following to the pawn.ini file. If Robot is already defined, then just change the heading,

[Android]
actorname = android1.act
actorrotation = 0 180 0
actorscale = 0.6
fillcolor = 128 128 128
ambientcolor = 128 128 128
subjecttogravity = true
boundingboxanimation = idle
shadowsize = 64

Save the file and then copy the android1.act file from the media package into the RF media\actors folder. Also, copy the robot.s script from the media package, into the RF Scripts folder. This script is similar to the one in the previous chapter except for a few changes. First change the Console command in the Spawn Order to false as so. Console(false); Then modify the Die order as follows.

```
Die[()
{
AddExplosion("GrenadeExplosion","Root",0,0,0);
        if(random(1,10)>5)
        {
        AnimateStop("die1",0,"");
```

```
        }
        else
        {
        AnimateStop("die2",0,"");
        }
FadeOut(2,0);
Remove(true);
}]
```

This modification does not re-spawn the pawn at a script point and instead, just removes it from the level. Now find the two instances of the FireProjectile command and change the first value to be PlasmaBullet instead of pistolbullet.

```
FireProjectile("PlasmaBullet","HandR",0,0,0,"health");
```

This will make the pawn fire the same projectile as the player, so it can be visible when the player is fired at. Be sure to only change the first parameter and not any others.

PawnType - Android
ScriptName - robot.s
SpawnOrder - Spawn

The proper environment for the pawns has now been established and the Pawn Entities an be added to the level. The side bar shows the pawns added to the level and the direction that they are set to face. Below that, is the properties for the pawn entity. In the World Editor, add a single pawn to the level and then copy / paste the rest into position. Be sure to position the entities, right against the floor of the level.

The previous chapter that described the script, was were all the had work took place. With a script already written for them, all that was needed to populate the level, was to copy and paste as many as were required. The level can now

be compiled and previewed and the combat system tested out. The pawns will charge at the player until within a certain distance and fire a shot every second or so. They will require 4 shots to be killed, since they are given an attribute of 20 in the script and the weapon removes 5 points per shot.

This completes two of the three tasks, that make up stage 5 of the design process. Since there is now a weapon and enemies added to the level, everything is in place to add a working Heads Up Display or HUD. The next chapter is devoted to making HUD graphics and adding them to RF. This will be the final step, in stage 5 of the Space Escape development process.

Heads Up Display

RF has a built in system for a Heads Up Display or a HUD. This is the 2D graphics, that makes up the display of the attributes that the player has. It also incorporates a built Radar and Compass display. The RF HUD, is defined through the hud.ini file. This chapter will discuss the RF HUD system and then build the HUD for Space Escape.

HUD Coordinates

The RF HUD definitions, will always require a X and Y coordinate in order to position the HUD element. Since RF can be set to many resolutions, the HUD system is designed to accommodate for this and expand to meet any screen size. On the next page is a diagram that shows how the HUD is broken up into four sections, that have positive and negative values. The coordinates that are marked in the diagram, represent the X and Y coordinates of a HUD element that would go there.

For example, to position a 256 x 256 image as a HUD element in the Top Left of the screen, it would require the coordinates 0,0. Placed at the Top Right of the screen, it would require the coordinates to be -256,0. An image that is 128 x 128 placed in the Bottom Right, would require the coordinates to be -128,-128. HUD elements can be placed in

the exact center by using the value, center.

HUD Graphics

RF HUD graphics, require an image to be used as the HUD element, and another image which defines the transparent portions of the graphic. The graphic which defines the transparent portions is termed the Alpha Map. The side bar contains two graphics, which can be used to display weapons ammo for an MP5. The top image is of the MP5 and the bottom image is the alpha map. Areas that are black (RGB 0,0,0) will be totally transparent and areas of the alpha map that are white, (RGB 255,255,255) will be completely

visible. Any value in between, is a semi-transparent value. For example, if the white areas were made to be grey, (RGB 128,128,128) then the MP5 HUD element, would be semi-transparent when displayed in the game.

HUD Types

RF has five HUD types that can be defined in the HUD.ini file. They include a Radar, a compass, a vertical type, a horizontal type and a numeric display. Below is a break down of each type of HUD display.

- **Radar** - The radar element for RF works by displaying either a pawn or a static entity indicator, within a circular area. A unique indicator graphic, can defined for pawn and static entities.
- **Compass** - RF also has a feature that can display a compass as a HUD element. The element will rotate left and right, in correlation with the direction that the player is facing.
- **Vertical and Horizontal** - HUD elements for all other attributes, are one of three types. Vertical, horizontal or numeric. Vertical or horizontal types, are used to display a graphic as the indicator for the attribute. If the the attribute uses the vertical type, then the graphic will fade downwards as the attribute decreases. A horizontal type, will have the attribute fade from left to right.
- **Numeric** - The simplest display for a HUD element is a numeric display. This simply overlays a numeric value representing the attribute, over a defined graphic.

The Space Escape HUD

The following example will detail the making of the HUD for Space Escape. It uses Adobe Photoshop 6 for the example but almost any paint program can be used to follow the necessary steps. The first HUD element will be used to display the energy contained by the weapon. Any files made here must be saved in the RF media\bitmaps\hud folder.

- **Step 1** - Create a new RGB image with a width of 256 and a height of 64. Select the option to display the Grid over the image, the Grid should be sized to 32 x 32 with 4 sub units. Be sure that, Snap to Grid is selected.
- **Step 2** - Paint the Background black (RGB 0,0,0) and then add a new Layer. Name the new Layer, Shell. Select the new Layer.
- **Step 3** - Change the foreground color to RGB 24,93,116 and paint the layer with that color.
- **Step 4** - Use the Select Tool to select a rectangle from 16,16 to 240,48 and Cut the area from the layer.
- **Step 5** - With the new Layer still selected, sect the Layers menu and the choose Layer Style - Bevel and Emboss. Click OK and use the default settings.
- **Step 6** - Click the Select Color Tool and select the area that had been cut out. The black area should become selected. Add a new Layer and name it, Indicate. Select the new Layer. Do not deselect the black area.
- **Step 7** - Change the Foreground Color to RGB 166,43,24 and paint in the selected area. This red colored area will serve as the indicator. Still, do not deselect the selected area that is now painted red.
- **Step 8** - Create a new Layer and name it Alpha1. Change

the Foreground Color to RGB 255,255,255 and set the Background Color to 0,0,0.

- **Step 9** - Paint the selected area white and then under the Select Menu choose, Inverse. Paint the inverted selection black. Still, do not deselect the inverted selection area.
- **Step 10** - Create a new layer and name it Alpha2. Select this layer and paint the selected area white. Now inverse the selection again, and paint the inverted area black.

This creates four layers that will each be a graphic that is used in the HUD. These layers must now we saved into individual images, that will make up the HUD element.

- **Step 1** - With the layer Alpha2 visible, save the image in the RF media\bitmaps\hud folder under the name, gun_shell_alpha.bmp.
- **Step 2** - Make the Alpha2 layer invisible so that Alpha1 can be seen and then save the file as, gun_indicate_alpha.bmp.
- **Step 3** - Now make layer Alpha1 invisible and the layer named shell invisible. The only layers which should be visible is the Background and the layer named indicate. Save this image under the name, gun_indicate.bmp.
- **Step 4** - Now make the shell layer visible again and make the indicate layer invisible, save the resulting image as gun_shell.bmp.

The side bar shows the images used for the Frame and the Frame alpha map. Once again, these must be placed in the RF media\bitmaps\hud folder.

gun_shell.bmp

gun_shell_alpha.bmp

Defining the HUD Element

Before continuing, open the escape.3dt file in the World Editor and edit the properties of the playersetup entity. Change the HudInfoFile property, to spacehud.ini. Compile the level with the new setting. To test the level, just run it by making it the default loading level for RF.

Open the RF install folder and edit the hud.ini file. Save it under the name spacehud.ini. Delete every entry from the newly saved file and add the following new entry.

[PlasmaBullet]
type = horizontal
frame = hud\gun_shell.bmp
framealpha = hud\gun_shell_alpha.bmp
framex = 0
framey = 0
indicator = hud\gun_indicate.bmp
indicatoralpha = hud\gun_indicate_alpha.bmp
indicatoroffsetx = 0
indicatoroffsety = 0
indicatorwidth = 256

type = horizontal	This sets the element to be the Horizontal type.
frame = hud\gun_shell.bmp	This defines the gun_shell.bmp graphic to be the frame image for the element.
frame = hud\gun_shell_alpha.bmp	This defines the gun_shell_alpha.bmp graphic, to be the alpha map for the frame image of the element. The Frame graphic will always be visible even if there is no more ammo.

framex = 0 framey = 0	These two lines define the X and Y coordinates for the frame image. In this case, the image is positioned in the Top / Left corner of the screen.
indicator = hud\gun_indicate.bmp indicatoralpha = hud\gun_indicate_alpha.bmp	The indicator image and the alpha used for it, are defined using the indicator and indicator alpha setting. They are defined just like the frame images.
indicatoroffsetx = 0 indicatoroffsety = 0	The indicator may not always be the same size of the frame image and in that case, the offset from the top left of the frame coordinates, is defined using these two values. In this case however, the indicator image is the same size as the frame image, and therefore requires no offset.
indicatorwidth = 256	Since the indicator is of the horizontal type, then the engine will need to know the width of the image, so that it can be divided into portions, that represent one unit out of the of the maximum attribute amount.

What this does, is define a HUD element for the attribute named PlasmaBullet, which is defined in the player.ini file. This creates a visual representation, of the amount of PlasmaBullet Attribute left. See the break down of the definition for an explanation of each entry.

Run RF and test out the HUD element. As shots from the Plasma Rifle are fired, the indicator should drop. When shots are not being fired, then the indicator will gradually climb up again to full capacity. Now a second HUD element will be added, to display the remaining health attribute that the player has. This will be a Numeric type display and will only require a frame image. The side bar contains two images

which are used as the frame for the health attribute HUD element. Add the following element entry to the spacehud.ini file.

[health]
type = numeric
frame = hud\health_shell.bmp
framealpha = hud\health_shell_alpha.bmp
framex = -64
framey = 0
indicatoroffsetx = 20
indicatoroffsety = 40
font = 9
width = 3

The only real differences between this definition and the other, is that it is a Numeric type, which removes the need for an indicator image and a font is defined instead. In this case, one of the default fonts is used. Fonts are defined in the menu.ini file. Save the spacehud.ini file and run RF. The health indicator should appear in the Top / Right corner of the screen and display the amount of health the player has.

This finishes stage 5, of the development process for the Space Escape demo. The level should be playable and almost complete. In the next chapters, a puzzle will be added to the level and then the menu will be customized. Then, a bit more detail will be added and finally the demo is packaged for distribution and testing.

Adding a Puzzle

Space escape has one simple puzzle added to it. It is a simple, Locked Door puzzle. The player must release the Air Lock Control, in order to enter the Air Lock. This forces the player to go through the Hallway and into the Control Room, where the Air Lock Control is located. This chapter will describe how this puzzle is added to the level and also add the ability to win the level and exit it.

Puzzle Components

Although this is a very simple puzzle, it will require a couple of visual enhancements to let the player know that the door is locked and that it has been unlocked. This will include an extra Message Entity for the door and a Wall Decal Entity for the Air Lock Release. Before beginning, copy the file named airlock.bmp from the media package for this book, into the RF media\bitmaps folder. This image will be required for the Wall Decal.

On the next page is a diagram that illustrates the puzzle and how each element is related to each other. It starts with the last set of Double Doors, that leads to the air lock. These will be the locked doors. Examine the diagram but before adding any entities, first build the World Model which will act as the Air Lock Release. Start by selecting the texture

Puzzle Components

named Com_03 and create a box with the dimensions 16 X - 64 Z - 96 Y. See the side bar for a reference to the position. The illustration below it, shows the distance of the Wall Decal from the floor.

With the brush created, select the front face of the brush in the camera window (be sure to refresh the camera window first) and texture it with Com_Tow_01. Set the Y scale to 0.75 and the X scale to 0.5. Align the texture, to match up with the face. See the screenshot, for how the tower should look in the level. With the model textured and selected, select Model from the Models Tab. Name the model, AirLock and click OK. This model will not require any animation.

Two entities will now be added to the level. Use the Template Tab to add a Trigger Entity and a Logic Gate Entity. The Logic Gate will be a type 5 gate, which acts as a Latch. What this does, is detect when the trigger goes from low to high and then lock itself in the high state. Other wise, the Trigger would go off before the player reached the door. By using a Latch, the logic gate always stays high after it has been activated by the trigger. Set the Trigger and Logic Gate Entities to have the following properties.

Trigger

- **Model** - AirLock
- **szEntityName** - airtrig
- **PlayerOnly** - true

LogicGate

- **szEntityName** - lgairtrig

- **Trigger1Name** - airtrig
- **Type** - 5

To use these as a Lock Release for the double doors to the airlock, set the TriggerName property to be lgairtrig. This is the szEntityName property of the Logic Gate. Compile and preview the level. The doors to the air lock should not open, until the player has made contact with the World Model in the Control Room. What this needs now, is a few visual cues to help guide the player.

Visual Aids

Four entities will be used to give the player, an indication of the state of the puzzle. These include a Wall Decal, a Dynamic Light and a Message / Logic Gate combination. Add these four entities to the level and set the properties as shown below. Position the Wall Decal as shown in the side bar, on the previous page.

Wall Decal

- **TriggerName** - lgairtrig
- **AlphaName** - airlock.bmp
- **BmpName** - airlock.bmp

Dynamic Light

- **TriggerName** - lgairtrig
- **MaxRadius** - 220
- **MinRadius** - 130
- **Color** - 255,0,0
- **Time** - 1.0

Message

- **TriggerName** - lgairdoor
- **Type** - Static
- **Text** - AirLock

LogicGate

- **szEntityName** - lgairdoor
- **Trigger1Name** - airdoor
- **Type** - 6

Although the Wall Decal does not need to be rotated for this example, it is important to not rotate the entity using the mouse and only use it by specifying the angle entry. Position the Wall Decal to point at the Air Lock Control world model and position the Dynamic light just behind the light in the Air Lock Room. In order to make the double doors work with the Message Entity, one of them must be given an szEntityName. Select one of the entities for the two doors and set the szEntityName property to, airdoor. Now the message defined in the Message Entity must be added to the message.txt file.

[AirLock]
Release the Air Lock

It is important to remember, that when the message.txt file has entries added to it, some extra carriage return lines, should also be added to the end of the file. This is to make sure, that RF reads to the very end of the file. Otherwise it may not load the message and nothing will be displayed when the player contacts the locked door.

With these four entities added and their properties configured, the complete puzzle system can be tested. If the player first tries to enter the Air Lock without releasing the lock first, they will be presented with the message, *Release the Air Lock*. When the player makes contact with the model in the Control Room, it will then activate the doors, the Wall Decal and the Dynamic Light.

Ending the Level

The level ends, when the player reaches the Air Lock. This is easily accomplished by adding a world model and a ChangeLevel Entity. This entity, is used to end the current level and switch to a new specified level. A level, can have as many ChangeLevel entities that are required, for the specific game scenario. To add the model that acts as the Air Lock, create a brush with the dimensions 64 X - 16 Z - 128 Y. Position it as shown in the side bar. Select the newly created brush and then from the Models Tab, select Model. Name the model, **exit**. Now with the Template Tab, add a ChangeLevel Entity and only change the model parameter to specify the **exit** model. Compile and preview the level. When the player makes it to the Air Lock and comes in contact with the **exit** model, the ChangeLevel entity will return the game to the Main menu. It does this, because no new level is specified in the szNewLevel property. By default, the entity returns to the main menu. One thing to keep in mind about the ChangeLevel Entity, is that the model used with it, must be solid and not transparent or empty.

exit model

The RF Menu

Reality Factory has a configurable menu system, that allows custom menus to be easily generated from defined images. Since the release of RF07, the menu system can display any number of image types, including animated GIF images and JPG images. This chapter examines the structure of the RF menu system, discussing the differences between the old menu and the new menu.

RF Menu Structure

All of the RF menu screens are predefined in the code. This means that RF always starts with a Main Menu, that branches off into specific sub-menus. New sub-menus can not be defined without editing the source code, but not all the sub-menus need to be defined in the menu.ini file. On the next page, is a diagram which details the structure of the RF menu. Each button made available in the Main Menu, will allow access to the sub-menu, that the button is associated with. Buttons in these sub-menus, then allow access to further sub-menus.

All that can really be defined by the user when creating a menu, are the graphics that will be used to make up these predefined menus and the buttons that access them. These are defined in the menu.ini file and the graphic

The RF Menu Structure

images themselves, are placed in the media\bitmaps\menu folder. This is the default root directory for all menu media, but can be defined in the RealityFactory.ini file.

Examine the diagram of the RF menu tree on the previous page. This shows the buttons which are found under each menu, for a total of nine menu screens. Each menu screen can have a unique Background Image and Title Image, along with the buttons. By not defining a button within a specific menu, it cuts off access to that sub-menu. For example, if the Credits button was not defined in the Main Menu, then the player would not be able to access the Credits Menu. If no Exit button was defined, then the player would never be able to exit the game. Be sure to always define the Exit button and the Go Back button, for each menu screen.

The Menu.ini File

Everything having to do with the menu, is defined within the menu.ini file. This is a well commented file, which describes each entry as it is defined. These comments are the best guide for editing the menu.ini file, as they describe the syntax required for each definition. The order of these definitions is also very important. The menu first requires the images and fonts to be defined and then these defined graphics are used to assemble the menu. Open the menu.ini file in NotePad and take a quick look at the definitions. It may seem confusing at first, but it can be broken down into different sections, that make it easier to understand. Next, is a breakdown of the required types of definitions and how they are used by the menu.

- **Design Size** - This is the first and most critical setting. It defines the resolution of the menu screen graphic coordinates. This does not effect the screen resolution. It only defines the size of the coordinate grid, that will be used to position the graphics that make up the menu. The default is 800 x 600, to match up with the default 800 x 600 screen resolution set in RealityFactory.ini.
- **Background Images** - Each image that will be used as a background must be defined with a number. The menu.ini file shows the syntax to be, *background=background number background image*. The first image defined in the default menu.ini is, *background=1 backimage1.png*. This sets background image number 1, to be backimage1.png Later, when this image needs to be referenced as a background, it will be referred to as background number 1.
- **Images** - These are the images that are used to make up the buttons, scroll bars, check boxes and sliders. The syntax defines an image number, the image to be used and then the alpha map for the image. The components are then taken from these images, by defining the areas that contain the components. This is explained more thoroughly later on.
- **Title Images** - Each menu screen can have a unique Title Image displayed along with it. These are defined in the menu.ini file in the same manner as images, using the Title Image number, the image and then the alpha map.
- **Animations** - Instead of static images such as the ones defined above, the RF menu can use animated GIF images in their place. It is important to remember, that GIF images are not placed in the bitmaps folder, but are instead stored within the RF media\video\menu folder. A common error, is to store animations in the bitmaps folder.

In the new RF07 default menu, all of the images used for buttons are animated GIF files.

- **Fonts** - All fonts used in RF by the HUD or Menu, are defined in the menu.ini file. Fonts are added to RF, using the Font Tool that comes with RF. The resulting DAT file is then defined in the menu.ini file. What is important to remember, is that font numbers referred to by the HUD or a level entity, always indicate one less then the defined number in the ini file. For example, Font 9 in the HUD, would be referred to as font 8.
- **Menu Titles** - The title images that are defined earlier in the file, were only to specify the image to be used. The actual placement of the image on screen, is defined as a menutitle. Before continuing with assembling a menu, the menutitle definition in the default RF menu.ini, can serve to show the difference between the oldmenu.ini and the menu.ini files.

Old Menu vs New Menu

Before RF07 the menu that RF used was made up of static images. As of RF07, the menu was changed to demonstrate the use of animated GIF images. The first defined menutitle in the RF07 menu, is an example of a defined animation used for a title. The rest of the definitions, are for static images. Below is the first menutitle defined the the RF menu.ini file.

menutitle=0 0 480 161 0 0 0 0 10

The syntax asks for 9 values to define the title. The first value, is the number of this menutitle. This is used as a

reference, when the menu screens are defined. The second value, defines the title image to be used. In this case though, there is no title image number 0. This makes the menu load the animation instead. The next two values, 480 and 161, are the X and Y coordinates of the image on the screen. The two values that follow, are the width and height. They are set to 0, because the image is using a single animation, instead of a static image taken from a larger bitmap. The next two values are used with the width and height parameters, to define the top / left corner, of the area of the smaller image. The final value is the animation number to use. In this case, it is animation number 10.

The next entry does not use an animation, but instead uses a static image. Below is the second menutitle, defined in the RF07 menu.ini file.

menutitle=1 1 480 161 152 100 0 0 -1

In this definition, title image number 1 is used as the source image. It is given the same X and Y coordinates, but also defines the height (152) and width (100) of the title image. This is in case the image is made from an area of a larger image. In this case however, it is not, so the next two values are 0,0. These are the top / left coordinates of the area, that the image would be taken from. Once again however, this definition uses a single image and does not take a smaller image from a larger one. The best way to understand this is to take a look at the image used to make the buttons for the RF old menu. Open the file, Images.bmp in the media\bitmaps\menu folder. This is an image, that contains all of the buttons used in the RF menu, previous to RF07. The old menu, divided up this single image to create

each button. The new menu however, uses a single image for each button. These are the GIF images, stored in the RF media\video\menu folder.

Defining Menu Screens

This is where all of the previously defined images, are now brought together to define each individual menu screen. The first menu screen, is the Main menu and unlike the others, it has a unique entry which defines the background image number and the title menu to be used. This is defined by the entry, *main=0 1*. Next comes the individual buttons for each specific menu. As stated previously, any button that is not defined does not appear in the menu. This means that the player will have no way of accessing that menu. A menu can be kept down to a bare minimum by only defining a few buttons that allow the game to start, exit and be configured. The Space Escape demo uses a minimum menu and is examined later in the chapter.

The syntax for defining a button, is similar to the syntax for defining a menu title. Below is the definition for the start game button.

newgame=26 158 0 95 57 0 0 0 0 0 1

The syntax demands the X and Y coordinates of the image on screen, followed by the image number itself. Since an animation is used instead of an image, the defined image is 0, which does not exist. The next values define the width and height of the image. This is required for the menu to know the mouse over area. The next settings are all 0,

because the image used is an animation and not a static image, taken from a larger image. All that is then required, is the final two values, which specify the animation number and the animation to be used as a mouse over. In this case, it is animation 0 and 1.

Buttons that lead to another menu require two more values. They require the background image number and the menu title to be used for the next screen. Below is the definition for the button that leads to the Credits menu.

credits=26 400 0 95 57 0 0 0 0 2 1 11 12

Before the two animations are defined, the background number and the title number for the credits menu are defined. In this case, it is background image number 1 and menu title 11.

Difficulty and Character Selection

Two more menus outside of the main menu, are available to players if required. These include the Difficulty Setting Menu and the Character Selection menu. These appear after New Game is selected from the Main menu. They are also defined within the menu.ini file, but will only be available, if activated using the RF INI Editor. Although these will not be discussed here, what is important to remember, is that they can not be removed from the menu.ini file or RF will crash. They must be defined and not commented out, in an attempt to save on the distribution of media. The same goes for the Video Options Menu and the Advanced Control menu.

The Space Escape Menu

The menu used for the Space Escape demo is a scaled down version of the RF menu, that uses different images for the menu titles and the backgrounds. None of the positions are altered and the same menu buttons are used. The only modification to the menu.ini file, is the commenting out of the entries which define the Save Game menu, the Load Game Menu and a few of the options. The following, shows the changes made to the different sections of the menu.ini file. The new file is saved as spacemenu.ini and is then set as the default menu, using the RF INI Editor.

The first changes are to the background images. Three different background images are set to be used with this menu and the others are commented out. Notice that the images used are JPGs and not PNGs for this menu.

```
;-----------------------------------------
; define the background bitmaps
; format : background# bitmap
;-----------------------------------------
background=1 spacemain1.jpg
background=2 spaceblank.jpg
background=3 spacecredit.jpg
; background=4 backload.png
etc..
```

The only changes to the images defined, is to the first image which is displayed with the Credits menu. By defining the same image as the alpha map, the image is displayed with no transparent regions.

```
;-----------------------------------------
; define the images bitmaps
; format : image# bitmap alpha_bitmap
;-----------------------------------------
images=6 spacecredit.jpg spacecredit.jpg
images=7 bartop.png bartop.png
etc..
```

No menu titles are displayed in the Space Escape demo menu screens, so in order to eliminate them, a totally transparent image is used instead. Each button definition, will then require the Menu Title value, to be the first Menu Title defined. In order to save media space, all the other title images are commented out or deleted.

```
;-----------------------------------------
; define the titles bitmaps
; format : titleimage# bitmap alpha_bitmap
;-----------------------------------------
titles=1 blank.bmp a_blank.bmp
; titles=2 optionstitle.jpg optionstitle.jpg
; titles=3 audiotitle.jpg audiotitle.jpg
etc...
```

All the animations are used as buttons, except for animation number 10, which is the Main Menu Title. It can be commented out to save media space. Now changes can be made to remove the defined Menu Titles and only use the new transparent menu title, that will not be seen.

```
;-----------------------------------------
; define the menu titles graphics
```

; format : title# titleimage# screen_X screen_Y width height image_X image_Y animation#

;----------

; menutitle=0 0 480 161 0 0 0 0 10

menutitle=1 1 0 0 32 32 0 0 -1

; menutitle=2 2 480 161 152 100 0 0 -1

; menutitle=3 3 480 161 152 100 0 0 -1

etc..

The modifications made to the Main Menu definitions, alter the background image, the menutitle for the options menu and the credits menu. It also removes the unused buttons, so in order to keep the buttons together, the Y values are also changed.

;----------

; Main Menu

;----------

main=1 1

newgame=26 158 0 95 57 0 0 0 0 0 1

; multiplayer=516 306 0 95 57 0 0 0 0 2 1 45 46

; loadgame=26 220 0 95 57 0 0 0 0 2 1 2 3

; savegame=26 280 0 95 57 0 0 0 0 2 1 4 5

options=26 220 0 95 57 0 0 0 0 2 1 6 7

credits=26 280 0 95 57 0 0 0 0 2 1 11 12

quitmain=26 340 0 95 57 0 0 0 0 8 9

returntext=318 575 FONT9

; mods=516 376 0 95 57 0 0 0 0 2 14 51 52

With the multiplayer, load game and savegame options commented out, their corresponding sub-menus can also be removed. This will help to save media space later on

when the package is distributed. All that's required to complete the new menu is to make a few changes to the definition for the Options menu and the Controls menu.

```
;-------------------
; Options Menu
;-------------------
audio=26 158 0 95 57 0 0 0 0 2 1 16 17
; video=26 224 0 95 57 0 0 0 0 2 1 18 19
control=26 224 0 95 57 0 0 0 0 2 1 20 21
; debug=26 352 0 95 57 0 0 0 0 2 11 22 23
quitoptions=26 288 0 95 57 0 0 0 0 13 14
```

For the Control menu, all that needs to be changed is the advance setting, which is just commented out.

```
;-------------------
; Control Menu
;-------------------
; advance=26 158 0 95 57 0 0 0 0 2 6 27 28
```

Although it seemed a bit of a task, that was about as minimum a modification, that can be made to the menu. All that's needed to use this menu, is to define it as the default menu, using the RF INI Editor.

Space Escape - Finishing Touches

The Space Escape demo is finally ready to have the finishing touches added and then it can be distributed. Right now, it is still in the development stage and needs to be made suitable for a player. This will include adjusting the lighting, performing a final texture pass, adding a bit of level detail and then adjusting the player attributes.

Final Lighting

The final release of the Space escape demo does not have colored lighting and instead uses a RGB value of 255, 255, 255. This allows the colors of the textures, to become clear in the game. The only light that is kept at RGB 255, 0, 0, is the Dynamic Light Entity.

All of the lights can be modified at once, by selecting one of the light entities and then while holding down shift, selecting the rest of them one at a time. If a brush or a different entity is selected, then simply click on it again, in order to deselect it. Once all of the light entities have been selected, open the Entity Properties dialog box and then change the Color property to RGB 255, 255, 255. This will simultaneously change the Color property for all of the light entities. This could have easily been performed one at a time as well, since the level only has a total of 11 lights.

Since the level was very small, having less then 100 brushes, the Light Scale for the level was reduced from 2 to 1. This helped to create tighter shadows and a more realistic lighting effect. From the File menu, select Properties and change the Light Scale entry to 1.

Adding Level Detail

The esthetic detail added to the Space Escape demo, is very minor and was left to the final stage of development. This detail included some boxes in the storage room, a computer panel in the Control room, some ceiling to floor cables and a different computer panel in the Air Lock room. In this case, the only brushes that were flagged as detail, were the crates in the store room and the cables in the Air Lock room.

Brushes that do not block the players view entirely, such as the cables, should be flagged as a Detail brush. This helps to reduce compiling time and remove the brush from specific engine processes, that can reduce frame rate. Brushes are flagged as detail, using the Brush Properties dialog.

One thing to remember about using detail brushes, is that they should always be positioned at least 1 texel away from a brush that is not flagged as detail. Practically, this is not always possible, but if not done, then the engine may draw extra polys that can again reduce the frame rate. It can also cause odd shadow mapping, depending on the light scale of the level.

Final Texture Pass

With very large levels, this can become one of the more tedious parts of the development process. This is another reason why it is good to start with a set of base textures when building the level. The final texture pass for the Space Escape demo, only requires the outer faces of the ship to be textured with Wall_Base_01 and the scale set to 1. These are the faces that are seen through the windows of the Hallway. Use the Camera view port, to select the three faces which make up one of the visible walls and then apply and adjust the scale simultaneously.

Player Attributes and Scripts

During the testing process, the players health attribute was set to a value that would prevent the player from being killed. It is now time to reduce that value to the regular playing amount and test the death of the player and the difficulty of the level. Edit the player.ini and change the health entry to the following.

[health]
initial=50
low=0
high=50

This sets the initial health for the player to be at 50. The player can take 10 hits from the robots before being killed. As for the robots, the script controlling them still has the setting Console(true) applied. This makes the debugging text,

be displayed in the top left corner. Remove this by setting the command to Console(false). This command, was the first command in the Spawn order.

Sound Effects

Sound and music, can really make the difference in a game. Without them, the game just seems like a series of moving pictures. For the Space Escape demo, things were kept to a minimum and the only sounds that were added, were a background song and weapons effects.

Music which is to be played in a level, is specified within the PlayerStart entity. RF can play a CD track, a MIDI file or a Wave file during run time. In this case, a Wave file is specified. Copy the file named spacemusic.wav, into the RF media\audio folder and change the following properties in the PlayerStart Entity.

- **bSoundtrackLoops** - true
- **szStreamingAudio** - spacemusic.wav

Compile and preview the level. The short wave file will continuously repeat itself, as soon as the level is loaded and continue to play until the completion of the level.

Two sound effects, were also added to the Plasma Rifle and the projectile it fires. Copy the files named zapper2.wav and raygun.wav, from the media package for the book, to the RF media\audio folder. First the sound that is played when the weapon fires is added to the weapon.ini file. Edit the weapon.ini file and add the following to the

PlasmaRifle entry.

[PlasmaRifle]
viewanimspeed = 1
viewidleanim = gun2idle
viewattackanim = gun2shoot
viewwalkanim = gun2walk
attacksound = ZAPPER2.WAV

This defines the sound file zapper2.wav to be played when ever the Plasma Rifle fires. Now, a second sound effect will be added to create an impact sound. Add the following to the PlasmaBullet entry.

[PlasmaBullet]
altdamage = 1
explosionradius = 0
explosiondamage = 1
decal = 0
impactsound = RAYGUN.WAV

With this addition, the Plasma Rifle now makes a sound when ever it is fired and when it impacts an object as well. Save the weapon.ini file and preview the level. The sound effects and music, do much to change the atmosphere of the game.

With these finishing touches added to the level, the Space escape demo is now complete. It is ready to go on to the final stage of development for the demo, the packaging and distribution stage.

Packaging a Demo

Before the release of RF07, demos made with Reality Factory, were compiled using an application called Game Builder. However, very many enhancements were made to RF, that Game Builder was not capable of recognizing. Game Builder also had some bugs, that made using it dangerous. Since the release of RF07, Game Builder has been discontinued and was replaced with RF07MIN.zip. This is a copy of RF, that only contains the necessary media that RF needs to run. This is used as a base, to build up a demo from the media used in the custom level. A similar method is outlined in this chapter, but does not require RF07MIN.

Pack Files

RF can store all media inside a pack file which is defined in the realityfactory.ini file. The default pack file for RF is pack.vfs and is located in the RF root folder. Pack files store everything from the menu media, to the configuration ini files. The one thing a pack file can not store however, are scripts. If using a pack file, the scripts folder must be kept separate.

The contents of a pack file are managed using the RF VFS Explorer tool. This tool can create new pack files and edit existing ones. The contents of a pack file, look exactly

like the RF directory tree, excluding the scripts folder and the files within the RF root directory. The only benefit of using a pack file, is that it can password protect the contents. This ensures that no one can extract the game media, for use in another game.

Assembling a Demo

As it was said in the first chapter, RF is a ready made game, that can be configured to be another game entirely. In order to prepare a demo, all that is required is to create a folder with the minimum required RF files, the game demo configuration files and the media used by the demo. The following is a step by step example, of how the Space Escape demo was assembled into a game that would be ready for packaging.

- **Step 1** - A new folder was created named SpaceEscape Demo. The first files that were copied into this folder were the files required to run RF. This includes every file in the root directory except for, temp.bin, changelog.doc and WireDrv.dll. Copy the rest of the files into the new folder named SpaceEscapeDemo.
- **Step 2** - Then the Scripts folder, the install folder and the media folder, are all copied to the new SpaceEscapeDemo folder as well.
- **Step 3** - The next step is to configure the ini files to remove the headings that are not required for the demo. If these are not removed, then RF will try and load the media that they define. The following is a list of the ini files used for Space Escape, and the only headings which should be defined in each. Any ini files which are not

mentioned, were not edited in any way during this process. With only these headings defined, the required media has been reduced to only what the Space Escape Demo uses.

- **Effects.ini**

⇒ [YellowLight]
⇒ [LargeExplosion]
⇒ [Corona]
⇒ [BotSpray]
⇒ [BlueCorona]

- **Explosion.ini**

⇒ [GrenadeExplosion]
⇒ [CorExpl]
⇒ [BotExpl]

- **Pawn.ini**

⇒ [Android]

- **Weapons.ini**

⇒ [PlasmaBullets]
⇒ [PlasmaRifle]

- **Player.ini**

⇒ [health]
⇒ [PlasmaBullets]
⇒ [PlasmaRifle]

- **Inventory.ini**

⇒ No Headings - Blank File

- **Character.ini**

⇒ No Headings - Blank File

- **Step 4** - With the configuration files reduced to the bare minimum, the unused media was then removed from the copied over media folder. Below is a list of the entire contents of the media folder used by the Space Escape Demo. Media that is required by default, even if it is not used in the demo, is shown with a *. Folders are in bold.

- **Actors**

⇒ android1.act

⇒ bullet1.act

⇒ Ernie.act

⇒ PlasmaRifle.act

- **Audio**

⇒ die.wav

⇒ die1.wav

⇒ footstep.wav

⇒ injury.wav

⇒ injury1.wav

⇒ Raygun.wav

⇒ spacemusic.wav

⇒ Zapper2.wav

⇒ **Menu**

⇒ pop1.wav

⇒ pop2.wav

- **Bitmaps**

⇒ **Conversation** - Empty *

⇒ **Explode - Large** - All Contents

⇒ **Fonts** - All Contents

⇒ **Hud**

⇒ gun_indicate.bmp

⇒ gun_indicate_alpha.bmp

⇒ gun_shell.bmp

⇒ gun_shell_alpha.bmp

⇒ health_indicate.bmp

⇒ health_indicate_alpha.bmp

⇒ **Inventory** - Empty *

⇒ **Menu** - This folder only contains the media which is defined in the spacemenu.ini file. This includes, the backgrounds, the title images, the images and the cursor.

⇒ **Terrain** - Empty *

⇒ a_bubl.bmp *

⇒ a_flame.bmp *

⇒ a_lvsmoke.bmp *

⇒ a_rain.bmp *

⇒ airlock.bmp

⇒ Bolt.bmp *

⇒ Corona.bmp *

⇒ Corona_a.bmp *

⇒ flame03.bmp *

⇒ g_bubble.bmp *

⇒ lvsmoke.bmp *

⇒ rain.bmp *

- **Levels**

⇒ escape.bsp

- **Video - Menu** - All Contents

These four steps copy and then reduce the media

which goes into the SpaceEscapeDemo folder. What was required after, was to try and run the demo and see if any media is missing. If RF does not find a defined media, it will crash and return an error specifying the name of the missing file and where it looked for it. If this happens when packaging a demo, then the missing file is added and then the demo is executed again. This process is repeated until the demo successfully runs.

Final Testing

When the demo does not crash and ask for media, then it is very important to test the game by playing it to the end and taking it through every possible scenario. This will reveal any missing media, that is not loaded before the level starts. This can include sounds that are defined in a pawn script, effects and level transition splash screens. These media elements, are only loaded as required and can cause RF to crash if not found.

Testing can not be stressed enough. Test, test and re-test the final demo before releasing it. Try every scenario including the players death sequence and the completion of the level. Failure to completely test a level before releasing it, is a very common error found in many game demos. Especially those released by a very small development team. Once again, be sure to completely test a level, before releasing it to the public.

Be sure to also test the game play of the level. Calculate the amount of ammo required to clear the game and then decide on the level of inaccuracy that will be allowed

for the player to still win. The same goes for the health attribute. In the case of the Space Escape demo, this is not necessary, because the weapon always recharges itself, but the health should be considered as well. How many times will the player be ale to be shot before dieing, will determine how easy or difficult the game will be.

Game Identity

When the developer is confident enough that the game is suitable for distribution, the final steps are to give identity to the game, by changing the icon and then to prepare a distribution package from the newly created files. this will require a couple of third party tools, that are readily available on-line.

First on the list is a tool that can change the executable icon of the RealityFactory.exe file. Two such tools include, Icon Forge and IcoChanger (Not Icon Changer). These can both permanently change the icon of a Windows exe file. Before changing the original icon though, there must be an icon that can replace it. There are many freeware icon making programs and most are capable of creating a 16 color, 32 x 32 icon file. Any one of these programs can be used to create a suitable replacement icon. As an alternative, there are also many freeware programs that can convert a bmp file into an ico file. Most of these can also be used. One such example is, QTAM - BMP to ICO.

After changing the icon for the RealityFactory.exe file in the demo folder, the next step is to change the name of the file, to the name of the demo. In this case, RealityFactory.exe

was renamed, SpaceEscape.exe. Also, in the realityfactory.ini file, there is an entry named, GameName. Change this to match the name of the demo. This is displayed in the task bar box, that becomes visible when RF runs.

Packaging the Demo

With the game demo ready to be distributed, all that is required is to turn the folder containing the game into a single file that can install the game on another system. The author has always used Install Maker Pro to release free game demos, because it is very simple to use and has minor restrictions applied to the shareware version. It can easily create a simple installation file, without any need to write a setup file. A release for a commercial demo however, should not use something as simple as Install Maker Pro and instead should use a program such as Inno Setup. This can let other files be defined in the Program Start menu, giving the player access to the Video Setup program and a help file. Also, the Inno Setup license has no splash screens or usage restrictions.

When preparing the distribution package, the most important thing to consider is the graphics mode for the game. Is it 16 bit or 32 bit? Will this be fixed at the start or will the player be allowed to select it after? Some computers can not use the 32 bit system so just to play it safe, it may be best to distribute any demos in 16 bit mode and then allow the player to select a 32 bit mode if they desire to.

Advanced Level Design

Reality Factory is a very powerful game engine and the Space Escape demo barely scratches the surface of what it can do. RF has many features, that allow it to create very large and sophisticated levels. As of the RF07D update, it also incorporates variable LOD for actors, environment mapping and a scriptable SkyDome, to name just a few features. By understanding how to use these features, a game can be designed with these features being considered. This can lead to a very sophisticated game, that can easily meet up to commercial standards.

Clipping and Fogging

The number one way to increase frame rate and to best keep it at a steady level, is to use the Clipping and Fogging feature that is defined in the EnvironmentSetup Entity. This system makes use of two elements that when combined, limits the rendering of objects and geometry in the level, to a small area that surrounds the player.

On the next page is a diagram which describes the relationship between the properties defined in the EnvironmentSetup Entity. The Fog settings should always be less then the Clip settings, so as to give the illusion that level geometry is faded into view. What's really going on however,

Fogging and Clipping

The outer square represents the Far Clip Plane and the circle is to represent the point at which the fog becomes solid. The fog should become solid just before the edge of the Clip Plane. This gives the appearance that objects are slowly fading into view.

is that behind the Fog, no geometry or objects are being rendered. Below is a breakdown of the settings in the EnvironmentSetup Entity, that control the fogging and clipping of a level.

- **FarClipPlaneDistHigh** and **FarClipPlaneDistLow** - These define the distance from the player, where level Clipping begins. All level geometry will not be rendered past this point. It should be at greater then the values used for the fogging distances.
- **FogStartDistHigh** and **FogStartDistLow** - This is the distance from the player where the fog has a transparency value of 0 or in other words, does not fog anything. It is the point where the fog starts.
- **TotalFogDistHigh** and **TotalFogDistLow** - These values define the point at which the Fog becomes solid and can not be seen through. The difference between the **FogStartDistHigh** and the **TotalFogDistLow**, is the

distance that the Fog gradually goes from invisible to solid.

- **EnableDistanceFog** and **UseFarClipPlane** - These must be set to true, in order to use the Fog and Far Clipping Plane.
- **LODdistance5** - If the level was made using version RF07D, then this value will define the distance, that an actor will not be rendered by the engine. This distance should equal the FarClipPlaneHigh property.

The Clip Plane can also be used with the SkyDome entity, by setting the width of the SkyDome to be less then the width of the Clip Plane. To make this work however, the SkyDome must be told to move with the player at all times. The SkyDome Script commands, that control the radius of the SkyDome and the player following are, **SetLandscapeSize()** and **ToggleMoveWithCamera()**, respectively.

Player Scale

Very large levels are created by using a very small actor scale. Actor scale is defined in the PlayerSetup Entity, with the properties, ActorScale and PlayerScale. An actor scaled down to 0.5, will effectively create the illusion that the level has doubled in size. The same level using a player scale of 0.1, will seem 10 times larger. This has many benefits, including the use of smaller brushes to make the world, which translates into smaller textures. These can be scaled down, to achieve a crisp appearance. However, when a player is scaled down, other actors will need to be scaled down as well. This includes the Scale settings in the weapon.ini file for each weapon, the pawns and the level actors, such as

StaticEntityProxies and Attributes. These must all be reduced, in proportion to the player scale. Also, two properties of the Environment Setup entity must also be adjusted. This include the JumpSpeed and the Speed settings. These will both need to be significantly reduced, if using a very small actor sale.

Beyond 4000 Texels

The Genesis Engine has trouble rendering past a distance of 4000 texels, but there are many ways to get around this. Many designers begin with a large hollow cube to surround a level, this makes sense with smaller levels, but can prove to be a problem when designing very large levels. This means that when designing a large level, the most efficient means of doing so is to make it from connected areas, that are no larger then the 4000 texel limit. The important thing to consider when building a level in this manner however, is that there can be absolutely no spaces between outer brushes whatsoever. Any non-connected brushes, will cause a level leak and prevent the level from compiling. Levels like this, are best built by creating the outer brushes using a Snap setting of 8 texels, with brushes that are sized to be a multiple of 8.

Using MAP Terrains

A popular program that can be used to create a MAP file of a terrain, goes by the name Nems 3D Terrain Maker. These MAP files, can then be imported into the World Editor as level geometry. When exporting a terrain from Nems, that is intended to be used in RF, it is important to not export the Hint Brushes as well. The terrain is then imported using the

Import option, under the *File* menu of World Editor. Maps are sometimes used to fill an entire level, but it may be more efficient to use smaller hills placed onto a flat brush. This is more commonly seen in commercial games and is a much easier way to mix terrains with flat planes.

Terrains can also sometimes slow things down, due to the large number of brushes that they are created with. A means of helping out the frame rate loss, is to flag all of the faces of the terrain and connecting flat brushes, to be Gourard shaded or Flat. This will not dynamically shade the faces of the terrain, greatly helping to restore frame rate losses.

Using Teleporters

Another means of creating very large levels, is to section them off into separate areas and connect them using the Teleporter and TeleportTarget Entities. These two entities work together to allow an instant repositioning of the player from the point defined by the Teleporter, to the Teleport-Target Entity.

Teleporter entities, require a model that will be the contact point for the teleportation to occur. To create a seamless transition, the model which activates the player teleportation sequence, should be an invisible brush, that is placed in the middle of a narrow passage. The same narrow passage, is duplicated as the entrance to another section of the level. This is where the TeleportTarget Entity is placed. By adding a second transparent brush before the TeleportTarget Entity, the passage can be used as a two way system.

Another practical use for the Teleporter Entity, is to create a level that mixes indoor areas with outdoor areas. For example, a level that is meant to start outdoors, but allows the player to enter into buildings. An efficient means of developing this, is to have the outdoor buildings made of solid geometry, which is separate from the indoor areas. The player is teleported to the indoor areas, when they come into contact with a door of a building. An example of this is demonstrated in the RF demo, The Ransom. When the player is lead to the Kidnapper, they are teleported to an area that has the inside of the apartment containing the hostage. The area has a portion of the surrounding outdoor area recreated, in order to allow interaction between the inside of the building and the outside streets.

Making 3D Games with Reality Factory

3D Games Without Programming

ISBN 0 - 968715 - 4 - X

Bookshock Publications

www.bookshock.com

ISBN 0-9687185-4-X

9 780968 718544