Chapter 1

FOLF-Ray Developer Reference

1.1 Resource Script Reference

All the images, models, sounds and code that belong to a game project are listed in a manifest file inside the project folder. If there is a folder next to the program binary named `res`, then it will look inside for a `manifest.fdd`. If this file is found, the engine will try compiling the game before executing it. This is a text file describing each asset using the following form:

```
// Optional comments
type name
{
   field = value
}
```

`type` must be a valid resource type, `name` will be the name that refers to it in scripting, and several field-value pairs are listed inside the {} brackets. Values can be names or numbers and engine-defined constants begin with the `$` symbol.

1.1.1 Resources

`script`

- `source` = Name of the Squirrel script (`.nut`)

Specifies a Squirrel script to be compiled and included in the game. Place these scripts at the top level of the `res` folder. These scripts will be compiled in the order listed in the manifest, so any scripts relying on functions or other references from another script must be listed after it.

`image`

- `source` = Name of the image file (`.png`)

Specifies a 2D image file located in the `img` subfolder within `res`.

`mesh`

- `source` = Name of the mesh file (`.obj`)

Specifies a file to be used for 3D mesh geometry. All faces must be triangles, and take care to use as few triangles as possible or the performance will drop further than expected. Place these in the `msh` subfolder within `res`.

`sprite`

- `source` = Name of the image file (`.png`)
- `wide` = *(Optional)* Defines the number of animation frames wide the sprite sheet is
- `tall` = *(Optional)* Defines the number of animation frames tall the sprite sheet is

Specifies a 2D image file to be used either as a billboard style sprite or a facing sprite within a 3D world. Place these in the `obj` subfolder within `res`. If you want a facing sprite that has animation frames for each direction, then your sheet must be 8 frames tall.

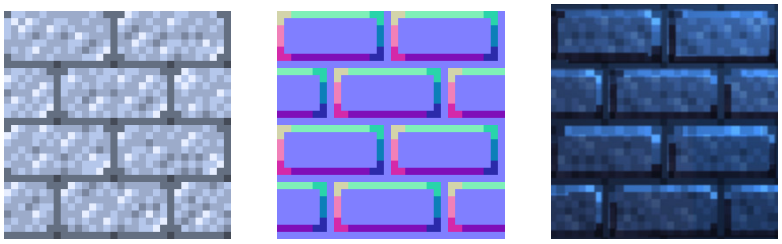
`sound`

- `source` = Name of the sound clip file (`.wav`)

Specifies a sound clip. These must be single channel (mono), 8-bits per sample and in 44,100, 22,050, or 11,025 hertz sampling rates. These clips can be played within the world (3D sound) or directly through either the left or right speakers.

`fog`

- `scaling` = Scaling of fog thickness with distance
- `start` = Distance at which fog starts to become thick
- `end` = Distance at which fog is completely thick
- `strength` = Strength of fog, at 1.0 it completely occludes everything beyond the end distance
- `red` = Red component of color (filter)
- `green` = Green component of color (filter)
- `blue` = Blue component of color (filter)



(a) Color Map (b) Normal Map (c) Render

Figure 1.1: The color map (a) and normal map (b) of a brick texture. The color map is how the material would appear with no lighting or shading, the normal map defines the variation on slopes for how the lighting is affected by the texture. (c) shows an example of how this material might appear in game.

Defines a fog that can be used for the world or materials. The values for scaling can be:

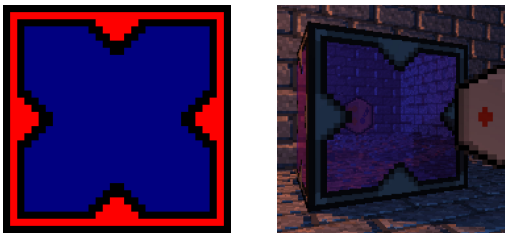
`$FOG_LINEAR`	0
`$FOG_QUAD`	1

`material`

- `texture` = Name of the color map image (`.png`)
- `normal` = Name of the normal map image (`.png`)
- `effect` = Name of the effect map image (`.png`)
- `alpha` = *(Optional)* Name of the alpha map image (`.png`)
- `specular` = *(Optional)* Defines the power of specular highlights
- `refract` = *(Optional)* Defines the index of refraction
- `fog` = *(Optional)* Assigns a fog for the interior of this material

Defines a material to be used by any 3D mesh in the world. The texture map defines the colors of the material or its basic appearance. The normal map defines the surface normal at each point using the $(red, green, blue)$ channels as (x, y, z) vectors where grey is the origin at $(0, 0, 0)$. The effect map defines the specular strength (red) of highlights, emissive strength $(green)$, and reflective strength $(blue)$ at every point. Specular highlights reflect light, but not the scene and can be adjusted using the `specular` field. Emissive portions of the material act as though they emit light, but won't light other nearby 3D objects. Reflective portions of the material will reflect the environment. The alpha channel will either be defined by the actual alpha channel in the `texture` image, or optionally as the (red) channel of the `alpha` image. Parts of the material that are translucent will have a refraction index defined by the `refract` field. Several helpful constants have been defined for refraction indices:

(a) Effect Map (b) Render



(c) Effect Map (d) Render

Figure 1.2: The effect map for the emissive example (a) has both full specular and emissive strength so the color used in the center is yellow $(1, 1, 0)$ and the gem in the center appears lit (b). The effect map for the reflective example (c) only reflects the environment in the blue region of the effect map (d).

`$REFRACT_AIR`	1.0
`$REFRACT_WATER`	1.33
`$REFRACT_ICE`	1.309
`$REFRACT_PLASTIC`	1.49
`$REFRACT_GLASS`	1.52
`$REFRACT_DIAMOND`	2.42

1.1.2 File Types

FOLF-Ray only accepts resources in certain file types, with some restrictions.

(`.nut`) Squirrel Program Script

> FOLF-Ray uses Squirrel as the programming language to write games in. Scripts are to be placed at the top level of the `res` folder and listed in the manifest. The engine will then compile and run them in the order specified in the manifest.

(`.png`) Portable Network Graphics (PNG) Image

This is the fundamental image file format used by FOLF-Ray. All images must have dimensions in multiples of 8 and be in 24-bit (RGB) or 32-bit (RGBA) color.

(`.obj`) Wavefront OBJ Format Mesh

Arbitrary meshes for entities use the Wavefront OBJ format. Meshes are imported as rigid geometry where only triangles are allowed for faces. Texture mapping coordinates are used, but any actual material defenitions are discarded as these are assigned during the game through scripting. Only one object may exist in meshes used for FOLF-Ray.

(`.wav`) Waveform Audio File (WAV) Format

This is the fundamental sound file format used by FOLF-Ray. Sound clips must have only one channel (mono) and samples at 8-bit depth. Only the sampling rates 44,100, 22,050, and 11,025 hertz are allowed. Stereo sound is achieved through the use of 3D sounds or 2D sounds played at varying balances.

1.2 API Reference

The engine uses Squirrel as the programming language for game projects. All the code you want to use needs to be defined, in order, as `script` entries in the resource manfiest. Upon starting the game, the engine will execute the scripts in this order.

1.2.1 Types

`bool`

Functions that return boolean values can be used in if statements, these are the base Squirrel boolean values.

`int`

The integers used in FOLF-Ray are the base Squirrel integers.

`number`

General numbers passed as arguments to API calls may either be integers or floating point numbers.

`entity`

Entity handles are `int`s that refer to entities in the world. Since entities can be added or removed arbitrarily during the game, these handles are only valid within the frame they were obtained. Attempting to use a handle out of the frame will usually cause a runtime exception.

`function`

Functions to be used for hooks and callbacks in FOLF-Ray are referred to by name. Pass the name of the function you want to use as a string. See the specific API function details to see what parameters you will be passed in this function.

`string`

Text strings used in FOLF-Ray are the base Squirrel strings.

`vector`

A vector in FOLF-Ray is a Squirrel array of three `number`s. These are commonly used to refer to locations $[x, y, z]$, orientations $[pan, tilt, roll]$, or colors $[red, green, blue]$.

`void`

This is used to denote a function that doesn't return anything.

1.2.2 Functions

Entities

```
entity fr_ent_create ( )
```

Creates an entity, adds it to the world, and then returns the handle value. By default, the entity will have no body to be rendered or collided with. Use the returned value to configure the entity further.

```
void fr_ent_morph_mesh ( entity who, material material, mesh mesh )
```

Gives the entity (`who`) a rigid mesh (`mesh`) body with the material specified (`material`). The entity will become visible after this.

```
void fr_ent_morph_sphere ( entity who, material material )
```

Gives the entity (`who`) a perfectly spherical body with the material specified (`material`). The textures in that material will be wrapped around the sphere in an equirectangular projection: the image wraps around horizontally so the x coordinate of each pixel corresponds to a longitude and the y coordinate corresponds to a latitude. The center of the texture will be facing towards the front where the entity would normally be facing, therefore, adjusting the entity's orientation will rotate the texture mapping to give the sense of rotation (since a sphere alone has nothing to distinguish its orientation). The radius of the sphere can be set through `fr_ent_set_scale` and will be interpreted in game units. If the sphere is inverted with `fr_ent_set_invert` it will allow all light traces to pass and will not cast shadows. Inverted spheres will only ever recieve ambient light, so you will need to light the material using the emissive channel of the effect texture (see `material`).

```
void fr_ent_morph_sprite ( entity who, sprite sprite, int billboard )
```

Gives the entity (`who`) a sprite body using the specified sheet (`sprite`). If (`billboard`) is 0, then the sprite faces the camera and has optional directional animation if its sprite sheet has 8 rows of frames. If 1, the sprite obeys the pan angle of its orientation and appears like a flat 2D billboard. The entity will become visible after this.

```
void fr_ent_morph_draw ( entity who, int back )
```

Converts the entity (`who`) into a 2D drawing one. Its behavior function will be run when its time to draw 2D elements on the screen instead of with the other entities. The 2D elements will appear either behind the world (background) or on top of it (user interface) depending on the value of (`back`):

`DRAW_TOP`	0
`DRAW_BACK`	1

```
void fr_ent_morph_light ( entity who, int type, int dropoff, number
focus, number radius )
```

Turns the entity (`who`) into a light source. The light can be a directional or omnidirectional light based on the value for (`type`):

`LIGHT_OMNI`	0
`LIGHT_CONE`	1

The attenuation of the light according to distance can be linear or quadratic based on the value for (`dropoff`):

`LIGHT_LINEAR`	0
`LIGHT_QUAD`	1

If this light is directional, then the (`focus`) increases the focus power of the light if you use higher values to be more like a spotlight, and lower values make the light more broad. The range of the light can be specified in (`radius`). The entity itself isn't visible, but a light is placed at its current location. Moving and orienting the entity will move and orient the light.

```
void fr_ent_set_scale ( entity who, number factor, int texunlock )
```

Changes the size of the entity (`who`). The scaling factor is given as (`factor`) and if you want the textures on the faces of the entity's mesh to not scale with it (unlocked texture scaling), then pass 1 to (`texunlock`) and 0 otherwise.

```
void fr_ent_set_invert ( entity who, int enable )
```

Enables or disables inversion for the entity (`who`). Inverting an entity flips the normals on the faces of the entity's mesh, giving it an inside out look.

```
void fr_ent_set_position ( entity who, vector pos )
```

Places the entity (`who`) in a new location specified by vector (`pos`). No collision detection will be performed.

```
vector fr_ent_get_position ( entity who )
```

Returns the current location of entity (`who`).

```
void fr_ent_set_orientation ( entity who, vector ang )
```

Changes the orientation of entity (`who`) using the angles in vector (`ang`). No collision detection will be performed.

```
vector fr_ent_get_orientation ( entity who )
```

Returns the current orientation of entity (`who`).

```
void fr_ent_set_color ( entity who, vector col )
```

Changes the color of entity (`who`). If this is an entity with a visible body like a mesh or sprite, the color specified (`col`) is multiplied with its sprite image or color map. If this entity is a light, the color of the light will change.

```
void fr_ent_set_emissive ( entity who, vector col )
```

Changes the emissive color of entity (`who`). If this entity is a mesh, the color specified (`col`) is multiplied with any emissive portions of the entity's `material` while anything not emissive is unaffected. If this entity is a sprite, this color will be added to the sprite image allowing it to 'glow'. This function has no effect on actual light entities, however.

```
void fr_ent_set_alpha ( entity who, number alpha )
```

Changes the overall alpha value of the entity (`who`) to the new value (`alpha`). This causes the whole entity to become more (0.0) or less (1.0) translucent.

```
void fr_ent_set_behave ( entity who, function fn )
```

Sets the behavior of the entity (`who`). The function (`fn`) will be called every frame and the handle of the associated entity (`me`) will be passed to it:

```
function fn(me)
{
   // Implement entity behavior here
}
```

```
void fr_ent_give_name ( entity who, string name )
```

Gives the entity (`who`) the name (`name`) so that you can obtain an `entity` handle to it at any point later. Since you cannot save entity handles, this is how you would reference an entity globally.

```
entity fr_ent_get_by_name ( string name )
```

Obtains an `entity` handle to a previously named entity with name (`name`). If an entity with that name does not exist in the world, a runtime exception will occur.

Render

```
void fr_render_set_ambient ( vector col )
```

Sets the world's ambient light color to (`col`). In absence of all light, this is the minimum light level and color you will see. If you set (`col`) to $[0, 0, 0]$ then anything not lit will be pitch black. The player would be unable to see that without light sources.

```
void fr_render_set_sun ( vector col )
```

Sets the color of the world's sun light to (`col`). If you don't want the world to have sun light, set (`col`) to $[0, 0, 0]$.

```
void fr_render_set_sun_direction ( vector dir )
```

Sets the direction of sunlight. The value you pass for (`dir`) should be a vector that points in the direction of the sun and it doesn't have to be normalized. If you want the sun directly overhead, you can pass $[0.0, 0.0, 1.0]$ for (`dir`).

```
void fr_render_set_clear_color ( vector col )
```

Sets the clear color to (`col`). Before anything is drawn, the screen is cleared to this color and therefore you will see this in parts of the screen that are absent

from any visible objects.

```
void fr_render_enable_fog ( int scaling, number start, number end,
number strength, color filter )
```

Enables fog for the current world. The (`scaling`) defines how the fog thickness scales with distance:

`FOG_LINEAR`	0
`FOG_QUAD`	1

The (`start`) distance defines when the fog starts to accumulate and the (`end`) distance is when the fog is entirely applied, occluding everything behind it if (`strength`) is 1.0. The (`strength`) defines the overall thickness of the fog (between 0.0 and 1.0). The color (filter) of the fog is set via (`filter`).

```
void fr_render_select_fog ( fog fog )
```

Enables fog for the current world using a (`fog`) defined in the resource manifest.

```
void fr_render_disable_fog ( )
```

Disables fog for the current world.

Camera

```
void fr_camera_set_position ( vector pos )
```

Places the camera in the world at location (`pos`).

```
void fr_camera_set_orientation ( vector ang )
```

Orients the camera according to the angles in (`ang`).

Input

```
bool fr_input_get_button ( int button )
```

Gets the state of the virtual button defined by (`button`). If the button is being held down, a true value is returned. The button codes are:

INPUT_FORWARD	0
INPUT_RIGHT	1
INPUT_BACKWARD	2
INPUT_LEFT	3
INPUT_SCREENSHOT	4
INPUT_OK	5
INPUT_FIRE	6
INPUT_ALT_FIRE	7

```
void fr_input_react ( function fn, int button )
```

Assigns a function (`fn`) to react when button (`button`) is pressed. This function will be called without parameters when the button is pushed down, but won't be called again until the player releases the button first. Button codes are listed under fr_input_get_button. A function similar to this is expected:

```
function fn()
{
   // Implement reaction here
}
```

```
number fr_input_aim_x ( )
```

When the mouse is in aiming mode, this returns the offset in the x direction the mouse is being moved in with positive numbers moving to the right relative to the screen.

```
number fr_input_aim_y ( )
```

When the mouse is in aiming mode, this returns the offset in the y direction the

mouse is being moved in with positive numbers moving to the bottom relative to the screen.

```
void fr_input_pointer_mode ( image ptr )
```

Enters mouse pointer mode and changes the mouse cursor to (`ptr`). This allows the user to move the pointer around with the mouse.

```
void fr_input_aim_mode ( )
```

Enters mouse aiming mode. The pointer will no longer be visible.

Math

```
vector fr_math_rotate_vector ( vector vec, vector ang )
```

Rotates the vector (`vec`) using the angles in (`ang`) about its origin $[0.0, 0.0, 0.0]$. This allows you to rotate points in 3D space using the same series of rotations the engine uses for all other orientations. The rotated vector is returned.

```
vector fr_math_add_vector ( vector vec1, vector vec2 )
```

Adds the two vectors given (`vec1`) and (`vec2`) element-wise and returns the result.

```
number fr_math_sin ( number deg )
```

Calculates the sine of a given angle (`deg`), in degrees.

```
number fr_math_cos ( number deg )
```

Calculates the cosine of a given angle (`deg`), in degrees.

```
vector fr_math_to_screen ( vector pos )
```

Converts from a 3D position in the world (`pos`) to one on the screen in pixels and returns it. The z component of this converted position is the depth, in game units, and negative values means the position is behind you. If you're using this value to label objects in the game world with 2D elements, you should check to see if the z component is positive before drawing.

Color

```
vector fr_color_from_rgb ( int red, int green, int blue )
```

Allows you to define a color conventionally using three 8-bit values (between 0 and 255). Specify the values for the (`red`), (`green`), and (`blue`) channels. The matching color vector will be returned.

Sound

```
int fr_sound_play ( sound snd, vector pos, int loop, number radius,
number volume, number speed )
```

Plays a sound (`snd`) in the world at location (`pos`). The audible range of the sound should be given as (`radius`) at an overal volume (`volume`) (between 1.0 and 0.0). The (`speed`) parameter changes both the pitch and duration of the sound and is relative to the original clip's speed (0.5 is half speed and 2.0 is double speed). Once played, this function returns an integer you can use to refer to the sound later if you intend to alter it while its playing. The sound will loop or play only once depending on the value of (`loop`):

```
PLAY_ONCE   0
PLAY_LOOP   1
```

```
int fr_sound_play_2d ( sound snd, int loop, number volume, number speed,
number balance )
```

Plays a sound (`snd`) directly in 2D. This is often used for first person sounds, UI sound, etc. See `fr_sound_play` for values to specify for (`loop`). Specify the overal (`volume`) between 1.0 and 0.0. The (`speed`) parameter changes both the

pitch and duration of the sound and is relative to the original clip's speed (0.5 is half speed and 2.0 is double speed). (`balance`) specifies which channel the sound will be played out of. -1.0 targets the left speaker, 1.0 targets the right one and 0.0 plays equally loud out of both. Once played, this function returns an integer you can use to refer to the sound later if you intend to alter it while its playing.

```
void fr_sound_adjust_volume ( int sndid, number volume )
```

Adjusts the sound volume of sound referenced by handle (`sndid`) to new volume (`volume`).

```
void fr_sound_adjust_speed ( int sndid, number speed )
```

Adjusts the sound playback speed of sound referenced by handle (`sndid`) to new speed (`speed`).

```
void fr_sound_adjust_position ( int sndid, vector pos )
```

Moves the location of a 3D sound referenced by handle (`sndid`) to new location (`pos`).

Time

```
number fr_ts ( )
```

Returns the current time step. Multiply this value with your speeds to get frame independent speeds. For example, to get an entity to move 5 units per second, you would add its position with `5.0*fr_ts()`.

```
number fr_t ( )
```

Returns the current uptime of the engine, in seconds.

Image

```
int fr_image_width ( image img )
```

Returns the number of pixels wide the image (`img`) is.

```
int fr_image_height ( image img )
```

Returns the number of pixels tall the image (`img`) is.

Draw

```
void fr_draw_image ( image img, int x, int y, int sx, int sy, int w, int
h )
```

Draws an image (`img`) onto the screen. The image will be drawn from the top left corner onto the screen at (`x`) (`y`) (in pixels). You can specify where within the image to start drawing from with (`sx`) and (`sy`). If your image is a sprite sheet or otherwise contains multiple smaller images, place this coordinate at the top left of the one you want to draw (in pixels). The actual width and height of the resulting image can be specified in (`w`) and (`h`). This can be smaller than the image, resulting in a cutout of it being drawn.

```
void fr_draw_image_simple ( image img, int x, int y )
```

Draws an image (`img`) onto the screen at location (`x`) (`y`) (in pixels). The whole image will be visible.

```
int fr_draw_width ( )
```

Gets the width of the screen, in pixels.

```
int fr_draw_height ( )
```

Gets the height of the screen, in pixels.