Fractal Block World 1.01.32
Creation Manual

by Dan Hathaway

February 21, 2026

Contents

Chapter 1

Introduction

1.1 An Important Link

See the following link

http://danthemanhathaway.com/ComputerGames/FractalBlockWorld/ReleaseMisc/Packages/

for tutorials and guides on making packages.

1.2 Two Types of Packages

There are two types of packages you can make: standalone package, which must be installed in Data/Packages, and mods for standalone packages which must be installed in Input/Packages. Often in this document we will assume that the package is a standalone package.

1.3 Your Package

Within the root folder of the Fractal Block World program, there is a folder called Data. Within that there is a folder called Packages. To create your own world, you can start by copying the folder Data/Packages/blank to something like Data/Packages/myworld.

For safety, never modify the **blank** package.

You should also never modify the "base" package.

When you create a new game, you can now select the "myworld" package. You can modify the relevant files within the "myworld" directory and its subdirectories to create your world. For the rest of this chapter, we assume that your package is called "myworld".

It might be a good idea to include an abbreviation for your package in the names of .lua files within the package. So for example, if you are making a package called fun_blocks, then you could have the lua files "game_fb_give_health.lua" and "bent_fb_armor_25.lua". This is especially important if you are making a mod (that way different mods do not accidentally have scripts of the same name).

Certain files are required to be named certain things because they are entry points. For example, "Game/top.lua".

1.4 dependencies.txt (Mandatory)

Your package must contain the file called dependencies.txt in its root directory. For example, if your package is called myworld and your package is installed in Data/Packages, then the file Data/Packages/myworld/dependencies.txt must exist. This file specifies which packages your package depends on. In terms of what this means, **the engine guarantees that the dependencies of your package are loaded before your package.**

While the system supports complicated dependencies, it is best to simply only depend on the "base" package if you are making a standalone package. So the dependencies.txt file should read as follows:

```
wf base
```

If you are making a mod for a standalone package called foo, your dependencies.txt should read as follows:

```
wf foo
```

Note: wf stands for "well-founded". If X is a well-founded dependency of the current package, then the package X cannot depend on the current package.

Indeed, there cannot be any "cycles" in the dependencies. For example, it is not allowed for package X to depend on package Y, package Y to depend on the package Z, and also package Z to depend on package X.

The dependencies.txt file is used so that those packages listed are loaded BEFORE your package. So if you have the line "wf xar", then the xar package will be loaded before your package.

The dependencies do not need to be "transitive". That is, if your package is called X and it depends on Y, which in turn depends on Z, then your dependencies.txt should list Y but it does not have to list Z.

When the user selects which mods are enabled, they must order the mods so that all these dependency relations are satisfied.

The order in which you list dependencies in your dependencies.txt actually does not matter. It is up to the user to order them in the game's mod selection GUI.

The following is very important: If you are referring to a package in the Input/Packages directory in the dependencies.txt file, you much prefix its name with __USER__ to differentiate it from packages in the Data/Packages directory.

1.5 Dependencies Example

Suppose you are making a mod called advanced_xar_mod and it depends on another mod called basic_xar_mod. So, here are the relevant packages:

```
Data/Pacakges/base
Data/Pacakges/xar
Input/Packages/basic_xar_mod
Input/Packages/advanced_xar_mod
```

Note that the dependencies.txt of the Data/Packages/xar package is

```
wf base
```

Let's say the dependencies.txt file of the Input/Packages/basic_xar_mod is

```
wf xar
```

Then the dependencies.txt file of Input/Packages/advanced_xar_mod should be

```
wf xar
wf __USER__basic_xar_mod
```

Actually, the "wf xar" is not needed here in this last file because it is listed in the basic_xar_mod, but it doesn't hurt to be safe.

1.6 Mods and a Simple Texture Pack Example

If instead of making your own standalone package from scratch, you can make a "mod" for a standalone package (for the xar package let's say). Instead of creating a package in Data/Packages you create it in Input/Packages. In this case, your mod package would depend on xar. See the game's website for how to make a mod like this. Specifically, you can follow the link:

http://danthemanhathaway.com/ComputerGames/FractalBlockWorld/ReleaseMisc/Packages/

and see the guide for creating a "texture pack" that will work with the xar package.

1.7 The file About/install_dir.txt (Mandatory)

Your package must contain the file "About/install_dir.txt". It should have exactly one line, which is either "Data/Packages" or "Input/Packages" (with no quotes), depending on where your package should be installed.

1.8 The file About/mod_for.txt (Mandatory)

Your package must contain the file "About/mod_for.txt". If your package is a standalone package, leave this file blank. If you are making a mod for a standalone package, have the About/mod_for.txt contain the name of the standalone package. For example, if you are making a mod for the xar package, then the file About/mod_for.txt should read as follows:

```
xar
```

1.9 About/about.txt and About/thumbnail.jpg

In the file About/about.txt (optional) you can have a small amount of text which describes your package. This can be seen in the game.

The file About/thumbnail.jpg (optional) should be a 512 by 512 jpeg image showing off you package. This can be seen in the game.

1.10 Subdirectories

Within Data/Packages/myworld there are the following directories:

- About
- BasicEnts
- EnvRects
- Game
- Meshes
- MovingEnts
- Music
- Sounds
- Textures
- WorldNodes

The directory WorldNodes is where the block data of the world is stored.

Within Data/Packages/myworld there are several files not in directories:

- binds.txt
- dependencies.txt
- globals.txt

- light_params.txt

The file "binds.txt" specifies what events occur when various keyboard keys or mouse buttons are pressed.

The file "dependencies.txt" was described in the previous section.

The file "globals.txt" declares game variables that the lua scripts are able to modify and access.

The file "light_params.txt" (optional) is only used if you are making a standalone package. This file is loaded early on when the package is loaded. Just see the contents of the file "Data/Packages/xar/light_params.txt". This file specifies the "chunk width" (which for now must be 16), the "version of the package", and the "version of the Fractal Block World program" that you designed your package for. This helps track of how "up to date" the package is with the current version of the engine.

1.11 Errors (causing the program to exit)

The program can exit from 3 types of errors: system errors, hard user errors, and soft user errors.

1.11.1 System Errors

System errors are generally due to bugs in the program. Sometimes these result in the program exiting without displaying an error message. **You can go to the file stdout.txt and go to the end to see what was the error.**

1.11.2 Hard User Errors

Hard user errors are caused by bad data given to the engine. "Hard" means the program will always exit when such an error is encountered. An error which is detected while loading a package is generally a hard user error (as opposed to a soft one). For example, a file not being found that was listed in "sound_names.txt" (or "texture_names.txt" or "mesh_names.txt") is a hard user error. Also, if the "get_tex" function of a (Lua) block script returns a texture name that does not exist, this is a hard user error.

1.11.3 Soft User Errors

Soft user errors, like hard user errors, are also caused by bad data given to the engine. Soft user errors are often more difficult to fix than hard ones. When the program encounters a soft user error, the program will exit if and only if the environment variable "engine.exit_on_error" is set to true. End users should play the game with engine.exit_on_error set to false, whereas developers should set this to true to help to find bugs.

1.12 Lua-to-C API's

There are several Lua APIs that various Lua scripts can call. These API functions are implemented on the C++ side of this program. Here are all the Lua APIs:

- Chunk Generation API
- Initialization API
- Game API

1.12.1 Chunk Generation API

The Chunk Generation API can only be used by

- Block (Lua) Scripts (WorldNodes/Nodes). More specifically, only by the "main" function of Block Lua Scripts.
- Helper Functions for Chunk Generation (WorldNodes/Helpers).

Functions part of the Chunk Generation API can optionally start with gen_

For example, a function in the Chunk Generation API is `set_pos`, but you can also write it `gen_set_pos`. The reason why we do not require the `gen_` prefix is that chunk generation code might very well be the vast majority of your package. So, we do not want to make writing this code tedious.

1.12.2 Initialization API

The Initialization API is only available when a package is being loaded. For example, a moving entity specified by the Lua script dragon.lua might call functions in this initialization API to set various parameters for dragon type moving entities.

Functions part of the Initialization API start with ia_

1.12.3 Game API

The Game API can be used by

- Block Scripts (WorldNodes/Nodes).
- Basic Entites (in BasicEnts/).
- Environment Rects (in EnvRects).
- Game Lua Scripts (modules) (in Game/).

- Moving Entities (in MovingEnts/).
- Windows (in Windows/).

Functions part of the Game API start with ga_

For example, Block Scripts have access to all three APIs, but there is a catch. Functions in Block Scripts are called within different "Lua States", and only certain APIs can be used within these states. The __type_init function is called within the "initialization state", and so code in that function can call functions that are part of the Initialization API. For example, a block script could contain the following lines:

```
function p.__type_init(id) --Called inside of the "init" Lua state.
    ia_block_new_var_i(id, "diamonds", 10) --Part of the init API.
end
```

The same block script could also contain the following lines:

```
function p.__main() --Called inside of the "chunk generation" Lua state.
    set_default_block("e") --Part of the chunk generation API.
    set_pos(7,7,7, "s")    --Part of the chunk generation API.
end
```

Finally, the same block script could also contain the following lines:

```
function p.__on_use(level, bp) --Called inside of the "game" Lua state.
    ga_set_i("player_gold", 500) --Part of the game API.
    ga_play_sound("gold")        --Part of the game API.
end
```

However, it would be a mistake to have the following code:

```
function p.__on_use(level, bp) --Called inside of the "game" Lua state.
    set_pos(9,9,9, "s") --Part of the chunk generation API. WRONG!
end
```

Chapter 2

Textures, Meshes, and Sounds

2.1 Textures

The directory of your package has a subdirectory called "Textures". In that directory, there must be a file called "texture_names.txt". This file lists the textures files that are part of the package, and assigns a name to each one.

If a name already exists in the system, then the old texture with that name will be replaced with the new one with that name.

Here is an example of what the file "texture_names.txt" can look like:

```
# Here are some textures.
crosshair    a  cool_crosshair.tga
block_grass  _  grass.jpg
block_iron   _  FromDad/iron.png
```

Empty lines, or lines that start with "#", are comment lines. Every other line should have exactly 3 strings. The first is the NAME of the texture. This is how the rest of the game will refer to the texture. The third string is the FILENAME. This should be a path relative to the directory that contains the "texture_names.txt" file. For example, in the example we have given, the file "grass.jpg" must be in the same directory as "texture_names.txt". The second string tells whether the texture has alpha (a) or does not (_).

- If the texture has alpha, the file type must be either a ".tga" or ".png".
- If the texture does *not* have alpha, the file type must be ".jpg" or ".png".

It is probably a good idea for textures to always have a width that is a multiple of 4. The game expects some textures to have alpha and others to not.

2.2 Meshes

The file Meshes/mesh_names.txt is a list of triples. The first element of the triple is the name of the mesh. This is how the rest of the system refers to the mesh. The second element is the texture name (listed in Textures/texture_names.txt). The third element is the file path of a wavefront.obj file relative to the Meshes directory. Here is an example of what Meshes/mesh_names.txt might look like:

```
health_10              health                medium_box.obj
health_25              health                large_box.obj
```

Here we see that there are two meshes, named "health_10" and "health_25". Both meshes use the texture with the name "health". One mesh uses the wavefront.obj file medium_box.obj whereas the other mesh uses the wavefront.obj file large_box.obj.

All mesh files must be of the format "wavefront.obj". See the Internet for this file format specification. Note: technically the file format supports some weird things, but the Fractal Block World program only supports the basic stuff.

2.3 Sounds

Sounds are declared in the file "Sounds/sound_names.txt". This is similar to "Textures/texture_names.txt" and to "Meshes/mesh_names.txt". The file "sound_names.txt" might look like:

```
bullet   bullet.ogg
laser    laser.ogg
song1    my_favorite_song.mp3
```

The first string is the NAME of the sound. The second string is the FILENAME of the sound file. The program supports the following types of files:

- .ogg files (Ogg Vorbis).
- .mp3 files.

2.4 Font

You can also change the font that the game uses. There are basically two ways to do this: One way is to specify a font texture atlas together with the file `Font/tex_font_desc.txt` which describes where the individual characters are located within the atlas. The other way is to have an image for each character. It is beyond the scope of this manual to describe this here, but see the guide on the game's website on Packages → All Guides → Font Guide.

Chapter 3

Block Lua Scripts Part 1: Intro and Some Chunk Creation

This chapter will discuss the basics of creating the geometry of your own world. Specifically, we will mainly discuss how to create the chunk generation aspects of "Block Lua Scripts". When a block is expanded into a chunk, the `__main` function of a Block Lua Scripts is called. These Block Lua Scripts are found in the "WorldNodes/Nodes" directory.

The `__main` function of these Block Lua Scripts have access to an API intended for the creation of chunks: the **Chunk Generation Lua-to-C API**. We will partially explain that API in this chapter, and continue the discussion in Chapter 4.

The `__main` functions of a Block Lua Script also has access to files in

"WorldNodes/Helpers".

In particular, the file

"base/WorldNodes/Helepers/std.lua"

has many usual functions that can be used from these Lua scripts. We describe the functions in this "std.lua" file in Chapter 8.

3.1 The WorldNodes Directory

This chapter will be concerned with the WorldNodes subdirectory of your package.

3.1.1 WorldNodes/StartingConfig

When a player creates a new game, he selects which package to use. After that, he selects his starting configuration. These starting configurations are specified in the `WorldNodes/StartingConfig` directory. There should be one .txt file for each starting configuration.

Let's say, for example, that there are two starting configurations. Then the directory WorldNodes/StartingConfig should contain the files `1.txt` and `2.txt`, and the following could be the contents of `1.txt`:

```
description         "Default Starting Configuration"
root_node           block_cave_world_top
player_offset       7.5 3 7.5
chunk_path          778_777
```

The line starting with `description` specifies the name of the starting configuration, and that will appear in the menu when the player is selecting his starting configuration.

The line starting `root_name` specifies the block type of the root of the world. That is, the world is a tree of chunks. The root chunk of the tree is created first, and this process is determined by the name of the block type of the root. In the example above, the following file must exist:

```
WorldNodes/Nodes/block_cave_world_top.lua
```

Note that all block scripts in WorldNodes/Nodes must start with `block_`. The line starting `player_offset`" specifies where the player starts within his starting chunk. The offset should be a triple (x,y,z) such that x,y,z are all between 0.0 and 16.0.

Finally `chunk_path` specifies the chunk path of the chunk where the player initially spawns. The format of the chunk path is a list of triples of hex characters (for x,y,z) separated by underscores (with the exception that the empty path is `EMPTY_PATH`). To specify the chunk path, the easiest way is to play the game and fly to the chunk you would like the starting position to be. Then open the console (press $\sim$) and enter the command "path dump". This will output the chunk path of your current location to `Output/path.txt`. Open that file and go to the line that starts `chunk_path`. You can then copy that line into the starting position file.

If you add the line

```
is_default
```

to a starting configuration file, the configuration will be set as the "default starting configuration".

You also create parameters that the user can set when creating a new game. To prevent this document from becoming too long, we direct the reader to the guide about "mutators" on the game's website: Packages $\rightarrow$ All Guides $\rightarrow$ Mutators.

3.1.2 WorldNodes/Nodes

Every block in the world has a type, which is represented by a string. This string is the same as the name of the Block Lua Script for the block, without the .lua at the end. When a block needs to turn into a chunk, the __main function of the appropriate Block Lua Script is called in the WorldNodes/Nodes directory. These scripts are "Lua Modules". Each script must end with ".lua".

For example, suppose there is a block of type "block_grass". Then, when the block needs to get subdivided into a chunk, the "main" function in the script

WorldNodes/Nodes/block_grass.lua

will be called. In general, if a block type is "X", then its associated chunk generation __main function is the file "WorldNodes/Nodes/X.lua". Recall that all block scripts must start with "block_". Here is what the file "block_grass.lua" might look like:

```
-- Block type: "block_grass".
-- (Comment lines start with "--").

function p.get_is_solid()
    return true
end

function p.get_tex()
    return "green_dark"
end

function p.main()
    set_default_block("block_r_green")
    for x = 0,15 do
        for y = 0,15 do
            z = 15
            set_pos(x,y,z,"block_grass2")
        end
    end
end
```

When a grass type block is expanded into a chunk, it is composed of solid block_r_green blocks, except for the top layer which consists of block_grass2 blocks. This example will be explained more in the section "The 3 Necessary Functions".

Note: the blocks defined in all packages that the current package depends on are also available. For example, if the current package is called "myworld" and it depends on the package "forestworld", and if the file

Data/Packages/forestworld/WorldNodes/Nodes/block_big_tree.lua

exists, then "block_big_tree" is a block type that is available to the "myworld" package.

For this reason, if you are planning on other people using your package as a dependency for their packages, then avoid common names for block types. That is, "block_grass.lua" is a poor choice for the name of a chunk generation Lua script. A better choice would be to prefix all block type names with your initials or something like this. So, if your name is Robert Paulson, then you could name the grass file "block_rp_grass.lua".

3.1.3 WorldNodes/Helpers

You can define helper functions that can be used by any __main function of Block Lua Scripts. When a package is first loaded, all the scripts in the directory "WorldNodes/Helpers" will be read. Specifically, a "lua state" is created by processing all these scripts (but the code in the functions in these scripts is not executed). Then, when the __main function of a Block Lua Script is executed, functions defined in the "Helpers" directory can be used.

For example, suppose there is a file called

WorldNodes/Helpers/my_first_helpers.lua

and it looks like this:

```
function p.put_iron_in_middle()
    set_pos(7,7,7,"block_iron")
end
```

Now the main function of any chunk generation script can call the function

my_first_helpers.put_iron_in_middle()

and the result will be to set the block at position (7,7,7) of the chunk to be of type "block_iron".

Note: the helper functions defined in all packages that the current package depends on are also available. So, just like what was said about the names of chunk generation scripts in the "WorldNodes/Nodes" directory, if you want others to create packages which depend on your own package, the helper functions that you define should probably by prefixed with something unique. So it would be better for "my_first_helpers.lua" to be called "rp_my_first_helpers.lua" instead, if your name is Robert Paulson for example. But again, it is probably better for user created packages to only depend on the "base" package.

3.2 Block Naming Conventions

Some blocks are solid and are subdivided into 16 by 16 by 16 blocks of the same type. For example, consider the following chunk generation file "block_r_concrete.lua":

```
function __get_is_solid() return true end
function __get_tex() return "block_concrete" end
function __main() set_default_block("block_r_concrete") end
```

When a type "block_r_concrete" block is subdivided, it turns into 16 by 16 by 16 smaller "block_r_concrete" blocks. For organization purposes, I would recommend including the "r_" (for "recursive") in the name .

It is also convenient to have a file called "block_s.lua" ("s" for "solid", and it is easy to type). The file "block_s.lua" should be as follows:

```
function __get_is_solid() return true end
function __get_tex() return "block_default" end
function __main() set_default_block("block_s") end
```

Indeed, in Base/WorldNodes/Nodes there is such a file "block_s.lua".

Or you could call it "solid", totally up to you. It also makes sense to have a file called "block_e.lua" ("e" for "empty", and it is easy to type). The file "block_e.lua" should be as follows:

```
function __get_is_solid() return false end
function __get_tex() return "" end
function __main() set_default_block("block_e") end
```

In Base/WorldNodes/Nodes there is such a file "block_e.lua".

3.3 The 3 Necessary Functions

Consider the file "WorldNodes/Nodes/block_grass.lua" presented in the section about the directory "WorldNodes/Nodes". This lua script is executed whenever a "block_grass" type block needs to be subdivided to become a chunk. There are 3 functions defined in "block_grass.lua", and these 3 functions must be defined in every Block Lua Script.

3.3.1 Function #1: p.__get_is_solid

The first function is the function "p.__get_is_solid":

```
function p.__get_is_solid()
    return true
end
```

This function is called when the package is first loaded, NOT when a block of type "grass" is being subdivided into a chunk. And so, this function does NOT have access to the chunk generation API. This function should return either "true" or "false". If true, then the block is solid and it has a texture associated to it. If false, then the block is empty (the player can move through it) and it has no texture associated to it.

Right now solid means both physically solid (the player cannot move through it) and visibly solid (the player cannot see through it). In the main Fractal Block World game (Xar) there are some visibly invisible but physically solid blocks and visa versa. Later we will talk about how to describe such blocks which are physically solid but not visibly solid or visa versa.

3.3.2 Function #2: p.__get_tex

The second function is "p.__get_tex":

```
function p.__get_tex()
    return "green_dark"
end
```

Like "p.__get_is_solid", this function is called *once* when the package is first loaded. It should return a string which is the name of the texture associated to the block type. If "p.__get_is_solid" returns false, then the "p.__get_tex" function should either not be defined or should return the empty string, like this:

```
function p.__get_tex()
    return ""
end
```

3.3.3 Function #3: p.__main

The third and most important function that must be defined in each chunk generation lua script (block script) is "p.__main". Again here is the main function in our example "block_grass.lua":

```
function p.main()
    set_default_block("block_r_green")
    for x = 0,15 do
        for y = 0,15 do
            z = 15
            set_pos(x,y,z,"block_grass2")
        end
    end
end
```

Unlike "p.__get_is_solid" and "p.__get_tex", this "p.__main" function is called each time a block of type "grass" is subdivided into a chunk. The first thing this main function does is to call the built in function "set_default_block". This function will be described soon. Next, the function has two nested "for" loops with the effect of setting the top block layer of the chunk to be "block_grass2" type blocks. The rest of the blocks in the chunk are of type block_r_green. The "set_pos" function will also be described soon.

There are various functions which can be called from the main function: Lua functions built into the language, functions defined in scripts in WorldNodes/Helpers,

and functions in the Chunk Generation Lua API. For the rest of the chapter we will describe part of the Chunk Generation Lua API. The rest of that API will be covered in the chapter Chunk Generation Lua Scripts Part 2.

3.3.4 What Does the "p." Mean?

When the Lua "module" X.lua is loaded into the program, the following two lines will be prepended to X.lua:

```
X = {}
local p = X
```

The modified file is then loaded into a Lua state L (that possibly other modules have been loaded into). This results in the Lua state L having a new global table with the name "X". If the file "X.lua" defined a function "p.foo", then in the lua State L, the (global) table "X" will have the member "foo".

It may not be wise to try to maintain state in a Lua module using a global Lua variable. It is better to use functions like "get_i" and "set_i" which modify an environment variable maintained by the engine. These functions are part of the Game Lua-to-C API.

3.3.5 Omitting the "block_" prefix for a block type

There are a small number of functions that allow you to omit the "block_" prefix when referring to block types. Here are some such functions:

- set_default_block
- set_pos
- create_rect

This is intended to make writing block scripts less tedious. Here is an example of what a block script "block_grass.lua" might look like:

```
function p.__get_is_solid() return false end
function p.__get_tex() return "" end

function p.__main()
    set_default_block("e")
    create_rect("grass", 0,0,0, 15,15,0)
    set_pos(7,7,0, "s")
end
```

Here the block type "e" is really "block_e", "grass" is really "block_grass", and "s" is really "block_s".

3.3.6 What does the double underscore mean?

Functions that start with a double underscore are called by the engine. For example, in the following script,

```
function p.__get_is_solid() return false end
...
```

the function __get_is_solid is called by the engine. If a function starts with a double underscore, it cannot have just any name.

3.4 The clear_all Function

```
void clear_all(string block_type);
```

This function clears all blocks, basic entities, moving entities, environment rectangles, etc. The default block type will become block_type.

3.5 Basic Block Functions

One of the most important tasks the __main function has to do is to specify the blocks in the chunk. For example, here is a main function that makes all the blocks be of "block_air" type, except one block which is of type "block_iron":

```
function p.__main()
    set_default_block("block_air")
    set_pos(7,7,7,"block_iron")
end
```

3.5.1 set_default_block

```
void set_default_block(string block_type);
```

You should ALWAYS call the "set_default_block" function at the beginning of the __main function. If you forget to call the set_default_block function, then the default block type will be set to a block which is purple with yellow letters which read as follows:

default block not set.

The function takes one argument, which is the block type to initially use for the 16 x 16 x 16 blocks within the chunk (as a string). Then, later calls to "set_pos" can change individual blocks.

Note: the implementation of the program stores the blocks within a chunk in a sparse way. Specifically, the default block type is stored, and every block in the chunk not of that default type is also stored.

Warning: a call to "set_default_block" does not replace any blocks created by "set_block" calls. For example, consider the following __main function:

```
function p.__main()
    set_default_block("block_air1")
    set_pos(7,7,7,"block_iron")
    set_default_block("block_air2")
end
```

You might think that the second call to "set_default_block" will replace the iron block with a block_air2 block. This is NOT the case. The final block state will be that there is an iron block at position (7,7,7), and every other block is of type block_air2. To override all blocks in the chunk to be of type "foo", use the create_rect("foo", 0,0,0, 15,15,15) function or the clear_blocks("foo") function described in the next section.

3.5.2 clear_blocks

```
void clear_blocks(string block_type);
```

This function will remove all blocks from the chunk and replace them with blocks of the type block_type. Calling this function is more efficient than calling create_rect(block_type, 0,0,0, 15,15,15). See also the function clear_all, which not only clears all blocks but clears all basic entities, moving entities, environment rectangles, etc.

3.5.3 set_pos

```
void set_pos(int x, int y, int z, string block_type);
```

The "set_pos" function is used to change an individual block. It is the most commonly used function.

In our example,

```
function p.__main()
    set_default_block("block_air")
    set_pos(7,7,7,"block_iron")
end
```

the "set_pos" function is used to set the block at position (7,7,7) to be of type "block_iron". The coordinates of a block are always (x,y,z) where x,y,z are integers between 0 and 15 inclusive.

In the example

```
function p.__main()
    set_default_block("block_air")
    set_pos(7,7,7,"block_iron")
    set_pos(7,7,7,"block_grass")
end
```

the position (7,7,7) is initially set to have type "block_air", then it is set to be of type "block_iron", and finally it is set to have type "block_grass".

3.5.4 get_pos

```
string get_pos(int x, int y, int z);
```

Theoretically, by keeping track of which functions you call from the main function, you should be able to determine the block type of any position within the chunk. However, to make life easier, the function "get_pos" is provided for this purpose. This function returns the block type of the specified block position. For example, consider the following:

```
function p.__main()
    set_default_block("block_air")
    set_pos(7,7,7,"block_iron")
    local block_type = get_pos(7,7,7)
end
```

The variable "block_type" is set to the string "block_iron".

Note: Consider the following modified example:

```
function p.__main()
    set_default_block("air")
    set_pos(7,7,7,"iron")
    local block_type = get_pos(7,7,7)
end
```

Then the variable "block_type" will still be set to "block_iron". That is, there will be the "block_" prefix.

3.6 Pseudo Random Functions

Chunks can be generated in a pseudo random fashion. The seed is set by calling the function "srand". A pseudo random float is obtained by calling "randf". A pseudo random int is obtained by calling "randi".

3.6.1 srand

```
void srand(int seed);
```

This function sets the pseudo random seed. Note: just before the chunk generation script is executed,

srand(seed_normal())

is called. That is, the seed is set using the chunk path of the chunk.

In general, you can call functions to get the chunk generation input (described soon) and use that to generate your own pseudo random seed, which you then pass to srand. There are also several helper functions, like "seed_normal", "seed_nearby", etc for creating a seed from the chunk generation input. Note: this srand function is not the same as the one in the C programming language.

3.6.2 randf

```
float randf();
```

The "randf" function pseudo randomly returns a float between 0.0 and 1.0. The following main function describes a chunk that has steel in the middle with an 80% probability, and has iron with a 20% probability:

```
function p.__main()
    set_default_block("block_air")
    if (randf() < 0.8) then
        set_pos(7,7,7,"block_steel")
    else
        set_pos(7,7,7,"block_iron")
    end
end
```

Note that these are pseudo random functions. So if you visit this chunk, then go far away and come back, the chunk will be generated again in the same way it was generated before. So if there was steel in the middle before, there will be steel in the middle again.

However, if there are two chunk locations with the same block type, then although the same chunk generation script will be executed, the pseudo random seed for the chunk, given by seed_normal(), will probably be different. So the chunks would look different.

3.6.3 randi

```
int randi(int min_value, int max_value);
```

The "randi" function returns a pseudo random int between min_value and max_value inclusive. The following main function describes a chunk with a single iron block at a random position:

```
function p.__main()
    set_default_block("block_air")
    local x = randi(0,15)
    local y = randi(0,15)
    local z = randi(0,15)
    set_pos(x,y,z,"block_iron")
end
```

3.7 Getting Chunk Generation Input

```
int    get_level();
int    get_input_path_length();
PATH   get_input_path();
```

```
BTS    get_input_path_bts();
string get_input_path_bt(int level);
string get_input_adj_bt(int dx, int dy, int dz);
string get_input_parent_adj_bt(int dx, int dy, int dz);
bool   get_input_path_block_var_exists(int level, string var, string type);
bool   get_input_path_block_b(int level, string var);
int    get_input_path_block_i(int level, string var);
float  get_input_path_block_f(int level, string var);
Vector get_input_path_block_v(int level, string var);
string get_input_path_block_s(int level, string var);
```

The functions get_level and get_input_path_length are identical, just with different names. That is, get_level returns the level that the chunk is on, which happens to be the same as the length of the path sequence of the chunk.

In order to generate a chunk, the chunk generation script is allowed access to the following:

1) the path PATH of the chunk from the root of the chunk tree,
2) the list BTS of block types of the chunks in that path, and
3) the block types of all the chunks in the 5x5x5 region surrounding the chunk.
4) the block types of all the chunks in the 3x3x3 region surrounding the parent chunk.
5) the block variables for the chunk being generated and also the parent of this chunk.

This data can be used to create a seed for "srand", although some built in functions like "seed_normal" and "seed_nearby" do this for you already.

Note: PATH and BTS are arrays that are zero indexed. The list BTS of block types is 1 longer than the path PATH of the chunk in the chunk tree (the root of the chunk tree has a block type but no position from its parent, because it has no parent). That is, BTS[0] is the block type of the root of the chunk tree, and PATH[0] is the offset of the *second* chunk (in the path from the root) from the *first* (the root chunk). If L is the length of PATH, then BTS has length L+1 and BTS[L].name is the block type of the chunk that is being generated.

A call to get_input_path_length() returns the length L of PATH. This L is the same as the result of calling get_level(). That is, L is the level of the chunk. A call to get_input_path() returns PATH. Then (PATH[0].x, PATH[0].y, PATH[0].z) is the first element of the path. Here PATH[0].x is an integer. A call to get_input_path_bts() returns BTS. The following code prints all this input data.

```
function p.__main()
    set_default_block("block_air")
```

```
    len = get_input_path_length()

    print("Printing path of chunk from root (chunk path).")
    PATH = get_input_path()
    for i = 0,len-1 do
        pos = PATH[i]
        print("Position:")
        print(tostring(pos.x))  --pos.x is an integer.
        print(tostring(pos.y))
        print(tostring(pos.z))
    end

    print("Printing the types of blocks in this path.")
    BTS = get_input_path_bts()
    for i = 0,len do --Notice this is len, not len-1
        block_type = BTS[i].name  --this is a string.
        print(block_type)
    end
end
```

If you only want to get a single block type from BTS, you can use the get_input_path_bt function:

```
function p.__main()
    set_default_block("block_air")

    --bt stands for "block type".
    local bt_of_chunk  = get_input_path_bt(level)
    local bt_of_parent = get_input_path_bt(level-1)
    -- ...
end
```

That code gets the block type of the chunk being generated, and also the block type of the parent chunk.

The function get_input_adj_bt gets the block type of a chunk in one of the 5x5x5 nearby chunks. For example, consider the following block with the script generation file WorldNodes/Nodes/dandelion.lua. The code is such that the dandelion only grows on top of a grass block:

```
function p.__get_is_solid() return false end
function p.__get_tex() return "" end

function p.__main()
    set_default_block("block_air")
    below_type = get_input_adj_bt(0,0,-1)
    if below_type == "block_grass" then
        --Can actually have a dandelion.
```

```
        create_rect("block_green", 7,7,0, 7,7,7)
        create_rect("block_yellow", 6,6,6, 8,8,8)
    end
end
```

The function get_input_parent_adj_bt gets the block type of one of the chunks int the 3x3x3 surrounding region of the *parent* of the chunk being generated.

Next, let us give an example of how to use the functions to get the block variables of the current chunk and the parent of the current chunk. Here is a block script "block_forest" that has more trees the deeper you go. Every forest block has a "depth" integer variable associated to it (a block variable), and this is set by the __main function to be one more than the depth of the parent chunk:

```
function p.__get_is_solid() return false end
function p.__get_tex() return "" end

--The "depth" variable of the block
--determines how many trees it has.
function p.__main()
    set_default_block("block_e")

    --Ground.
    create_rect("block_forest", 0,0,0, 15,15,0)

    --Trees.
    local parent_depth = 0    --Will try to set now.
    local level = get_level() --Identical to "get_input_path_length".
    --Note: should make sure level-1 >= 0.
    --
    --At this point we could get the block type of the
    --parent chunk and make sure it is of type "block_forest".
    --This could be done with "local parent_bt = get_input_path_bt(level-1)".
    --However instead let us just make sure the parent chunk
    --has a depth variable.
    if( get_input_path_block_var_exists(level-1, "depth", "i") ) then
        parent_depth = get_input_path_block_i(level-1, "depth")
    end
    --Setting the depth var of the block that is the current chunk.
    local depth = parent_depth + 1
    chunk_set_i("depth", depth)

    --Creating trees.
    --The greater the depth, the more trees there is.
    local num_trees = 2 * depth
    for i = 1,num_trees do
        local x = randi(0,15)
        local y = randi(0,15)
```

```
        set_pos(x,y,0, "block_r_green") --Tree.
    end
end

--Declaring that forest type blocks have
--and integer variable called "depth".
--These variables are called "block variables".
function p.__type_init(id)
    ia_block_new_var_i(id, "depth", 0)
end
```

Note that the get_input_path_block_var_exists requires a "type string". This should be either "b" for bool, "i" for int, "f" for float, "v" for vector, or "s" for string. It only looks for a variable with the given name with the given type.

Note that the level passed to either get_input_path_block_var_exists or one of the get_input_path_block_X functions can only be either get_level() or get_level()-1.

3.8 Generating Pseudo Random Seeds

3.8.1 seed_normal

```
int seed_normal();
```

Just before the chunk generation script is executed,

srand(seed_normal())

is called automatically. This causes the pseudo random seed to be set based on the path of the chunk from the root of the chunk tree, and not on any of the block types of the chunks in the path. Also, the block types of chunks in the 5x5x5 surrounding region are ignored.

Take for example the following code. Every time the chunk (in the same location) is created, it will be created the same way. That is, it will either always have steel or always have iron.

```
function p.__main()
    srand(seed_normal())
    set_default_block("block_air")
    if (randf() < 0.8) then
        set_pos(7,7,7,"block_steel")
    else
        set_pos(7,7,7,"block_iron")
    end
end
```

Like we said, the "srand(seed_normal())" at the beginning of the main function is not necessary.

For those that are curious, here is exactly how "seed_normal" works: the program has a list L_1 of the first 100 or so prime numbers after 1,000,000. Let PATH be the chunk path of the chunk from the root of the chunk tree. Consider the list L_2 which is as follows:

PATH[0].x, PATH[0].y, PATH[0].z, PATH[1].x, PATH[1].y, ...

For each n, the program multiplies the n-th element of L_1 by the n-th element of L_2. (Once we reach the end of L_1, we loop back around). Then, the program adds all these numbers together. That number is the seed.

3.8.2 seed_nearby

```
int seed_nearby(int dx, int dy, int dz);
```

This function first calculates the path of a nearby chunk from the root of the chunk tree. Then it is as if "seed_normal" gets called, but using that path instead. For example, the following could be the main function for a forest type block:

```
function p.__main()
    srand(seed_normal())
    set_default_block("block_air")
    for i = 1,10 do
        local x = randi(0,15)
        local y = randi(0,15)
        -- Making a "tree".
        set_pos(x,y,0,"block_tree")
    end
end
```

Below a forest type block could be a block of type forest_dirt, with the following main function:

```
function p.__main()
    srand(seed_nearby(0,0,1))
    set_default_block("block_dirt")
    for i = 1,10 do
        local x = randi(0,15)
        local y = randi(0,15)
        -- Making a "tree root".
        set_pos(x,y,15,"block_tree_root")
    end
end
```

The blocks of type "tree" will be above the blocks of type "tree_root". The call to "seed_nearby" with the triple (0,0,1) makes the seed come from the chunk that is one above in the z direction.

Note: the reason for the "seed_normal" function to ignore block types is so that the "seed_nearby" function can work.

Note: a different way to accomplish this "roots below trees" example is to use the function "get_input_adj_bt(0,0,1)" in the "tree_root" chunk generation script.

3.8.3 seed_xy, seed_xz, seed_yz

```
int seed_xy();
int seed_xz();
int seed_yz();
```

Let PATH be the path of the chunk from the root of the chunk tree. PATH is a list of triples (x,y,z), where x,y,z are integers between 0 and 15 inclusive. The function "seed_xy" sets the pseudo random seed to be based on the PATH, however it ignores all z components of the triples. For example, if two chunks with the same main function as below are on top of one another, the "shafts" will line up:

```
function p.__main()
    srand(seed_xy())
    set_default_block("block_dirt")
    -- 10 shafts:
    for i = 1,10 do
        -- Creating an air shaft.
        local x = randi(0,15)
        local y = randi(0,15)
        for z = 0,15 do
            set_pos(x,y,z,"block_air")
        end
    end
end
```

3.8.4 _chop type seed functions

```
int seed_normal_chop(int num_chop);
int seed_nearby_chop(int dx, int dy, int dz, int num_chop);
int seed_xy_chop(int num_chop);
int seed_xz_chop(int num_chop);
int seed_yz_chop(int num_chop);
```

The function "seed_normal_chop" sets the pseudo random seed using the path of the chunk from the root of the chunk tree. However, it only uses an initial segment of the path. For example, if 1 is passed as the argument to "seed_normal_chop", then the last triple (x,y,z) in the path will be ignored. If 2 is passed, then the last two triples will be ignored, etc. The other functions behave similarly.

3.8.5 seed_from_last_of_type

```
int seed_from_last_of_type(string block_type);
```

Let PATH be the path of the chunk from the root of the chunk tree. Let BTS be the list of block types that occur in this path. Out of all seed functions described, "set_from_last_of_type" is the only one that uses the BTS list to generate a seed. The function works by first finding the largest index i such that BTS[i] = block_type. Then, the triples PATH[0] through PATH[i-1] inclusive are used to generate the seed. Said another way, let C be the chunk on the chunk path PATH farthest away from the root whose block type is block_type. Then the seed is obtained by calling "seed_normal" inside C.

This function can be used to create planets where all the treasure rooms within the planet, no matter how small, all have the same type of treasure. For example, suppose the block type "block_mars_like_planet" has already been created. Here is what the __main function of "block_mars_like_planet_treasure.lua" might look like:

```
function p.__main()
    set_default_block("block_dirt")

    srand(seed_from_last_of_type("block_mars_like_planet"))

    if (randf() < 0.5) then
        add_ent(7,7,7,"gold_10")
    else
        add_ent(7,7,7,"gold_20")
    end
end
```

Within a Mars like planet, either all treasure rooms will have 10 gold, or all treasure rooms will have 20 gold.

3.9 Block Variables

```
void block_set_b(int x, int y, int z, string var, bool value)
void block_set_i(int x, int y, int z, string var, int value)
void block_set_f(int x, int y, int z, string var, float value)
void block_set_v(int x, int y, int z, string var, Vector value)
void block_set_s(int x, int y, int z, string var, string value)
```

Use these to set block variables (for blocks that are within the chunk that is being generated). Note that these variables (if they are not built-in) should be created in the type_init function of the block script of the block that is being modified (see Block Type Initialization Functions).

3.10 Environment Rects

3.10.1 add_env_rect

```
void add_env_ent(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    string ent_type);
```

Another type of game entity is a rectangle (box) of blocks (which the user can possibly move through) that affects the player whenever he touches it. These are called "environment rects". Here is a main function that adds a single "death" rect in the middle of the chunk:

```
function p.__main()
    set_default_block("block_air")
    add_env_rect(6,6,6, 8,8,8, "death")
end
```

A death rect is invisible. As soon as the player touches a death rect, he immediately dies. The values for the 6 ints given to the function "add_env_rect" should all be between 0 and 15 inclusive, with min_x less than or equal to max_x, etc.

The lua script for the death environment rect can be found in

base/EnvRects/death.lua

3.11 Basic Entities

3.11.1 add_bent

```
void add_bent(int x, int y, int z, string ent_type);
```

This is the function used to add basic entities that take no other parameters. Basic entities do not move. The position of a basic entity is always a block position. There can only be one basic entity in a block position at a time.

We highly recommend users use blocks instead of basic entities. For example, a text box block instead of a text box basic entity.

Here is an example of a main function which adds a green shrink ring:

```
function p.__main()
    set_default_block("block_air")
    add_bent(7,7,7,"bent_base_ring_green")
end
```

Here are basic entity type strings, added by the "base" package, that can be passed to "add_bent" as the ent_type:

```
bent_base_save
bent_base_ring_green
bent_base_ring_green_danger
bent_base_ring_red
bent_base_ring_red_danger
bent_base_ring_pink_source
bent_base_ring_pink_dest
bent_base_ring_blue
bent_base_respawn_point
bent_base_waypoint_out_only
bent_base_picture_gato4
```

Again the package "base" should always be a dependency. If you depend on other packages, such as "xar", then you can use all the entities that they define. But it is advised to only depend on the "base" package.

3.11.2 add_bent_i

```
void add_bent_i(
    int x, int y, int z, string ent_type,
    int int_param);
```

Some basic entities take in an integer parameter when they are constructed. If you use the wrong function, add_bent_s in place of add_bent_i for example, then this can result in a bug.

3.11.3 add_bent_s

```
void add_bent_s(
    int x, int y, int z, string ent_type,
    string str_param);
```

Some basic entities take in a string when they are constructed. For these you should use the function "add_bent_s" to add them. Here are all basic entity type strings, added by the "base" package, that can be passed to "add_bent_s":

```
bent_base_txt
bent_base_waypoint
bent_base_waypoint_in_only
```

Here is an example:

```
function p.__main()
    set_default_block("block_air")

    msg = "You better have enough rockets. "
        .. "Seriously. "
    add_ent_s(7,7,7,"bent_base_txt", msg)
end
```

In the above example, ".." is the string concatenation operator (in the Lua programming language).

3.11.4 bent_set_param_i

```
void bent_set_param_i(int x, int y, int z, int new_param_value);
```

Once you add a basic entity (BEnt) to the current chunk, you can change its unique integer parameter by calling this function.

3.11.5 bent_set_param_s

```
void bent_set_param_s(int x, int y, int z, string new_param_value);
```

Once you add a basic entity (BEnt) to the current chunk, you can change its unique string parameter by calling this function.

For example, you might have the following code for the chunk main function:

```
function p.__main()
    set_default_block("e") --Empty block.
    add_bent_s(2,2,3,"bent_base_txt", "Beware of the warevulves") --I can't spell.
    bent_set_param_s(2,2,3, "Beware of the werewolves")
end
```

3.12 Moving Entities

3.12.1 add_ment

```
void add_ment(int x, int y, int z, string type);
```

This function will add a moving entity (MEnt) centered at the center of the block (x,y,z) of the current chunk. The type of the ment is specified by the type variable.

Note that another way to add a ment is with the ment_start and ment_end functions.

The scripts for moving entities are put in the MovingEnts directory.

3.12.2 ment_start

```
void ment_start(int x, int y, int z, string type);
```

With this method of adding a moving entity (MEnt), you first call the ment_start function, then call various functions (such as ment_set_b) to set parameters of the ment, then you call ment_end which actually adds the ment to the chunk.

3.12.3 ment_set_b, ment_set_i, ment_set_f, ment_set_v, ment_set_s

```
void ment_set_b(string var_name, bool value);
void ment_set_i(string var_name, int value);
void ment_set_f(string var_name, float value);
void ment_set_v(string var_name, float x, float y, float z);
void ment_set_s(string var_name, string value);
```

You call these functions after calling ment_start but before ment_end. This "set" functions will set the various parameters of a moving entity (MEnt) before it is added to the current chunk.

"b" stands for bool, "i" stands for int, "f" stands for float, "v" stands for vector, and "s" stands for string.

3.12.4 ment_end

```
void ment_end();
```

After you call ment_start and then call functions such as ment_set_b to set parameters of the moving entity, you call this function ment_end to finally add the moving entity to the chunk.

Chapter 4

Block Lua Scripts Part 2: More Chunk Creation

In the chapter Block Lua Scripts Part 1, we covered the basics of writing chunk generation Lua scripts (these are the lua scripts that get called when a block is expanded into a chunk). We also started to discuss the Chunk Generation API. We will complete the discussion of that API in this chapter.

Recall that the functions that can be called from the "p.__main" function of a Chunk Generation Script are

1) the functions built into the Lua language,

2) the functions defined in WorldNodes/Helpers, and

3) the functions defined as part of the "Chunk Generation API".

Some notes: for 1), not actually every function in the Lua language is available. To see which Lua functions are available, call dump_lua_env() from the main function of a Chunk Generation Script. The list of all available functions will be outputted to "Output/lua_env_dump.txt".

For 3), the "Chunk Generation API" is the collection of functions such as "set_pos". Many of these functions were discussed in the chapter Chunk Generation Lua Scripts Part 1. In this chapter we will discuss the rest of the functions in this API.

So far, the only functions we have seen related to *block data* are "set_default_pos", "set_pos", "get_pos", and "clear_blocks". In this chapter we will see several more. Although some of these new functions can for the most part be defined from these old functions, we provide these new functions as built-in for convenience and for speed reasons.

4.1 The Full Chunk Generation Lua-to-C API

We now list the complete "Chunk Generation Lua-to-C API":

```
//-----------------------------------------------
//              Clearing Everything
//-----------------------------------------------

//Clearing all.
void clear_all(string block_type);

//-----------------------------------------------
//           Getting "State" Variables
//-----------------------------------------------

bool   state_get_b(string var);
int    state_get_i(string var);
float  state_get_f(string var);
Vector state_get_v(string var);
string state_get_s(string var);

//-----------------------------------------------
//            Setting Chunk Variables
//-----------------------------------------------

//Setting blocks variables of the block
//that is the current chunk being generated.
void chunk_set_b(string var, bool   value);
void chunk_set_i(string var, int    value);
void chunk_set_f(string var, float  value);
void chunk_set_v(string var, Vector value);
void chunk_set_s(string var, string value);

//-----------------------------------------------
//            Setting Block Variables
//-----------------------------------------------

//Setting block variables (of blocks within the chunk).
void block_set_b(int x, int y, int z, string var, bool value)
void block_set_i(int x, int y, int z, string var, int value)
void block_set_f(int x, int y, int z, string var, float value)
void block_set_v(int x, int y, int z, string var, Vector value)
void block_set_s(int x, int y, int z, string var, string value)

//-----------------------------------------------
//            Block Type Integer Codes
//-----------------------------------------------

int    bt_str_to_code(string block_type);
string bt_code_to_str(int block_type_code);
```

```
void set_default_block_c(int block_type_code);
void clear_blocks_c(int block_type_code);
void set_pos_c(int x, int y, int z, int block_type_code);
int  get_pos_c(int x, int y, int z);
void create_rect_c(
    int block_type_code,
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z);
void create_sprinkles_c(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    float prob,
    int block_type_code);

//-----------------------------------------------
//                   Blocks
//-----------------------------------------------

//Basic block functions.
void set_default_block(string block_type);
void clear_blocks(string block_type);
void set_pos(int x, int y, int z, string block_type);
string get_pos(int x, int y, int z);

//More block functions.
void create_rect(
    string block_type,
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z);
void create_sprinkles(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    float prob, string block_type);

//Exotic block functions: Mazes.
void maze_start();
void maze_add_vertex(int x, int y, int z);
void maze_add_edge(
    int x1, int y1, int z1,
    int x2, int y2, int z2);
void maze_end();
bool maze_edge_open(
    int x1, int y1, int z1,
    int x2, int y2, int z2);
int maze_num_edges_from_vertex(
```

```
    int x, int y, int z);
POS maze_deepest_vertex(LIST source_vertices);

//Exotic block functions: Caves.
void caves_start();
void caves_set_5x5x5();
void caves_set_num_nodes(
    float min_nodes, float max_nodes);
void caves_set_nodes(
    float frac_large_node,
    float small_node_min_rad,
    float small_node_max_rad,
    float large_node_min_rad,
    float large_node_max_rad);
void caves_set_edges(
    float max_edge_dist,
    float frac_large_edge,
    float small_edge_min_rad,
    float small_edge_max_rad,
    float large_edge_min_rad,
    float large_edge_max_rad);
void caves_end();
bool caves_close_to_node(
    int x, int y, int z);
INFO caves_close_to_node2(
    int x, int y, int z);
bool caves_close_to_edge(
    int x, int y, int z);

//-----------------------------------------------
//                Pseudo Random
//-----------------------------------------------

//Pseudo random functions.
void srand(int seed);
float randf();
int randi(int min_i, int max_i);

//-----------------------------------------------
//                Getting Time
//-----------------------------------------------

int   get_sys_time();
float get_game_time();

//-----------------------------------------------
```

```
//        Getting Chunk Generation Input
//-----------------------------------------------

//Getting the input.
int    get_level();
int    get_input_path_length();
PATH   get_input_path();
BTS    get_input_path_bts();
string get_input_path_bt(int level);
string get_input_adj_bt(int dx, int dy, int dz);
string get_input_parent_adj_bt(int dx, int dy, int dz);
bool   get_input_path_block_var_exists(int level, string var, string type);
bool   get_input_path_block_b(int level, string var);
int    get_input_path_block_i(int level, string var);
float  get_input_path_block_f(int level, string var);
Vector get_input_path_block_v(int level, string var);
string get_input_path_block_s(int level, string var);

//-----------------------------------------------
//  Creating Seeds from Chunk Generation Input
//-----------------------------------------------

//Pseudo random seeds.
int seed_normal();
int seed_nearby(int dx, int dy, int dz);
int seed_xy();
int seed_xz();
int seed_yz();
int seed_normal_chop(int chop);
int seed_nearby_chop(int dx, int dy, int dz, int chop);
int seed_xy_chop(int chop);
int seed_xz_chop(int chop);
int seed_yz_chop(int chop);
int seed_from_last_of_type(string type);

//-----------------------------------------------
//              Environment Rects
//-----------------------------------------------

//Env rects.
void add_env_rect(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z, string type);

//-----------------------------------------------
//               Basic Entities
```

```
//-----------------------------------------------

//Basic ents (BEnts).
void add_bent(int x, int y, int z, string type);
void add_bent_i(int x, int y, int z, string type, int param);
void add_bent_s(int x, int y, int z, string type, string param);
void bent_set_param_i(int x, int y, int z, int new_param_value);
void bent_set_param_s(int x, int y, int z, string new_param_value);

//-----------------------------------------------
//                Moving Entities
//-----------------------------------------------

//Moving ents (MEnts).
void add_ment(int x, int y, int z, string type);
void ment_start(int x, int y, int z, string type);
void ment_set_b(string key, bool value);
void ment_set_i(string key, int value);
void ment_set_f(string key, double value);
void ment_set_v(string key, float x, float y, float z);
void ment_set_s(string key, string value);
void ment_end();

//-----------------------------------------------
//                Block Types
//-----------------------------------------------

//Getting information about block types.
bool bt_get_is_solid_physically(string block_type);

//-----------------------------------------------
//                Virtual Chunks
//-----------------------------------------------

CLASS get_vchunk_data(int chop, int dx, int dy, int dz);
CLASS chunk_bbox_to_vchunk(int chop);

//-----------------------------------------------
//                Perlin Noise
//-----------------------------------------------

CLASS get_perlin_data_xyz(     int chop, int salt);
CLASS get_perlin_data_xyz_tail(int chop, int salt);
CLASS get_perlin_data_xy(      int chop, int salt);
CLASS get_perlin_data_xy_tail( int chop, int salt);
```

```
void cache_perlin_data_xyz(int handle, LIST seeds);
void cache_perlin_data_xy( int handle, LIST seeds);

float perlin_noise_xyz(int handle, float x, float y, float z);
float perlin_noise_xy( int handle, float x, float y);

//-----------------------------------------------
//           Xar Chunk Generation
//-----------------------------------------------

void create_xar_chunk(string bt);

//-----------------------------------------------
//                 Deprecated
//-----------------------------------------------
//Do not use these functions.

//Blue type.
void set_blue_type_up();
void set_blue_type_down(int x, int y, int z);
void set_blue_type_terminal(int x, int y, int z);

//-----------------------------------------------
//                 Debugging
//-----------------------------------------------

//Debugging functions.
void print(string str);
void exit();
void dump_lua_env();
```

We will only discuss the functions not already covered in the chapter "Block Lua Scripts Part 1".

4.2 Getting "State" Variables

```
bool   state_get_b(string var)
int    state_get_i(string var)
float  state_get_f(string var)
Vector state_get_v(string var)
string state_get_s(string var)
```

You can use these functions to get the values of certain global variables (variables defined in the `globals.txt` file of the package).

You should use these with care. It is intended that these are set when a new game is created and then are never modified afterward. Note that depending

on how you write your worldgen code, if the user changes one of these variables in the middle of the game, it might totally obliterate all the areas that they explored.

Let us give an example of how this works. In the `globals.txt` file of your package, add the line

```
b worldgen.state.lots_of_rockets true
```

This adds a certain boolean variable. The variables that worldgen has access to must all start with `worldgen.state`. To be clear, this variable in the engine's variable envivonment is

`game.globals.worldgen.state.lots_of_rockets`.

Then, in the `__main` function of a block script, you can call

```
local value = state_get_b("lots_of_rockets")
```

to get the value of this bool.

4.3 Getting and Setting Chunk Variables

The following functions can be used to set the values of block variables for the block that is the current chunk being generated:

```
void chunk_set_b(string var, bool   value);
void chunk_set_i(string var, int    value);
void chunk_set_f(string var, float  value);
void chunk_set_v(string var, Vector value);
void chunk_set_s(string var, string value);
```

For example, the following script "block_stone.lua" sets the "health" variable of the chunk to 78 when the chunk is generated:

```
function p.__get_is_solid() return true end
function p.__get_tex() return "stone" end

--The "health" variable of the block
--determines how many blocks are generated.
function p.__main()
    set_default_block("block_s")
    chunk_set_i("health", 78);
end

--Registering that stone blocks
--have a "health" variable.
--We are setting the default value to 100.
function p.__type_init(id)
    ia_block_new_var_i(id, "health", 100)
end
```

Before the stone block was expanded into a chunk, its health variable could have been something different. Indeed, its heath variable would have been 100 if no other changes were made, because that is the default value we specified.

More common than setting chunk variables is *getting* chunk variables. This can be accomplished with the `get_input_path_block_XXX` functions we have already described. Let us give an example. The following is the script "`block_concrete.lua`" for a block that has a variable called health. Health is an integer and it is intended to be between 0 and 100 inclusive. The amount of health specifies the fraction of blocks inside the chunk that are solid:

```
function p.__get_is_solid() return true end
function p.__get_tex() return "block_room" end

--The "health" variable of the block
--determines how many blocks are generated.
function p.__main()
    set_default_block("block_e") --Empty block.
    local level = get_level() --Identical to "get_input_path_length".
    local health = get_input_path_block_i(level, "health")
    --
    local frac = health / 100.0
    for x = 0,15 do
    for y = 0,15 do
    for z = 0,15 do
        if( randf() < frac ) then
            set_pos(x,y,z, "block_s") --Solid block.
        end
    end end end
end

function p.__type_init(id)
    ia_block_new_var_i(id, "health", 100)
end
```

4.4 Block Type Integer Codes

```
int    bt_str_to_code(string block_type);
string bt_code_to_str(int block_type_code);
```

For technical reasons, you might want to use an integer in the place of a string to describe a block type. When the engine loads the game, it associates an integer "code" to every block type that was found in the package. You can use these functions to convert back and forth between them.

For example, consider the following code from a block script:

```
function p.__main()
```

```
    set_default_block("block_e") --Empty.
    local block_stone_int = bt_str_to_code("block_stone")
    set_pos_c(7,7,7, block_stone_int)
```

Note that the API functions that create blocks have versions with `_c` at the end. These modifications take a block type integer code instead of a block type string. Here are the block type integer code versions of other API functions:

```
void set_default_block_c(int block_type_code);
void clear_blocks_c(int block_type_code);
void set_pos_c(int x, int y, int z, int block_type_code);
int  get_pos_c(int x, int y, int z);
void create_rect_c(
    int block_type_code,
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z);
void create_sprinkles_c(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    float prob,
    int block_type_code);
```

4.5 More Block Functions

4.5.1 create_rect

```
void create_rect(
    string block_type,
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z);
```

Calling this function creates a box of blocks, from the block at position (min_x,min_y,min_z) to the position (max_x,max_y,max_z). Calling

create_rect(type, 0,0,0, 15,15,15)

will replace all blocks in the chunk with the block of the specified type.

Note: Calling create_rect is faster than calling set_pos once for each block in the box.

4.5.2 create_sprinkles

```
create_sprinkles(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    float prob, string type);
```

Calling this function is equivalent to the following:

```
for x = min_x,max_x do
    for y = min_y,max_y do
        for z = min_z,max_z do
            if( randf() < prob ) then
                set_pos(x,y,z, type)
            end
        end
    end
end
```

4.6 Exotic Block Functions: Mazes

These maze creation functions are basic. These are intended for basically a hello world purpose. We recommend you create your own maze creation functions in WorldNodes/Helpers if you are making a significantly complicated world.

4.6.1 Creating a Maze

You can create mazes inside the chunk being generated. You create the maze by first calling "maze_start()", and then you call some functions to set up the creation of the maze. You then call "maze_end()" to finish creating the maze.

```
void maze_start();
void maze_add_vertex(int x, int y, int z);
void maze_add_edge(int x1, int y1, int z1, int x2, int y2, int z2);
void maze_end()
```

After calling "maze_start()", you first add vertices. You must add all vertices before adding any edges. You add a vertex by calling

maze_add_vertex(x,y,z).

You specify the x,y,z coordinates of a block within the chunk.

To add an "edge" between two vertices, you call

maze_add_edge(x1,y1,z1, x2,y2,z2)

where (x1,y1,z1) is the position of one vertex and (x2,y2,z2) is the position of the other.

Then you call "maze_end()". This will trigger the engine to assign a random weight to each edge, between 0.0 and 1.0. Then a minimal spanning tree will be formed. The maze consists of all vertices and all edges in this minimal spanning tree. Because the result is a tree, there are no "cycles".

4.6.2 Basic Querying of the Maze

```
bool maze_edge_open(x1,y1,z1, x2,y2,z2);
```

Once the maze has been created (after calling "maze_end"), to determine whether an edge is in the minimal spanning tree, call

maze_edge_open(x1,y1,z1, x2,y2,z2)

where (x1,y1,z1) is the position of one vertex and (x2,y2,z2) is the position of the other. It returns true iff the edge is in the minimal spanning tree.

4.6.3 Example

Here is an example of the main function of a chunk generation script which creates a maze:

```
function p.__main()
    set_default_block("block_e") --Empty block.

    --The vertices and the edges of the maze
    --will be solid (of type "block_s").
    --Everything else will be empty (of type "block_e").
    --This way if you look at the chunk from the
    --distance, you can easily see the maze.

    --Start creating the maze.
    maze_start()

    --Adding vertices to the maze.
    --The first for loop starts x at 0 and
    --goes to 15 inclusive, stepping by 2
    --each time.
    for x = 0,15,2 do
    for y = 0,15,2 do
        maze_add_vertex(x,y,7)
        set_pos(x,y,7,"block_s")
    end
    end

    --Adding edges to the maze.
    --Only some of these will remain
    --in the final minimal spanning tree.
    for x = 0,15,2 do
    for y = 0,15,2 do
        if (x+2 <= 15) then
            maze_add_edge(x,y,7, x+2,y,7)
```

```
        end
        if (y+2 <= 15) then
            maze_add_edge(x,y,7, x,y+2,7)
        end
    end
    end

    --Finish creating the maze.
    maze_end()
    --The graph (minimal spanning tree)
    --for the maze has been created.

    --An edge (between two vertices) is called "open"
    --if it is in the final minimal spanning tree.
    for x = 0,15,2 do
    for y = 0,15,2 do
        if (x+2 <= 15) then
            if maze_edge_open(x,y,7, x+2,y,7) then
                set_pos(x+1,y,7,"block_s")
            end
        end
        if (y+2 <= 15) then
            if maze_edge_open(x,y,7, x,y+2,7) then
                set_pos(x,y+1,7,"block_s")
            end
        end
    end
    end
end
```

4.6.4 More Querying of the Maze: Part 1

```
int maze_num_edges_from_vertex(int x, int y, int z);
```

The function "maze_num_edges_from_vertex" tells you the number of open edges incident to the given vertex. This is useful for determining which vertices are "dead ends".

Note: you could also probably use "get_pos" for the same purpose. Note that you can also use "get_input_adj_bt" within the chunk generation script for the block that occupies a vertex of the maze.

Here is code that you can add to the example above that colors the dead ends black:

```
    --Coloring dead ends black.
    for x = 0,15,2 do
    for y = 0,15,2 do
```

```
        if maze_num_edges_from_vertex(x,y,7) == 1 then
            set_pos(x,y,7,"block_r_black")
        end
    end
    end
```

4.6.5 More Querying of the Maze: Part 2

```
POS maze_deepest_vertex(LIST source_vertices);
```

To use the function "maze_deepest_vertex", you first create a list of "source vertices". These must all be vertices of the maze. You pass these to the function, and it will return the position of the vertex that is farthest away from any of the source vertices. You can add the following code to the main example of this section to color the deepest vertex brown:

```
--Making the source positions green.
set_pos(0,0,7,"block_r_green")
set_pos(15,15,7,"block_r_green")

--Putting the source positions into a list.
local sources = {}
sources[1] = {x=0, y=0, z=7}
sources[2] = {x=15, y=15, z=7}

--Making the deepest vertex brown.
local pos = maze_deepest_vertex(sources)
set_pos(pos.x, pos.y, pos.z, "block_r_brown")
```

4.7 Exotic Block Functions: Caves

These cave creation functions are basic. These are intended for basically a hello world purpose. We recommend you create your own cave creation functions in WorldNodes/Helpers if you are making a significantly complicated world.

There are also functions for creating caves. These are "stick and ball" caves, meaning there are balls (nodes) connected by tubes (edges). Adjacent chunks that use the same cave creation code will have caves that connect with each other in the expected way.

4.7.1 Cave Creation

```
void caves_start();
void caves_set_5x5x5();
void caves_set_num_nodes(
    float min_nodes, float max_nodes);
```

```
void caves_set_nodes(
    float frac_large_node,
    float small_node_min_rad,
    float small_node_max_rad,
    float large_node_min_rad,
    float large_node_max_rad);
void caves_set_edges(
    float max_edge_dist,
    float frac_large_edge,
    float small_edge_min_rad,
    float small_edge_max_rad,
    float large_edge_min_rad,
    float large_edge_max_rad);
void caves_end();
```

To start creating the caves, you call "caves_start()". There are then a variety of functions you can call to create the stick and ball style caves.

By default, all sticks and balls in the surrounding 3x3x3 chunks will be created. The engine accomplishes this by having a way to create all sticks and balls within any given chunk (using the chunk path of the chunk as input to the pseudo random number generator). So if there is a node in one chunk and a node in an adjacent chunk, and if there is an edge between the two nodes, we can carve out a shaft surrounding the edge that goes between the two nodes.

If you have a node in one chunk A and then a node in a chunk B that is 2 chunks away from B, by default there is no way to have an edge from the first node to the second one. However, if you call "caves_set_5x5x5()" after "caves_start()", but before "caves_end()", then all sticks and balls in the surrounding 5x5x5 chunks will be created. (Note: the 5x5x5 mode is slower for the computer than 3x3x3 mode). Nodes that are two chunks away from each other can then be connected by edges.

The function "caves_set_num_nodes" specifies how many nodes should be created in each chunk. You specify the min and the max number, and then for each chunk the actual number will be chosen at random between the two. Specifically, the min and max values are floats, a float is chosen at random between these two, and then the integer floor of that is used.

There are two types of nodes: small ones and large ones. You specify the radii of these two types of nodes by calling the function "caves_set_nodes". You specify the min and max radius of a small node, and then each small node will have a radius randomly picked between the min and max. You also specify the min and max radius of a large node. You also specify the fraction of nodes that are large.

Each edge has a radius (so each edge is really a tube, or cylinder). You call "caves_set_edges" to set the radii of these edges. There are two types of edges: small and large. You specify the min and max radius of small edges. You do the same for the large edges. You also specify the fraction of edges that are large versus small. Finally, you specify the max edge distance. If 3x3x3 mode

is being used and two nodes are in adjacent chunks (or the same chunk), then they will be connected by an edge iff the distance between them is less than the max edge distance. If 5x5x5 mode is being used, then the same is true but now for nodes that are in chunks that are at most 2 chunks apart.

When you are finished specifying the caves, call "caves_edge()" to finish creating the maze.

4.7.2 Querying the Caves: Part 1

```
bool caves_close_to_node(int x, int y, int z)
bool caves_close_to_edge(int x, int y, int z)
```

The next step is iterating through every block position P in the chunk to see if P is inside a node or an edge. You then "carve out" all such block positions. You call "caves_close_to_node" and "caves_close_to_edge" to see if a block position is inside a node or edge.

4.7.3 Example

Here is the main function of a chunk generation script that creates some caves.

```
function p.__main()
    --The chunk is by default solid to start with.
    set_default_block("block_s")

    --Creating the stick-and-ball data
    --structure for the caves.
    caves_start()

    --Making the cave connect
    --together nodes that are at most 2 chunks apart
    --(as opposed to 1 chunk apart).
    --Setting the 5x5x5 option makes cave creation slower.
    caves_set_5x5x5()

    --Between 2 and 3 nodes per chunk (random).
    caves_set_num_nodes(2.0,3.99)

    --Only 0.01 of nodes are large, the rest are small.
    --Small nodes have radius between 3.0 and 4.3, and
    --large nodes have radius between 17.5 and 18.0.
    caves_set_nodes(0.01, 3.0,4.3, 17.5,18.0)

    --Max dist between two nodes that can be connected
    --with an edge is 20.0 (to go beyond 16.0 for this
    --number, must call caves_set_5x5x5).
```

```
    --No edges are large.
    --Small tubes (around edges) have radius between 1.0 and 2.0.
    --Large tubes have radius between 7.0 and 8.0.
    caves_set_edges(20.0, 0.0, 1.0,2.0, 7.0,8.0)

    caves_end()
    --Now, the stick-and-ball data structure
    --for the caves has been created.

    --****************************************
    --****************************************
    --****************************************

    for x = 0,15 do
    for y = 0,15 do
    for z = 0,15 do
        local close = caves_close_to_node(x,y,z)
            or caves_close_to_edge(x,y,z)
        if close then
            --Carving out the position.
            set_pos(x,y,z,"block_e")
        end
    end
    end
    end
end
```

4.7.4 Querying the Caves: Part 2

```
INFO caves_close_to_node2(int x, int y, int z)
```

A common task it to add one item to the center of each node in the stick and ball caves. To do this, you can use the "case_close_to_node2" function which returns an object data with the following members:

```
data.close      //the result of case_close_to_node
data.which_node //which nodes is closest to
data.dist       //distance to closest node
data.is_big     //whether the nodes is closest to is "big"
```

Each node has a number. You can keep track of this so you only place one item per node. Here is a modification of the cave creation code from this section that only puts one "gold_10" item in the center of each node. This code should occur after the "caves_end()" function.

```
    --A table, whose keys are the IDs of the nodes
    --that have a power up placed in them.
```

```
    local filled_nodes = {}

    for x = 0,15 do
    for y = 0,15 do
    for z = 0,15 do
        local close_to_edge = caves_close_to_edge(x,y,z)

        local data = caves_close_to_node2(x,y,z)
        local close_to_node = data.close

        --Carving the position if need be.
        if close_to_node or close_to_edge then
            set_pos(x,y,z,"block_e")
        end

        --Adding gold in the center (for each node).
        if close_to_node then
            local which_node = data.which_node
            local dist = data.dist
            local is_big = data.is_big

            if (dist < 1.5) and
                filled_nodes[which_node] == nil
            then
                filled_nodes[which_node] = true
                add_bent(x,y,z, "bent_gold_10")
            end
        end
    end
    end
    end
```

4.8 Getting Time

```
int   get_sys_time();
float get_game_time();
```

Use the function `get_sys_time` to get the system time. Under the hood this returns `time(NULL)` (where `time` is the function defined in the C++ programming language). So, `get_sys_time` returns the number of seconds elapsed since the Unix epoch (midnight January 1, 1970). You can use this in conjunction with the `srand` function to make it so that every time you generate a chunk, it generates the chunk completely randomly (not psuedo randomly). We are not saying this is a good idea, we are just saying it is possible. Of course, parts of a chunk could be generated pseudo randomly but other parts could be completely random.

The function `get_game_time` returns the game time at the instant the request was made to generate the chunk.

4.9 Block Types

It is possible for the `__main` function of a block script to get information about any particular block type. Specifically, the following function returns whether or not the given block type is (physically) solid:

```
bool bt_get_is_solid_physically(string block_type);
```

Here is an example of a block script that creates a dandelion only if the block type of the chunk below is physically solid:

```
function p.__get_is_solid() return false end
function p.__get_tex() return "" end

function p.__main()
    set_default_block("block_e")

    --Getting the block type of the chunk below
    --the one being generated.
    local below_bt = get_input_adj_bt(0,0,-1)

    --Seeing if the block type of the chunk
    --below is physically solid.
    local below_is_solid = bt_get_is_solid_physically(below_bt);

    if( below_is_solid ) then
        --Can actually have an (Onion the cat) dandelion.
        create_rect("block_r_green", 7,7,0, 7,7,7)
        create_rect("block_r_onion_the_cat", 6,6,6, 8,8,8)
    end
end
```

Suppose you want block types to have other attributes that these `__main` functions have access to. This can be accomplished by calling functions of other blocks scripts. That is, suppose you want your block scripts to have the function "`get_is_funky`" which returns a bool. Here is how a block script can query whether or not other block types are funky:

```
function p.is_bt_funky(bt)
    local mod_name = bt
    local func_name = "get_is_funky"
    if( _G[mod_name] and
        _G[mod_name][func_name] )
    then
```

```
        --Calling the function in the block lua script.
        return _G[mod_name][func_name]()
    else
        --Either the Lua module or the function
        --could not be found.
        return false --Not funky.
    end
end
```

4.10 Virtual Chunks

```
CLASS get_vchunk_data(int chop, int dx, int dy, int dz);
CLASS chunk_bbox_to_vchunk(int chop);
```

Understanding virtual chunks is a prerequiside to understanding Perlin Noise. It is too complicated to fully describe this here, but please see the game's website. Specifically, Packages → All Guides → Virtual Chunks. That is a guide which explain what virtual chunks are and how to use them with these API functions.

Here is a simple example. Suppose, in each of the 27 chunks in the 3x3x3 region around the chunk being generated, you want to randomly place a sphere with radius 10. When you are considering one of those chunks, we call that "the virtual chunk". For example, the virtual chunk that is one above the chunk being generated has the "data" given by

`get_vchunk_data(0, 0, 0, 1)`.

That is, we set chop = 0 because the virtual chunk has the same size as the chunk being generated. If we set chop = 1, the virtual chunk would be twice as wide as the chunk being generated. If we set chop = 2, the virtual chunk would be four times as wide as the chunk being generated, etc. We set $(dx, dy, dy) = (0, 0, 1)$ to go up in the z direction by one.

The function `get_vchunk_data` returns a table with the following members:

```
Vector min
Vector max
Vector min2
Vector max2
Vector rmin
Vector rmax
int    seed
string bt
float  scale
```

The vector `min` and `max` are the min and max positions of the virtual chunk in the coordinate system of the chunk being generated. For example, if the

virtual chunk and the chunk being generated were the same, then min would be `(0.0, 0.0, 0.0)` and max would be `(16.0, 16.0, 16.0)`.

The vectors `min2` and `max2` are just min and max divided by 16.

The vectors `rmin` and `rmax` are the coordinates of the chunk being generated in the coordinate system of the virtual chunk.

The integer `seed` can be used to procedurally generate data assocated to the vitual chunk. That is, `seed` is uniquely determined by the virtual chunk itself.

The string `bt` is the block type of the smallest chunk which contains the virtual chunk (the virtual chunk might not literally be a chunk). You probably will not need to use `bt`.

The float `scale` is the width of the virtual chunk divided by the width of the chunk being generated. For example, if chop = 4, then scale = 16.

The function `chunk_bbox_to_vchunk` is just like `get_vchunk_data` but it sets `(dx,dy,dz) = (0,0,0)` and it returns a table that only has `rmin` and `rmax`. The reason why this function exists is because it covers a common case that is used for Perlin noise.

4.11 Perlin Noise

```
CLASS get_perlin_data_xyz(     int chop, int salt);
CLASS get_perlin_data_xyz_tail(int chop, int salt);
CLASS get_perlin_data_xy(      int chop, int salt);
CLASS get_perlin_data_xy_tail( int chop, int salt);

void cache_perlin_data_xyz(int handle, LIST seeds);
void cache_perlin_data_xy( int handle, LIST seeds);

float perlin_noise_xyz(int handle, float x, float y, float z);
float perlin_noise_xy( int handle, float x, float y);
```

Make sure you already understand virtual chunks (see Section 4.10) before reading this. See also the Perlin Noise demo on the game's website. It is beyond the scope of this manual to give a full Perlin noise example here.

Consider 3D Perlin noise. There are three steps: 1) Call a function to return the "Perlin noise data". 2) Give some of this data to a "cache Perlin data" function. This causes the engine to precompute some information associated to the data you pass it. To refer to this precomputed data layer, you use an integer "handle". You can use whatever integer handle you want here. 3) You use that integer handle to call a Perlin noise function (probably hundreds of times) while generating the current chunk.

The `_tail` versions of the get Perlin Data functions just use a tail segment of the chunk's path, instead of the entire path to generate random seeds. We recommand using the `_tail` functions because they are faster. The only downside is that the world will repeat itself, but the player would have to be clever to find this.

The get Perlin data xyz functions return a table with the following members:

```
LIST seeds
Vector rmin
Vector rmax
```

The values of `rmin` and `rmax` could be instead obtained from `get_vchunk_data` or `chunk_bbox_to_vchunk`. The member `seeds` is an array of 8 integers, which are the "seeds" of the 8 corners of the virtual chunk. Do not worry about this `seeds` array: You simply pass it to `cache_perlin_data_XXX`.

The Perlin data xy functions return a similar table, but now there are only 4 elements of the seed list.

4.12 Xar Chunk Generation

```
void create_xar_chunk(string bt);
```

This will generate the chunk as if it was of the specified xar block type. That is, it will run C++ xar chunk generation code. The string bt must start with `XAR_`. Here is an example of a block type that is just like a xar small yellow flower, except there is a solid cube at the top:

```
--Let this file be called block_new_small_yellow_flower.lua

function p.__get_is_solid() return false end
function p.__get_tex() return "" end

function p.__main()
    set_default_block("e")

    --Generating the chunk as if
    --it was of type XAR_SMALL_YELLOW_FLOWER.
    create_xar_chunk("XAR_SMALL_YELLOW_FLOWER")

    --Replacing the yellow block in the yellow flower
    --with a meme block.
    for x = 0,15 do
    for y = 0,15 do
    for z = 0,15 do
        if( get_pos(x,y,z) == "XAR_SMALL_YELLOW_FLOWER_ROOM" ) then
            set_pos(x,y,z, "s") --Solid cement block.
        end
    end end end
end
```

Also, see the Xar Block Overrides Guide here

http://danthemanhathaway.com/ComputerGames/FractalBlockWorld/ReleaseMisc/Packages/

for how to replace xar block types with lua Block scripts. For example, you could have

```
XAR_SMALL_YELLOW_FLOWER
```

be replaced with the block

```
block_new_small_yellow_flower
```

that we defined above.

4.13 Debugging

4.13.1 print

```
void print(string str);
```

This will print the given string str to standard output. The following will be prepended at the start of the output line:

```
Proc world gen: CHUNK_FILENAME.lua:
```

4.13.2 exit

```
void exit();
```

This will exit the program. Before doing so, the following will be printed to standard output:

```
Proc world gen: CHUNK_FILENAME.lua: exiting program.
```

4.13.3 dump_lua_env

```
void dump_lua_env();
```

This will print to `Output/lua_env_dump.txt` all functions in the current Lua state that are available.

4.14 Deprecated functions

Do not use any of the Chunk Generation API functions in this section. One day these functions may be removed.

4.14.1 Blue Type Functions

```
void set_blue_type_up();
void set_blue_type_down(int x, int y, int z);
void set_blue_type_terminal(int x, int y, int z);
```

These functions determine the behavior when the player touches a blue ring device. Here is a summary of how blue ring devices work:

Every chunk in the chunk tree is one of 3 types: A blue UP, a blue DOWN, or a blue TERMINAL. Once you touch a blue ring, you travel. If you are in a blue UP chunk, you go up one chunk in the chunk tree (towards the root). If you are in a blue DOWN chunk, you go down one chunk (to a child specified by the current chunk). This process repeats until you reach a blue TERMINAL chunk, at which point you stop.

The x,y,z in the `set_blue_type_down` function specify which chunk you travel down to. The x,y,z in the `set_blue_type_terminal` specify the final block position of the player (within the current chunk).

These functions have been deprecated. This is because Blue Rings and Blue Ring teleportation is no longer built into the engine. Instead of using these functions, you should use their replacements that are defined in

`Data/Packages/base/WorldNodes/Helpers/base_blue_tele.lua`.

For example, from the `__main` function of a chunk generation script you would call `base_blue_tele.set_blue_type_up()` instead of `set_blue_type_up()`.

Chapter 5

Block Lua Scripts Part 3: Type Init Functions

5.1 More Block Lua Module Functions

In Section 3.3 we saw 3 functions that appear in all Block Lua Scripts:

- `__get_is_solid`
- `__get_tex`
- `__main`

Those 3 functions are mandatory. In this chapter we describe more "module" functions which can appear in these Block Lua Scripts. The functions that we describe in this chapter are similar to `__get_is_solid` and `__get_tex`.

Moreover, we will list only the functions called in the *initialization Lua state*. These functions are called by the engine once for each Block Lua script when the package is loaded.

Here is a list of such functions:

```
//-----------------------------------------------
//  The Main Initialization Function For The Block Type
//-----------------------------------------------

void __type_init(int type_id);

//-----------------------------------------------
//        Specifying If The Block Is Solid
//-----------------------------------------------

bool __get_is_solid(); //Mandatory.
bool __get_is_solid_physically();
```

```
bool __get_is_solid_move_body();

bool __get_is_solid_visibly();
bool __get_is_solid_visibly_glass();
bool __get_is_solid_visibly_water();

//-----------------------------------------------
//      Specifying The Texture Of The Block
//-----------------------------------------------

string __get_tex(); //Mandatory.

string __get_tex_x_pos();
string __get_tex_x_neg();
string __get_tex_y_pos();
string __get_tex_y_neg();
string __get_tex_z_pos();
string __get_tex_z_neg();

string __get_inv_tex_x_pos();
string __get_inv_tex_x_neg();
string __get_inv_tex_y_pos();
string __get_inv_tex_y_neg();
string __get_inv_tex_z_pos();
string __get_inv_tex_z_neg();

bool __get_partially_transparent();
```

5.2 p.__type_init

```
void __type_init(id);
```

This function is called once (when the package is being loaded) when the engine is getting information about the block type. It is intended that this function call Initialization API functions to add variables to the block type. Here is an example for a "soda machine block":

```
function p.__type_init(id)
    ia_block_new_var_i(id, "num_cans", 50)
end
```

See Chapter 16 for more about the Initialization API.

5.3 p.__get_is_solid

```
bool __get_is_solid();
```

We already described this function. It must appear in every Block Lua Script. This function returns whether on not a block is "solid". However there are several notions of solid:

- physically solid
- move body solid
- visibly solid

A chunk is physically solid iff game projectiles cannot move through the block.

A chunk is move body solid iff the player cannot move through the block.

A chunk is visibly solid iff the player cannot see through the chunk. If blocks A and B are adjacent, A is NOT visibly solid but B is visibly solid, then the game will display a square on the side of the B block that faces into the A block.

The __get_is_solid must appear in each Block Lua Script, and then the functions __get_is_solid_physically, __get_is_solid_move_body, __get_is_solid_visibly should be defined only if they return a value different than the value returned by __get_is_solid. In this way we can specify whether or not a block is physically solid, move body solid, and visibly solid.

5.4 p.__get_is_solid_physically, etc

```
bool __get_is_solid_physically();
bool __get_is_solid_move_body();
bool __get_is_solid_visibly();
```

Use these functions to define whether or not a block is physically solid, move body solid, and visibly solid. These three attributes are described in the previous section. Each of these functions only needs to be defined if it returns a value different from get_is_solid. Here are some examples:

Here is a block invisible_wall.lua which is physically solid, move body solid, but not visibly solid. In other words, this is an invisible wall that the player cannot move through or shoot through but the player can see through it. Furthermore, it appears completely invisible:

```
function p.__get_is_solid() return true end
function p.__get_is_solid_visibly() return false end
function p.__get_tex() return "" end
function p.__main()
    set_default_block("block_invisible_wall")
end
```

Note the following:

The __get_tex function should return a non-empty string iff the block is visibly solid.

Here is "block_secret_wall.lua" which is physically solid and visibly solid, but not move body solid. This is a block that the player can move through, but nobody can see through it or shoot through it:

```
function p.__get_is_solid() return true end
function p.__get_is_solid_move_body() return false end
function p.__get_tex() return "concrete" end
function p.__main()
    set_default_block("block_secret_wall")
end
```

Here is a slightly different variant of this. Here is a "block_secret_wall2.lua" which is visibly solid but is neither physically or move body solid. So the player can move and shoot through this block, but they cannot see through it:

```
function p.__get_is_solid() return true end
function p.__get_is_solid_physically() return false end
function p.__get_is_solid_move_body() return false end
function p.__get_tex() return "concrete" end
function p.__main()
    set_default_block("block_secret_wall2")
end
```

Note that it does not hurt to define all of __get_is_solid_physically, __get_is_solid_move_body, and __get_is_solid_visibly.

5.5 p.__get_is_solid_visibly_glass

```
bool __get_is_solid_visibly_glass();
```

It is actually not as simple as each block either being visibly solid or not. There are four possibilities: visibly solid, visibly empty, visibly glass, or visibly water. By having only

```
function __p.get_is_solid_visibly() return true end
```

this makes the block visibly solid.

By having only

```
function __p.get_is_solid_visibly() return false end
```

this makes the block visibly empty.

However to make a block visibly glass, you need the following in the Block Lua Script:

```
function p.__get_is_solid_visibly() return false end
function p.__get_is_solid_visibly_glass() return true end
```

When a block is visibly glass, it should have a texture that is partially transparent. It is fastest to have each pixel in the texture be either 100% transparent or 100% opaque. If that is the case, then the block does not need to be sorted by the distance to the camera (which is slow).

If a block is visibly glass, this texture will be displayed in certain circumstances.

More precisely, if A and B are adjacent blocks and A is visibly empty and B is visibly glass, then there will be a partially opaque square displayed on the face of B that faces A. If A and B are adjacent blocks and they are BOTH visibly glass, then no texture will be displayed on the face of each block facing the other. If A and B are adjacent blocks and A is visibly glass but B is visibly solid, then there will be a square displayed on the face of B that faces A.

In other world, visibly glass blocks appear in the way that normal glass blocks appear in the Fractal Block World package Xar.

Here is the example `block_glass.lua`. It is physically and move body solid, but it is visibly glass. So the player cannot shoot or move through it, but the player can see through the block. Also, a partially transparent texture appears on the boundary of the glass blocks:

```
function p.__get_is_solid() return true end
function p.__get_is_solid_visibly() return false end
function p.__get_is_solid_visibly_glass() return true end
function p.__get_tex() return "orange_glass" end
function p.__main()
    set_default_block("block_glass")
end
```

Here `orange_glass` should be a texture which is partially transparent and partially opaque.

5.6 p.__get_is_solid_visibly_water

```
bool __get_is_solid_visibly_water();
```

Being visibly water is similar to being visibly glass. The difference is that when you are inside the water, you can see the outside surface of the water. Here is an example of a block that is visibly water:

```
function p.__get_is_solid() return true end
function p.__get_is_solid_visibly() return false end
function p.__get_is_solid_visibly_water() return true end
function p.__get_tex() return "tex_water" end

function p.__get_partially_transparent() return true end

function p.__main()
```

```
    set_default_block("block_water")
end
```

The function `__get_partially_transparent` must be defined and return true whenever the block has a pixel whose alpha is strictly between 0% and 100%. Note that the rendering of blocks where `__get_partially_transparent` returns true is slower than the blocks where it does not return true.

See the guide on the game's website about water for more information.

5.7 p.__get_tex_x_pos, p.__get_tex_x_neg, etc

```
string __get_tex(); //Mandatory.
string __get_tex_x_pos();
string __get_tex_x_neg();
string __get_tex_y_pos();
string __get_tex_y_neg();
string __get_tex_z_pos();
string __get_tex_z_neg();
```

The __get_tex is mandatory. It specifies the texture to be used on all 6 sides of the cube. However the texture used for each of the 6 sides can be overridden using the functions __get_tex_x_pos, __get_tex_x_neg, etc.

For example, here is a block script "block_mostly_blue.lua" which is blue on all 6 sides except for the top where it is red:

```
function p.__get_is_solid() return true end
function p.__get_tex() return "blue" end
function p.__get_tex_z_pos() return "red" end
function p.__main()
    set_default_block("block_mostly_blue")
end
```

5.8 p.__get_inv_tex_x_pos, p.__get_inv_tex_x_neg, etc

```
string __get_inv_tex_x_pos();
string __get_inv_tex_x_neg();
string __get_inv_tex_y_pos();
string __get_inv_tex_y_neg();
string __get_inv_tex_z_pos();
string __get_inv_tex_z_neg();
```

Note: "inv" stands for inverse.

Suppose A and B are adjacent blocks and A is either visibly empty or glass and B is visibly solid. So far, we have said in this situation we will display a

square texture on the B block that faces the A block. Which texture we use is determined by the B block. However if the functions __get_inv_tex_x_pos are defined in Block Lua Script for A, then these can override the texture used.

For example, here is a block which is not visibly solid but if we place it on the ceiling, it changes the ceiling texture. Call this Block Lua Script "block_ceiling_paint.lua":

```
function p.__get_is_solid() return false
function p.__get_is_solid_visibly() return false --Not needed.
function p.__get_tex() return "" end
function p.__get_inv_tex_z_neg() return "blue" end
function p.__main()
    set_default_block("block_e") --Empty.
    create_rect("block_ceiling_paint", 0,0,15, 15,15,15)
end
```

There is one more point: if A and B are adjacent blocks and if A is visibly empty (or glass) and B is visibly solid such that A defines an inverse texture, then on the face of B that faces A, should we use the texture specified by the B block or the A block? Currently there is a simple algorithm: the "inv" texture always takes precedence, assuming it is not the empty string.

5.9 p.__get_partially_transparent

See the section about `__get_is_solid_visibly_water`.

If this function is absent, it is equivalent to having the function return false. Note that you should ONLY have this function return true if there will be a pixel in the block that is not either 100% clear or 100% transparent. Blocks that are partially transparent are rendered quite a bit slower than blocks that are not.

Chapter 6

Block Lua Scripts Part 4: Game Functions

6.1 Even More Block Lua Module Functions

In Fractal Block World, we have blocks which have typical block functions but these blocks also turn into chunks themselves. In this chapter we describe more "module" functions of Block Lua Scripts, specifically focused around the typical block-like aspect of blocks. When these functions are called (by the engine), they have access to the Game API, not the Chunk Generation API or the Initialization API. We can call these the *auxiliary block functions*.

Here are functions that can be put into Block Lua Scripts that are described in this chapter:

```
//-----------------------------------------------
//            Called During Main Game
//-----------------------------------------------

void __on_close(int level, BlockPos pos);
void __on_adj_block_changed(
    int level, BlockPos pos, int side,
    string old_bt, string new_bt);
void __change_to(
    int level, BlockPos pos,
    string cur_bt, string new_bt);
bool __get_can_use(int level, BlockPos pos);
string __get_use_msg(int level, BlockPos pos);
void __on_use(int level, BlockPos pos);
void __on_use2(int level, BlockPos pos);
void __on_chunk_update(int level, BlockPos pos);
void __on_block_update(int level, BlockPos pos);
void __on_render(int level, BlockPos pos);
```

6.2 __on_close

```
void __on_close(int level, BlockPos bp);
```

This function of the Block Lua Script is called when the bounding box of the player is at most 1.0 units from the block.

Note: a BlockPos is a class with 3 float members: x,y,z.

Let's say that the player's body is in ground mode, which means the player's body is a cylinder. This is what the close function might look like:

```
__function p.on_close(level, bp)
    --Getting the (cylinder) body dimensions.
    local radius = ga_get_sys_f("game.player.move.ground.radius")
    local bot_to_eye = ga_get_sys_f("game.player.move.ground.bot_to_eye")
    local eye_to_top = ga_get_sys_f("game.player.move.ground.eye_to_top")

    --Do things with these numbers...
end
```

6.3 __on_adj_block_changed

```
void __on_adj_block_changed(
    int level, BlockPos bp, int side,
    string adj_old_bt,
    string adj_new_bt);
```

Let B be the block of which this on_adj_block_changed function is being called. The function on_adj_block_changed is called when a block adjacent to B has its block type changed. Note that a block being expanded into a chunk does not count as a block type change. However if A is a block adjacent to B and the block type of A changes from "dirt" to "air", then the "on_adj_block_changed" function will be called on the block B.

"Side" refers to the side relative to the block B. Side is an integer between 0 and 5 inclusive. 0 = x positive, 1 = x negative, 2 = y positive, etc.

"adj_old_bt" is the old block type string of the block adjacent to B. "adj_new_bt" is the new block type string of the block adjacent to B.

6.4 __change_to

```
void __change_to(
    int level, BlockPos pos,
    string cur_bt, string new_bt);
```

This is called when the type of the block changes. When it is called, the block currently has type cur_bt, but after this call the system will change the block type to new_bt.

6.5 __get_can_use

```
bool __get_can_use(int level, BlockPos pos);
```

This returns whether or not the user can "use" the block.

6.6 __get_use_msg

```
string __get_use_msg(int level, BlockPos pos);
```

This gets the string that is displayed when the player looks at the block. The function `__get_can_use` must return true in order for this message to be displayed (and `__get_use_msg` must return a non-empty string.

6.7 __on_use

```
void __on_use(int level, BlockPos pos);
```

This is called when the user "uses" the block. Note that if you open the console and type the command "use", it uses the current entity you are looking at.

6.8 __on_use2

```
void __on_use2(int level, BlockPos pos);
```

This is called when the player uses the "secondary" use ability of the block. Note that if you open the console and type the command "use2", it uses the current entity you are looking at.

6.9 __on_chunk_update

```
void __on_chunk_update(int level, BlockPos pos);
```

When this function exists, it is called periodically (many times per second). More precisely, whenever a *chunk* with the given block type exists in the active chunk tree, this function is passed the level and block position of the block that is that chunk.

6.10 __on_block_update

```
void __on_block_update(int level, BlockPos pos);
```

This is called when the current block is within a chunk that is being updated.

There is a catch: if the block script has BOTH an `__on_chunk_update` and a `__on_block_update` function, then in a situation when both can be called, the engine will only call `__on_chunk_update`.

6.11 __on_render

```
void __on_render(int level, BlockPos pos);
```

Blocks can have their own custom render functions. When this happens, they should be "visibly empty". The center of the coordinate system for rendering is the center of the block. This is very similar to the `__on_render` function for basic entities.

Chapter 7

Block Lua Scripts Part 5: Chunk Generation Lua State

Recall that in Fractal Block World there are three Lua states in which Lua code can be run: Initialization, Game, and Chunk Generation. Within each Lua state only certain API functions can be called. Also, various Lua scripts are only loaded into certain Lua states. The purpose of this chapter is to talk more about the Chunk Generation Lua state.

7.1 Entry Points

The engine will call certain functions defined in Lua scripts while the game is in the Chunk Generation Lua State. It just so happens that all these "entry points" are in block scripts (not in any other kind of script). Here is the full list of all entry points for the Chunk Generation Lua state:

```
//-----------------------------------------------
//          Called During Initialization
//-----------------------------------------------

void __chunk_gen_init();

//-----------------------------------------------
//          Called to Generate a Chunk
//-----------------------------------------------

void __main(); (mandatory)

//-----------------------------------------------
```

```
//        Called to Extract Custom Data
//-----------------------------------------------

string __main_dummy(string arg);
```

7.2 __chunk_gen_init

This function is called when the package is being loaded. You can use this to initialize local variables of your block script, which can then be used in the `__main` function of that block script. Here is an example of such a block script:

```
--File: block_forest.lua

local block_grass_int --An integer.

--Called during initialization.
function p.__chunk_gen_init()
    block_grass_int = bt_str_to_code("block_grass")
end

--Called to actually generate a chunk.
function p.__main()
    set_default_block("block_air")

    --At this point, the local variable block_grass_int
    --has already been set.
    --We can use this integer in place of a string block type
    --for functions that end with "_c".
    set_pos_c(7,7,0, block_grass_int)
end
```

Note that even though `__chunk_gen_init` is called in the Chunk Generation Lua state, it does not have access to all Chunk Generation API functions because there is currently no chunk being generated when it is called.

Also note that each Chunk Generation worker thread has their own Chunk Generation Lua state. So, we highly recommend NOT writing to local variables of the block script from within the `__main` function itself.

7.3 __main

This is the function that is called when it is time to actually generate a chunk. While this function is being called, you have access to all of the Chunk Generation API functions.

7.4 __main_dummy

This function is used for Lua code in the Game Lua state to get information from the Chunk Generation Lua state. It takes a string as an argument and it must return a string.

For example, suppose we have a block type script called `block_forest.lua` and we want the main game to get the value of a local variable stored in the block script. Suppose this is the code for the block script:

```
--File: block_forest.lua

local biome = "woodland"

--Called to actually generate a chunk.
function p.__main()
    set_default_block("block_air")
    set_pos_c(7,7,0, block_grass)
end

function p.__main_dummy(arg)
    if( arg == "get_biome" ) then
        return biome
    end
    if( arg = "get_favorite_color" ) then
        return "blue"
    end
    return ""
end
```

Then, from within a game script (run in the Game Lua state) we could have the following code:

```
local biome_of_block_forest =
    ga_worldgen_main_dummy("block_forest", "get_biome")
ga_print("The biome is: " .. biome_of_block_forest)
```

Note that the `__main_dummy` function is called at a time when the block script is NOT generating a chunk (that is the "dummy" part). So, the code in the `__main_dummy` function does not have access to the full Chunk Generation API.

Another way to get information from Chunk Generation code is to have the `__main` function (as opposed to the `__main_dummy` function) set "chunk variables" of the chunk that it is generating. Then from the Game Lua state we can read these variables.

7.5 Passing information to chunk generation code

Global variables that start with `worldgen.state` are automatically passed to chunk generation code. Let us explain. Suppose there is a global variable called `worldgen.state.num_trees`, which is an integer and it is set to 10. Because this is a global variable, it must be declared in the file `globals.txt`. Let this be a block script:

```
--File: block_forest.lua

--Called to actually generate a chunk.
function p.__main()
    set_default_block("block_air")
    local num_trees = state_get_i("num_trees")
    for i = 1,num_trees do
        local x = randi(0,15)
        local y = randi(0,15)
        set_pos(x,y,0, "block_tree")
    end
end
```

When a block of type `block_forest` is generated, it will place 10 trees at random.

Next, if we want to change this num trees variable from the Game Lua state, we can run the following code:

```
ga_set_i("worldgen.state.num_trees", 20)
ga_worldgen_refresh_state_var("worldgen.state.num_trees")
```

We need to call the `ga_worldgen_refresh_state_var` to actually flush the new value of the variable to the Chunk Generation code. The next time a Chunk Generation worker thread gets a task, it will be given the updated value of the variable.

Chapter 8

STD Lua Chunk Generation Helpers

In addition to the Chunk Generation Lua API, there are also built-in Lua helper functions, found in `base/WorldNodes/Helpers`. These functions can be used within any main function of a Block Lua Script.

If you are making a significantly large world, we suggest you do NOT use any of these standard chunk generation script helper functions. Instead, define your own. We recommend this because it would be best if your world is as self contained as possible. However, we encourage you to use these helper functions as inspiration. You can certainly copy them.

As we just said, because these functions may change in the future, instead of directly using one of these functions, you should copy the script file that contains it to your package's `WorldNodes/Helpers` directory (or even better you could copy one function at a time). For example, if you want to use `std.create_center()` which is defined in `std.lua`, then copy

`base/WorldNodes/Helpers/std.lua`

to

`myworld/WorldNodes/Helpers/std.lua`

Better yet, copy it to

`myworld/WorldNodes/Helpers/mystd.lua`

then you can call `mystd.create_center()`.

8.1 More Block Functions

8.1.1 std.create_center

```
void std.create_center(int diameter, string type);
```

The function creates a box of blocks in the center of the chunk. Calling

```
std.create_center(1,"stone")
```

is equivalent to calling

```
create_rect("stone", 7,7,7, 7,7,7).
```

Calling

```
std.create_center(2,"stone")
```

is equivalent to calling

```
create_rect("stone", 7,7,7, 8,8,8).
```

Calling

```
std.create_center(3,"stone")
```

is equivalent to calling

```
create_rect("stone", 6,6,6, 8,8,8),
```

etc.

8.1.2 std.create_tube

```
void std.create_tube(int diameter, string axis, string type);
```

This function creates a box of blocks centered along one of the three axes. Acceptable values for axis are "x", "y", and "z". For example,

```
std.create_tube(2, "z", "stone")
```

is equivalent to calling

```
create_rect("stone", 7,7,0, 8,8,15).
```

Here is a way you can create a hollow tube:

```
std.create_tube(4, "z", "stone")
std.create_tube(2, "z", "air")
```

8.1.3 std.create_half_tube

```
void std.create_half_tube(int diameter, string dir, string type);
```

This function is like create_tube except instead of the tube going from one side to the other, it goes from one side to the middle. Acceptable values for dir are "x_pos", "x_neg, "y_pos", "y_neg", "z_pos", "z_neg". For example, here is how to create a lollipop:

```
create_half_tube(1, "z_neg", "white_paper")
create_center(3, "red_cherry_candy")
```

8.1.4 std.create_edges

```
void std.create_edges(string type);
```

This function will create the 12 edges of the chunk using the specified block type.

8.1.5 std.create_shell

```
void std.create_shell(string type);
```

This function will create the outer shell of the chunk (without changing the 14x14x14 inside).

8.1.6 std.create_2x2_door

```
void std.create_2x2_door(
    string dir,
    string rim_type,
    strong hole_type);
```

This can be use in conjunction with `create_shell` to make a room with doors in it.

For example, the following code creates a door in the positive x direction:

```
std.create_2x2_door("x_pos", "block_s", "block_e")
```

It creates a hole (using the "empty" block "block_e") with a rim that is the solid block "block_s".

Chapter 9

In Game Tools

There are several tools available to debug your world while you are in the game.

9.1 The Path Command

If you open the console (press ~) and run the command `path last` it will print to the console the name of the script for whatever chunk you are in.

If you enter the command `path pos`, it will print that path but also show you the position of each chunk from its parent.

If you enter the command `path` this will print to the console the chunk names of the ancestors of the chunk you are in, starting from the root of the chunk tree all the way to the chunk you are in. That is, it will print the names of the chunks in your current chunk path.

If you enter the command `path dump` this will output to "Output/path.txt" your chunk path from the root of the chunk tree.

The format of the chunk path (on the line starting `chunk_path` in the file Output/path.txt) is a list of triples of hex characters (for x,y,z) separated by underscores (with the exception that the empty path is `EMPTY_PATH`).

9.2 The Script Command

If you open the console (press ~) and run the command `script` it will print to the console the Block Lua Script for whatever chunk you are in.

Remember you can use your mouse wheel or page up / page down to scroll up and down in the console.

9.3 The Gendoc Command

If you open the console (press ~) and run the command `gendoc`, the program will generate documentation files in the directory Output/Documentation.

One notable file is

```
Output/Documentation/xar/xar_builtin_block_types.txt
```

which is a list of all the built-in (xar) block type names.

Also, the gendoc command creates the folder

```
Output/Documentation/ProgrammingAPI
```

and generates several useful files. These files list the functions you can call from Lua, and the Lua functions which are called by the engine. The latter functions all start with a double underscore. You might want to have some of these files open while you read this manual.

Chapter 10

Coordinates

The layout of the world in Fractal Block World is unusual, so in this chapter we will describe the different coordinate systems that the engine uses.

10.1 The Chunk Tree (and the Active Chunk Tree)

The world consists of **chunks**, each of which is a 16x16x16 region of blocks. However every block can be subdivided into a chunk itself. So if C_1 is a chunk and B_1 is one of the blocks of C_1 (either solid or empty), then B_1 can be subdivided into its own chunk C_2. We say that the chunk C_2 is a **child** of the chunk C_1 (and C_1 is the **parent** of C_2). If B_2 is a block of C_2 and it is subdivided into a chunk C_3, then here C_3 is a child of C_2. We say that C_3 is a **descendant** of C_1 (and C_1 is an **ancestor** of C_3). We say that every chunk is a descendant of itself and an ancestor of itself. In this way, we have a *tree of chunks* (the **chunk tree**).

The **level** of a chunk is which level the chunk occurs in the chunk tree. So the root chunk of the world is in level 0 (and the root chunk is the ONLY chunk in level 0). Then there are 16x16x16 chunks in level 1. These are the children of the root chunk. Then there are 256x256x256 chunks in level 2, etc.

At any point in time the game can only interact with a few thousand chunks. These chunks form what we call the **active chunk tree**. When a chunk is added to the active chunk tree, we first procedurally generate the chunk from scratch and we then load any modifications to the chunk that have been saved previously. Later, we may remove a chunk from the active chunk tree.

10.2 Viewer Centric Position

The viewer (the eye of the player) is always in a chunk in the active chunk tree. In a sense, a chunk which contains the viewer position is the center of the world.

The viewer is always "on a certain level" which we call the **viewer level**. The **chunk of the viewer** is the chunk which contains the viewer which is on the viewer's level. A chunk is a **center chunk** iff it is the (unique) chunk of a level which contains the viewer position. In other words, a chunk is a center chunk iff it is an ancestor of the chunk of the viewer.

Within a level, the center chunk is said to have "viewer centric position" $(0, 0, 0)$. The chunk of that same level that is 1 chunk to the right (right is the positive x direction) is $(1, 0, 0)$, etc. We call the viewer centric position the **vcp** of the chunk for short.

So the viewer's chunk has vcp $(0, 0, 0)$. The parent of the viewer's chunk also has vcp $(0, 0, 0)$, etc.

When the viewer moves from one chunk to an adjacent one, this will change the vcp's of the chunks on his level L. However the vcp's of chunks on level $L - 1$ may or may not change, etc. The movement of the player is like walking on a treadmill: when the player moves from chunk to chunk, he keeps looping back and the world is the thing that actually moves.

10.3 Ways to describe the position of a chunk

There are three main ways to describe the position of a *chunk*:

- The path of the chunk.
- The level of the chunk together with the chunk's vcp (viewer centric position).
- The chunk id of the chunk.

10.4 Ways to describe the position of a block

There are four main ways to describe the position of a *block*. If the block is itself subdivided into a chunk, we can refer to it using the three methods above. However we can also refer to the block's position using the level of the block and the (x, y, z) "block position" of the block. Note that by the *level* of a block, we always mean the level of the chunk which contains the block.

10.4.1 chunk path

The path of the chunk is the path of the chunk from the root of the chunk tree. This is described as a string of triples of hex characters, separated by underscores. For example

"`7a3_221`"

is the path of a chunk C_2 on level 2 which we can reach as follows: start at the root C_0 and go to block $(7, 10, 3)$ of that chunk. That block is the same as lets

say chunk C_1. Then in C_1 we go to the block $(2, 2, 1)$. That block is the chunk for C_2.

The root has chunk path "`EMPTY_PATH`".

The main advantages of using the chunk path to refer to a chunk are 1) these paths do not change when the player moves or restarts the program and 2) the path of a chunk is valid even if the chunk is not in the active chunk tree.

The main disadvantage of using the chunk path to refer to a chunk is that this is slower than the other methods (when the chunk is very deep in the tree).

10.4.2 level + vcp

We can also refer to the position of a chunk using the level it is on (which is an integer) and its vcp (viewer centric position).

An advantage of this method is that the level + vcp combination only uses 4 integers. Another advantage is the level + vcp combination makes it easy to talk about the positions of vectors in a level. We discuss this later with "level positions".

The main disadvantage of this method is that vcp's will likely change whenever the viewer moves from one chunk to another.

10.4.3 chunk id

Every chunk in the active chunk tree has a **chunk id** (which is an integer). The system that maintains the active chunk tree has a counter N which starts at zero. Every time a chunk is added to the active chunk tree, is gets assigned the chunk id N and then N gets incremented.

Every time the user loads a game, this system is rebooted and so it goes back to zero. For this reason, chunk ids cannot be used for long term storage.

10.5 Level and local positions (for vectors)

Consider an entity, like a bullet or a rocket. This entity exists on some level L. We want to represent its position as a vector (x,y,z). There are two ways to do this: with a *local position* or with a *level position*.

10.5.1 Local positions

Every chunk has its own coordinate system. The origin of this coordinate system is in the left back bottom position of the chunk. So if (x,y,z) is a point in the chunk then each of x,y,z is between 0.0 and 16.0 inclusive.

Given a point in a chunk, we call the point's position relative to the chunk's coordinate system the **local position** of the point. For example, if a bullet is at the center of chunk C, then the bullet has the local position $(8.0, 8.0, 8.0)$.

10.5.2 Level positions (LP)

Consider a point on level L. The **level position** (LP) of the point is the position of the point relative to the center chunk of level L.

For example, if a point is in the center chunk of a level, then its local position is the same as its level position.

In a sense a point in space exists in more than one level. So for example we can convert a point's level position for level 13 into its level position for level 12, etc.

Note that when the player moves from one chunk to another, this will likely change an entity's level position.

10.6 Block Positions (BP) and Local Block Positions (LBP)

Block Positions are to Level Positions as Local Block Positions are to Local Positions.

10.6.1 Local Block Positions (LBP)

A chunk contains 16x16x16 blocks (either solid or empty). The positions of these blocks within the chunk are called the **local block positions** (LBPs) of the blocks. The left back bottom block in a chunk has the lbp (0,0,0). The right front top block in a chunk as the lbp (15,15,15). So an lbp for a block in a chunk is a triple (x,y,z) of integers such that each x,y,z is between 0 and 15 inclusive.

Actually each of x,y,z is a signed 8 bit integer (signed char). This allows representing the positions of blocks that are slightly outside the current chunk. This is sometimes useful. However you should not compute the hashcodes of local block positions outside the chunk.

A local block position can be represented by a single 4 byte integer, which we call a **local block position hashcode**. See the Lua script base/Game/std.lua which has the functions lbph_to_lbp and lbp_to_lbph to convert back and forth between an lbp and an lbph. This is how lbp_to_lbph is defined for example:

```
function p.lbp_to_lbph(lbp)
    return lbp.z + (16 * lbp.y) + (256 * lbp.x);
end
```

You can create an lbp using the function std.bp. This is code to convert an lbp into an lbp hash and then back again:

```
local lbp = std.bp(3,4,5)
local lbph = std.lbp_to_lbph(lbp)
local lbp2 = std.lbph_to_lbp(lbph)
--Now lbp should equal lbp2.
```

10.6.2 Block Positions (BP)

Every block is in some level. If a chunk C is in level L, then the blocks of C we also say are in level L (but when we subdivide each such block, the chunk the block becomes is in level $L + 1$).

The **block position** (BP) of a block is the position of the block relative to the center chunk of whatever level the block is in. A block position is a triple (x,y,z) of integers.

If a block is in the center chunk of a level, then the block's block position is the same as its local local block position.

Consider the block B_1 with block position (15,3,4) of the center chunk of a level. The block B_2 one to the right of this has block position (16,3,4). This is not located in the center chunk C_1, but instead it is the the chunk C_2 one to the right of the center chunk. The block B_2 has local block position (0,3,4) inside the chunk C_2.

Chapter 11

Environment Rect Lua Scripts

An Environment Rect can be added in the `__main` function of a Block Script by calling the function `add_env_rect`. Recall that this function has the following syntax:

```
void add_env_rect(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z, string type);
```

An Environment Rect is an invisible box. To make a 3x3x3 "death" type env rect that starts at `(1,2,3)` and goes to `(3,4,5)` inclusive, you would put

```
add_env_rect(1,2,3, 4,5,6, "death")
```

in the chunk's Block Lua Script `__main` function. An env rect is entirely contained in a single chunk. So when a chunk generation script calls `add_env_rect` to create an env rect, then `min_x`, `min_y, min_z`, `max_x`, `max_y`, `max_z`, must all be from 0 to 15 inclusive.

Environment Rect Lua Scripts are put in the folder EnvRects (in your package's top folder).

11.1 Environment Rect Lua Script Module Functions

Here are all the module functions of Environment Rect Lua Scripts. There is only one:

```
//-------------------------------------------
//            Called During Main Game
//-------------------------------------------
void __on_touch();
```

11.1.1 p.__on_touch

The `__on_touch` function of an env rect lua script is called when then player's bounding box intersects the env rect box. The base package has the "death" env rect script. That is, there is the script file `base/EnvRects/death.lua` and it reads as follows:

```
function p.__on_touch()
    damage_player(true, 10000)
end
```

Let's say that `damage_player` is a function we have defined elsewhere which deals damage to the player.

The game is updated many times per second, so if the player is touching an env rect, then the `__on_touch` function of the env rect will be called many times. So if you want an env rect with is a jump pad, then you may want to use a global variable (using `ga_get_f` and `ga_set_f`) to record the last time a jump pad was used. This way you can impose a rule that a jump pad cannot be used twice in a 0.1 second time interval for example.

11.2 Disclaimer

Environment Rect Lua Scripts are only used for basic purposes so far. At some point we might make changes to how these scripts work. For example, the `__on_touch` function might take an integer id as an argument and this could be used to query information about the env rect via the Game Lua-To-C API. Perhaps in this way the `__on_touch` function can get access to the parameters `min_x`, `min_y`, `min_z`, `max_x`, `max_y`, `max_z`.

Chapter 12

Basic Entity Lua Scripts

A basic entity (BEnt) occupies a block position and does not move. It is only rendered if you are on the same level as the entity. A basic entity is very lightweight. Gold in the xar package is an example of a basic entity.

Eventually we plan to make it so that everything you can do with a BEnt you can do with a more advanced type of entity: a moving entity (MEnt).

Basic Entity Lua Scripts are put in the folder BasicEnts (in your package's top folder).

12.1 Initialization BEnt Script Functions

Here are all the functions that Basic Entity Lua Scripts can define that are called by the engine in the *initialization Lua state*. Indeed, they are all called while the package is being initialized. They must start with a double underscore.

```
//-----------------------------------------------
//        Called During (Type) Initialization
//-----------------------------------------------
string __get_mesh();
string __get_mesh2();
bool   __get_pulsates();
float  __get_scale();
float  __get_touch_dist();
```

12.2 Game BEnt Script Functions

Here are all the functions that Basic Entity Lua Scripts can define that are called by the engine in the *game Lua state*. They must start with a double underscore.

```
//-----------------------------------------------
```

```
//            Called During Main Game
//-----------------------------------------------
void   __on_touch(int level, BlockPos bp);
bool   __get_can_use(int level, BlockPos bp);
string __get_use_msg(int level, BlockPos bp);
void   __on_use(int level, BlockPos bp);
void   __on_use2(int level, BlockPos bp);
void   __on_render(int level, BlockPos bp);
```

12.3 Initialization Functions

These functions are called when the package is loaded. They are called one time for each Basic Entity Lua Script (not one time for each Basic Entity itself in the world). For example, if there is the script `BasicEnts/cheese.lua`, then the function `p.get_mesh` of `cheese.lua` will be called only once when the package is loaded to determine the mesh of basic entities of type cheese.

12.3.1 p.__get_mesh

```
string __get_mesh();
```

This function is called by the game to determine the mesh of the basic entity. If the following is in the basic entity's Lua script `grenade_box.lua`, then the mesh `small_box` will be used for the mesh of the basic entity:

```
function p.__get_mesh() return "small_box" end
```

Here `small_box` must be a mesh name that is listed in the file

`Meshes/mesh_names.txt`.

If the function `p.get_mesh` is not defined in the basic entity Lua script, then the mesh name that is used is the name of the basic entity lua script itself. In our example, the mesh name `grenade_box` would be used if `p.__get_mesh` was not defined.

12.3.2 p.__get_mesh2

```
string __get_mesh2();
```

A basic entity actually uses two meshes for rendering, the second being optional. Both meshes are rendered centered in the block that the basic entity is in. Use `p.__get_mesh2` to specify the second mesh name. If `p.__get_mesh2` is not defined in the basic entity Lua script, then only the first mesh will be used (specified by `p.__get_mesh`).

12.3.3 p.__get_pulsates

```
bool __get_pulsates();
```

This specifies whether or not the basic entity "pulsates". If it pulsates, then its size changes sinusoidally over time. This is only used for rendering purposes. If this function is not defined, then the basic entity will pulsate by default.

12.3.4 p.__get_scale

```
float __get_scale();
```

This allows you to change the size of a basic entity (for rendering purposes only) without having to change the entity's mesh. If `p.__get_scale` is not defined, then the scale number will be 1.0. Suppose the basic entity Lua script `grenade_box.lua` includes the following:

```
function p.__get_mesh() return "small_box" end
function p.__get_scale() return 2.0 end
```

Then a `small_box` mesh will be used for rendering the basic entity, but it will be scaled by a factor of 2.

12.3.5 p.__get_touch_dist

```
float __get_touch_dist();
```

Let R be the touch distance of a basic entity. When the player's eye is within distance R from the center of the basic entity, then the basic entity's `on_touch` function will be called. The `p.__get_touch_dist` function specifies this distance. For example, suppose the basic entity Lua script `grenade_box.lua` includes the following:

```
function p.__get_touch_dist() return 3.0 end
```

Then when the player is within 3.0 units of the grenade box, then the on_touch function of the grenade box will called. Note that the width of a block is 1.

12.4 Game Functions

These functions are called during normal game execution. The engine calls these functions during various times, and it passes to these functions the chunk_id of the chunk containing the entity along with the local block position of the entity in that chunk. The local block position is passed as a *local block position hash code*, which is an integer which codes the lbp. See Section 10.6.1 for how to use the functions `std.lbph_to_lbp` and `std.lbp_to_lbph` to convert back and forth between local block positions and local block position hash codes.

12.4.1 p.__on_touch

```
void __on_touch(int level, BlockPos bp);
```

Let R be the touch distance of the basic entity (see the function `get_touch_dist`). When the player's eye is within R units of the center of the basic entity, the basic entity's `__on_touch` function will be called.

So suppose the basic entity Lua script `grenade_box.lua` includes the following:

```
function p.__on_touch(level, bp)
    local num_grenades_old = ga_get_i("num_grenades")
    local num_grenades_new = num_grenades_old + 10
    ga_set_i("num_grenades", num_grenades_new)
end
```

When the player is sufficiently close to the grenade box, then the `on_touch` function of the grenade box will be called and this will give the player 10 grenades.

12.4.2 p.__get_can_use

```
bool __get_can_use(int level, BlockPos bp);
```

The player is able to "use" certain entities. When the player is relatively close to a basic entity, is looking at the entity, and the player presses their "use key", then the game asks the entity if it can be used (via this `__get_can_use` function).

Here is part of a basic entity Lua script which makes it so the entity can only be used if the player is at most 2.0 units from the basic entity:

```
function p.__get_can_use(level, bp)
    local dist = ga_block_dist_to_viewer(level, bp)
    return ( dist < 2.0 )
end
```

12.4.3 p.__get_use_msg

```
string __get_use_msg(int level, BlockPos bp);
```

When the player looks at a basic entity (and is close enough), a text message is displayed at the center of the screen. The function `__get_use_msg` determines this message. If this function returns the empty string, then no text will be displayed. So to be clear, for the use message to be displayed, the function `__get_can_use` must return true and `__get_use_msg` must return a non-empty string.

Here is code for the `grenade_box.lua` lua script that displays the text "10 grenades". The text will be in green if the player can use the box to get more grenades, and it will be in red if the player already has the max number of grenades in their inventory.

```
function p.__get_use_msg(level, bp)
    local max_grenades = 100
    local player_grenades = ga_get_i("num_grenades")
    local color_str = ""
    if( player_grenades < max_grenades ) then
        color_str = "^x00ff00" --Green.
    else
        color_str = "^xff0000" --Red.
    end
    return color_str + " 10 grenades"
end
```

12.4.4 p.__on_use

```
void __on_use(int level, BlockPos bp);
```

If the p.__get_can_use function returns true and the player uses the basic entity, then the on_use function is called.

```
function p.__on_use(level, bp)
    local num_grenades_old = ga_get_i("num_grenades")
    local num_grenades_new = num_grenades_old + 10
    ga_set_i("num_grenades", num_grenades_new)
end
```

12.4.5 p.__on_use2

```
void __on_use2(int level, BlockPos bp);
```

This is just like __on_use, except it is called when the player uses the *secondary* use function of the entity.

We recommend being careful about having too many entities with secondary use functions, because it is can be a bit much for the player to keep track of.

12.4.6 p.__on_render

```
void __on_render(int level, BlockPos bp);
```

If this function exists, then this function is used for rendering instead of the one that is hardcoded into the engine. See the functions in the Game API that start with `ga_render_`.

At least for now, for backwards compatability you can also have the function be called `__render` instead of `__on_render`.

12.5 An example

Here is the full code for a grenade_box.lua basic entity Lua script. The player can pick up the grenade box by either 1) touching it, 2) using it, or 3) using telekinesis.

```
function p.__get_mesh() return "small_box" end
function p.__get_mesh2() return "" end      --Not needed.
function p.__get_pulsates() return true end --Not needed.
function p.__get_scale() return 1.0 end     --Not needed.
function p.__get_touch_dist() return 1.5 end

--This function actually gives the player grenades.
function p.payload()
    local max_grenades = 100
    local num_grenades_old = ga_get_i("num_grenades")
    local num_grenades_new = num_grenades_old + 10
    if( num_grenades_new > max_grenades ) then
        num_grenades_new = max_grenades
    end
    ga_set_i("num_grenades", num_grenades_new)

    --Removing the entity for one hour.
    ga_bent_remove_temp(level, bp, 60*60)
end

function p.__get_can_use(level, bp)
    local max_grenades = 100
    local player_grenades = ga_get_i("num_grenades")
    if( player_grenades >= max_grenades ) then return false end

    local dist = ga_lbp_dist_to_viewer(chunk_id, lbp_hash)
    if ( dist > 5.0 ) then return false end

    return true
end

function p.__get_use_msg(level, bp)
    local can_use = p.get_can_use(chunk_id, lbp_hash)
    if can_use then
        color_str = "^x00ff00" --Green.
    else
        color_str = "^xff0000" --Red.
    end
    return color_str .. " 10 grenades"
end
```

```
function p.__on_use(int chunk_id, int lbp_hash)
    p.payload(chunk_id, lbp_hash)
end

function p.__on_touch(int chunk_id, lbp_hash)
    p.payload(chunk_id, lbp_hash)
end
```

Chapter 13

Moving Entity Lua Scripts

A moving entity (MEnt) exists within a chunk, but it can move from one chunk to another. Every moving entity is said to be "in" a unique chunk.

Moving Entity Lua Scripts are put in the folder MovingEnts (in your package's top folder).

13.1 Roaming vs Non-Roaming Moving Entities

Moving entities are put into major categories: roaming and non-roaming. Roaming moving entities are ments that are created during game play by the usual game system. Non-roaming moving entities, on the other hand, are the same thing as moving entities that were originally created from procedural world generation.

A roaming ment only exists for a certain amount of time, and then it vanishes completely (leaving nothing behind). A non-roaming ment (a ment from procedural world generation) can be modified and these modifications are stored (for a certain amount of time).

Consider a troll monster ment that comes from procedural world generation (so it is non-roaming). If the player kills the troll, then it will remain removed from the world for a certain amount of time. However the troll will respawn after a certain amount of time. So if the player kills the troll, walks away for a minute, and then comes back, the troll will still be gone. However if the player kills the troll, walks away for many hours, and then comes back, then the troll will have respawned.

13.2 Type IDs, Instance IDs, and Code IDs

Every moving entity type has an id (its "type_id"). Every instance of a moving entity has an instance id (its "inst_id"). However these only refer to moving entities that exist in the active chunk tree. When we save the game, we do not save the instance ids of moving entities. Instead we save their "code ids".

When a moving entity is procedurally generated (when a chunk is procedurally generated), it is assigned as pseudo random code id. If the chunk is created a second time it will be assigned the same code id. That is, this explains how *non-roaming* moving entities get their code ids. On the other hand, when a *roaming* moving entity is created, it is assigned a truly random code id. Roaming moving entities have positive code ids whereas non-roaming moving entities have negative code ids.

13.3 Initialization MEnt Script Functions

Here are all the functions that Moving Entity Lua Scripts can define that are called by the engine in the *initialization Lua state*. Indeed, they are all called while the package is being initialized. They must start with a double underscore.

```
//---------------------------------------------
//        Called During (Type) Initialization
//---------------------------------------------
void __type_init(int type_id);
```

13.4 Game MEnt Script Functions

Here are all the functions that Moving Entity Lua Scripts can define that are called by the engine in the *Game Lua state*. They must start with a double underscore.

```
//---------------------------------------------
//            Called During Main Game
//---------------------------------------------
void __on_add_to_live_world(
    int inst_id);
void __on_update(
    int inst_id, float elapsed_time, float elapsed_level_time);
void __on_alarm(
    int inst_id, string alarm_name);
void __on_too_fine(
    int inst_id, int fine_chunk_id, Vector fine_offset);
bool __on_block_hit(
    int inst_id, int level,
    BlockPos bp, Vector lp,
    int normal_side, Vector normal);
bool __on_block_hit_nonfertile(
    int inst_id, int level,
    BlockPos bp, Vector lp,
    int normal_side, Vector normal);
bool __on_ment_hit(
```

```
    int hitter_inst_id, int hittie_inst_id,
    int level, Vector lp,
    Vector normal);
void __on_level_travel(
    int inst_id, int level,
    Vector lp_start, Vector lp_end);
void __on_closest(
    int inst_id,
    float dist_to_viewer,
    Vector dir_to_viewer);
void __on_telefragged(int inst_id);
bool __get_can_use(int inst_id);
string __get_use_msg(int inst_id);
void __on_use(int inst_id);
void __on_use2(int inst_id);
void __on_render(int inst_id, float radius);
```

13.4.1 __type_init

```
void __type_init(int type_id);
```

The `__type_init` function of each moving entity is called exactly once while the package is being loaded (not during main game play). Only the Initialization Lua API is available when this `__type_init` is called (there is a chapter devoted to that API in this manual). The `__type_init` function is passed an integer number which identifies the moving entity type. Here is what the `__type_init` function might look like for a troll monster moving entity:

```
function p.__type_init(tid)
    ia_ment_new_static_var_i(tid, "max_health", 200)
    ia_ment_new_var_i(tid, "health", 200, 60.0 * 60.0)
    ia_ment_set_builtin_var_f(tid, "__radius", 2.5);
end
```

Here `ia_ment_new_static_var` is a function that is part of the Initialization Lua API. The call to that function creates a new variable called "max_health" that is associated to the moving entity type.

The call to the function `ia_ment_new_var_i` creates a new variable "health" associated to every instance of the moving entity.

Moving entities also have built-in variables that always exist. The call to the function `ia_ment_set_builtin_var_f` changes the built-in variable `__radius` to "2.5". Later in this chapter we list all the built-in variables and explain what they do.

13.4.2 __on_add_to_live_world

```
void __on_add_to_live_world(int inst_id);
```

When a moving entity is added to the active chunk tree, this function is called. There is one other time this function is called. Every chunk in the active chunk tree is either active or passive. A passive chunk is basically asleep: it does not get updated. When a passive chunk is changed to become active, the `__on_add_to_live_world` of each moving entity in the chunk is called.

Something you might want to put into the `__on_add_to_live_world` function are calls to set "alarms".

13.4.3 __on_update

```
void __on_update(
    int inst_id, float elapsed_time, float elapsed_level_time);
```

The game updates the world about 25 times per second. We call these "discrete updates". If a chunk (in the active chunk tree) is active, then during each discrete update the chunk gets updated. When a chunk is updated, the `__on_update` function of every moving entity in the chunk is called.

The `elapsed_time` is how much time has passed since the last discrete update. Every level (level of the chunk tree) has its own time system. Time on coarser levels passes slower. The `elapsed_level_time` is how much time has elapsed on the level that the moving entity is in.

13.4.4 __on_alarm

```
void __on_alarm(
    int inst_id, string alarm_name);
```

The engine maintains a collection of "alarms". A moving entity alarm is a triple (`inst_id`, `alarm_name`, `time`) where inst_id is the instance id of a moving entity, `alarm_name` is a string, and time is a time (either in game time or in a level's time). The time is when the alarm should "go off". When a moving entity type alarm goes off, the engine calls back the function `p.__on_alarm` of the moving entity with instance id `inst_id`.

Here is example code for the `__on_alarm` function. It deals 10 damage to the moving entity and then sets another alarm.

```
function p.__on_alarm(inst_id, alarm_name)
    if( alarm_name == "poison" ) then
        local health = ga_ment_get_i(inst_id, "health")
        health = health - 10
        ga_ment_set_i(inst_id, "health", health)
        local cur_time = ga_get_game_time()
        local next_time = cur_time + 1.0 --One second in the future.
        ga_ment_set_alarm(inst_id, next_time, "poison")
    end
end
```

The `ga_ment_set_alarm` function is described in the chapter about the Game Lua-to-C API. Note that there is also the function `ga_ment_set_alarm_level`, which sets an alarm that goes off at a given *level* time (as opposed to a *game* time).

13.4.5 __on_too_fine

```
void __on_too_fine(
    int inst_id, int fine_chunk_id, Vector fine_offset);
```

Every moving entity has a max and a min level that it can exist on. The max level L is the finest level on which the entity can exist. If we attempt to move the entity to an even finer level (level $L+1$, then the on_too_fine function is called. This function is passed the offset of the moving entity in the fine chunk on level $L+1$ (as well as the chunk id of that chunk).

13.4.6 __on_block_hit

```
bool __on_block_hit(
    int inst_id, int level,
    BlockPos bp, Vector lp,
    int normal_side, Vector normal);
```

This is called when the moving entity hits a block.

The function should return true iff the hit is "terminal", meaning the moving entity should not move any farther. Also, if the block hit is terminal, the moving entity will be removed afterwards. However right now the engine ignores the return value of `__on_block_hit` and pretends that the function returns true. So all block hits are terminal. In the future we may make the engine more general, where there can be non-terminal block hits.

The arguments level and bp describe the position of the block that is being hit. Recall that bp has the three integer members x,y,z. The vector lp describes the position of the hit. It is the "level position" (the position of the hit in the given level). The Vector normal is a length one vector that is normal to the surface of intersection. That is, the normal vectors points out from the intersection point away from the surface. The `normal_side` integer describes the side of the block that was hit. Recall that 0 = x_pos, 1 = x_neg, 2 = y_pos, 3 = y_neg, 4 = z_pos, 5 = z_neg.

Here is the code for a moving entity which is a projectile that when it hits a block, it creates a stone block adjacent to the block of impact. The stone block will exist for 60 seconds.

```
function p.__on_block_hit(
    inst_id,
    level, bp, lp,
    normal_side, normal)
--
```

```
    --Getting the adjacent block position.
    local adj_bp = std.get_adj_bp(bp, normal_side)

    --Adding a stone block that will
    --exist for 60 seconds.
    ga_block_change_rl(level, adj_bp, "stone", 60.0)

    return true --Terminal hit.
end
```

13.4.7 __on_block_hit_nonfertile

```
bool __on_block_hit_nonfertile(
    int inst_id, int level,
    BlockPos bp, Vector lp,
    int normal_side, Vector normal);
```

This is just like `__on_block_hit`, except it is called when the ment hits a block that is outside the *fertile* radius. That is, the ment hits a block in a chunk in which only blocks have been loaded.

13.4.8 __on_ment_hit

```
bool __on_ment_hit(
    int hitter_inst_id, int hittie_inst_id,
    int level, Vector lp,
    Vector normal);
```

This is called when the moving entity (the hitter) hits another moving entity (the hittie).

The function should return true iff the hit is "terminal", meaning the moving entity should not move any farther. Also, if the block hit is terminal, the moving entity will be removed afterwards.

The arguments level and lp describe the position of the block that is being hit. The vector lp describes the position of the hit. It is the "level position" (the position of the hit in the given level).

The Vector normal is a length one vector that is normal to the surface of intersection. This could be used for blood spurting, say if the hitter is a bullet and the hittie is a monster.

The Lua function __on_ment_hit can optionally call

```
ga_return_b("remove", false)
```

before the function returns to make it so the ment is not removed by the engine (even if `__on_ment_hit` function returns true).

Here is code for a bullet moving entity. If the hittie has a health variable, it will deal 10 damage to the hittie.

```
bool function p.__on_ment_hit(
    hitter_inst_id, hittie_inst_id,
    level, lp, normal)
--
    local hittie_type = ga_ment_get_type(hittie_inst_id)
    if ga_ment_var_exists(hittie_type, "health") then
        local health = ga_ment_get_i(hittie_inst_id, "health")
        health = health - 10
        ga_ment_set_i(hittie_inst_id, "health", health)
    end

    --It is NOT a terminal hit:
    --the bullet can pass through this monster and
    --go on to hit other monsters.
    return false
```

13.4.9 __on_level_travel

```
void __on_level_travel(
    int inst_id, int level,
    Vector lp_start, Vector lp_end);
```

This function is called when a moving entity moves from one point (`lp_start`) to another (`lp_end`), all when the particle is on a certain level.

Here is code for a bullet which leaves a trail of smoke. The key is the function `ga_particle_trail`, which creates a trail of particles.

```
void __on_level_travel(
    int inst_id, int level,
    Vector lp_start, Vector lp_end)
--
    local args = {}
    args.level = level
    args.pos_start = lp_start
    args.pos_end = lp_end
    args.ttl_min = 0.5
    args.ttl_max = 0.5
    args.size_min = 0.1
    args.size_max = 0.1
    args.color = std.vec(1.0, 1.0, 1.0)
    args.fade_time_min = 0.5
    args.fade_time_max = 0.5
    args.speed_min = 0.0
    args.speed_max = 0.0
    args.tex = "particle_2"
    args.radius_min = 0.0
    args.radius_max = 0.0
```

```
    args.avg_len = 1.0
    args.use_min_dist = false
    ga_particle_trail(args)
end
```

13.4.10 __on_closest

```
void __on_closest(
    int inst_id,
    float dist_to_viewer,
    Vector dir_to_viewer);
```

If this function exists in the moving entity script, then the following will happen: every discrete update the engine will calculate the distance from the moving entity to the viewer (the player). As long as this distance goes down, nothing will happen. However once this distance increases, the moving entity's `__on_closest` function will be called.

This can be used, for example, to have a rocket explode when it is at its closest point to the player.

For convenience, this function is passed both the distance to the player and also a length one Vector which points from the moving entity to the viewer.

13.4.11 __on_telefragged

```
void __on_telefragged(int inst_id);
```

This is called when a moving entity is telefragged (right before it is removed).

13.4.12 __get_can_use

```
bool __get_can_use(int inst_id);
```

Moving entities, just like basic entities, can be "used". This function determines whether or not the moving entity can be used. If this function is missing, then the entity cannot be used.

```
function p.__get_can_use(inst_id)
    --Getting the global variable for player health.
    local player_health = ga_get_i("health")
    if( player_health < 100 ) then
        return true --Can use the entity.
    else
        return false --Cannot use the entity.
    end
end
```

13.4.13 __get_use_msg

```
string __get_use_msg(int inst_id);
```

When the player looks at a moving entity, the string that is returned from this function is displayed in the center of the screen. This happens even if the `__get_can_use` function returns false. When the `__get_use_msg` returns the empty string, no message is displayed when the player looks at the moving entity. If the `__get_use_msg` does not exist, then that is equivalent to the function existing and returning the empty string. So to be clear, for the use message to be displayed, the function `__get_can_use` must return true and `__get_use_msg` must return a non-empty string.

Continuing the example from the subsection about `__get_can_use`, here is code for a "healing shrine" which heals the player if their health is below 100:

```
function p.__get_use_str(inst_id)
    local can_use = p.__get_can_use(inst_id)
    if( can_use ) then
        return "Use this to get 100 health"
    else
        return "You already have full health"
    end
end
```

13.4.14 __on_use

```
void __on_use(int inst_id);
```

When the player attempts to "use" a moving entity, first the `__get_can_use` function of the entity is called. If that function returns true, then this `__on_use` function is called.

Continuing our example from the last two subsections, here is code for a "healing shrine":

```
function p.__on_use(inst_id)
    --get_can_use must have returned true.

    --Setting the player health to 100.
    ga_set_i("health", 100)
end
```

13.4.15 __on_use2

```
void __on_use2(int inst_id);
```

This is just like `__on_use`, but it is called for the secondary use function of the moving entity.

13.4.16 __on_render

```
void __on_render(int inst_id, float radius);
```

If this function exists, then this function is used for rendering instead of the one that is hardcoded into the engine. See the functions in the Game API that start with `ga_render_`.

At least for now, for backwards compatability you can also have the function be called `__render` instead of `__on_render`.

13.5 Moving Entity Vars Overview

Every moving entity type has a list of variables associated to it. Each variable has a type, being either "bool", "int", "float", "vector", and "string". This list of variables must be specified during the package initialization phase. That is, no new moving entity variables can be added during normal game play.

Each variable has a **default value** (associated to the moving entity type). A moving entity (instance) only stores a variable if that variable has a value different from its default value (we use a sparse system for storing variables).

13.5.1 Static variables

Some of these variable values are only associated to the "type" of moving entity itself. These are called **static** variables. We can think of a static variable as a normal variable but it only has a default value.

On the other hand, non-static variable values are associated to each moving entity instance. These values can be different from their default value.

The value of a static variable for a moving entity can only be set during the package's initialization phase. Similarly, the default value of non-static variables for a moving entity can also only be set during the package's initialization phase.

13.5.2 Revert lengths

Every non-static variable has a **revert length** (rl). Once a non-static variable is changed, then (assuming it is not changed again) after the revert length many seconds have passed, the variable will be reset to its default value.

Note that when a variable is reverted, this only finally takes place when the player leaves the chunk of the moving entity so that the chunk is removed from the active chunk tree.

13.5.3 Built-in variables

Some moving entity variables are automatically created by the engine. All these variables start with double underscores. Take the built-in variable `__mesh` for example. This is created by the engine, but the user can modify this during initialization by calling

```
ia_ment_set_builtin_var_i(tid, "__mesh", "sphere_100_poly");
```

This modifies the built-in variable.

It is currently impossible to modify the revert length of a built-in variable, but this may change in later versions of the game.

Both static and non-static built-in variables can be changed like this. However some built-in variables are **read-only**. A read-only variable should not be modified. The engine may modify read-only variables, but you may not.

If a built-in read-only variable is modified, this will result in undefined behavior (although in future versions of the game we may simply make the program exit if any read-only variables are illegally modified).

13.6 List of all moving entity built-in vars

Here is a list of all the built-in variables for moving entities. We also list the default value of each variable and also the revert length (`rl`).

The largest revert length we use is one hundred thousand hours. We do not recommend using anything larger than that. Some of the variables are static. **To avoid an overload of information, we will list the revert lengths here but not when we talk about the variables later.**

```
//-----------------------------------------------
//          Moving Entity Builtin Vars
//-----------------------------------------------
static bool __disable_saving = false

READ_ONLY bool __from_world_gen = false (rl = 100K hours)

READ_ONLY bool   __grounded
READ_ONLY Vector __grounded_offset
READ_ONLY Vector __grounded_offset_old

static float __ttl = -1.0
static float __ttl_grounded = 60*60 (1 hour)
float        __game_end_time = -1.0            (rl = 100K hours)
float        __respawn_length = 60*60 (1 hour) (rl = 100K hours)

READ_ONLY float __add_to_live_world_time = -1.0 (rl = 5 minutes)

static int __extra_min_levels = 0
static int __extra_max_levels = 0

READ_ONLY int __start_level = -1 (rl = 1 minute)
READ_ONLY int __min_level = -1   (rl = 1 minute)
READ_ONLY int __max_level = -1   (rl = 1 minute)
READ_ONLY int __level = -1       (rl = 1 minute)
```

```
READ_ONLY int __chunk_id = -1    (rl = 1 minute)

READ_ONLY Vector __offset     = Vector(7.5,7.5,7.5) (rl = 100K hours)
READ_ONLY Vector __offset_old = Vector(7.5,7.5,7.5) (rl = 100K hours)

bool __hide = false (rl = 100K hours)

Vector __vel          = Vector(0,0,0) (rl = 100K hours)
bool   __vel_disabled = false         (rl = 100K hours)

string __mesh         = ""  (rl = 1 hour)
float  __alpha        = 1.0 (rl = 1 hour)
string __tex_override = ""  (rl = 1 hour)
bool   __force_no_lighting = false (rl = 1 hour)

float __min_render_dist          = -1.0 (rl = 1 minute)
float __max_render_dist          = -1.0 (rl = 1 minute)
float __max_screen_size          = -1.0 (rl = 1 minute)
float __max_screen_size_time_len = -1.0 (rl = 1 minute)

int __team_id_source = 0 (rl = 1 minute)
int __team_id_target = 0 (rl = 1 minute)

bool __collides            = true  (rl = 1 minute)

bool __solid_wrt_player    = false (rl = 1 minute)

bool __point_block_correct = false (rl = 1 minute)
bool __ment_correct        = false (rl = 1 minute)

bool __player_can_telefrag = true (rl = 1 hour)

float __radius        = 1.0   (rl = 1 minute)
bool  __radius_lvlinv = false (rl = 1 minute)

bool          __homing                 = false (rl = 1 minute)
float         __homing_speed           = 1.0   (rl = 1 minute)
float         __homing_min_dist        = 0.0   (rl = 1 minute)
READ_ONLY int __homing_target          = -1    (rl = 1 minute)
bool          __homing_player_pathing  = false (rl = 1 minute)
bool          __homing_player_vis_test = false (rl = 1 minute)
bool          __homing_only_diff_level = false (rl = 1 minute)

float           __gas_cloud_period       = -1.0          (rl = 1 minute)
READ_ONLY float __gas_cloud_last_time    = -1.0          (rl = 1 minute)
float           __gas_cloud_ttl          = 2.0           (rl = 1 minute)
```

```
Vector          __gas_cloud_color        = Vector(1,1,0) (rl = 1 minute)
float           __gas_cloud_radius       = 2.0           (rl = 1 minute)
float           __gas_cloud_trigger_dist = 48.0          (rl = 1 minute)

float  __turn_speed               = 1.0           (rl = 1 minute)
bool   __turning_disabled         = false         (rl = 1 hour)
bool   __turn_towards_player      = false         (rl = 1 hour)
Vector __turn_towards_player_axis = Vector(1,0,0) (rl = 1 hour)
bool   __turn_around_vel          = false         (rl = 1 hour)
Vector __turn_around_vel_axis     = Vector(0,0,1) (rl = 1 hour)
bool   __mesh_fixed_frame         = false         (rl = 1 minute)
Vector __mesh_fixed_frame_v1      = Vector(1,0,0) (rl = 1 minute)
Vector __mesh_fixed_frame_v2      = Vector(0,1,0) (rl = 1 minute)
Vector __mesh_fixed_frame_v3      = Vector(0,0,1) (rl = 1 minute)

READ_ONLY bool   __towards_viewer_valid = false         (rl = 1 minute)
READ_ONLY Vector __towards_viewer_vec   = Vector(0,0,0) (rl = 1 minute)
READ_ONLY Vector __towards_viewer_dir   = Vector(0,0,1) (rl = 1 minute)
READ_ONLY float  __dist_to_viewer       = -1.0          (rl = 1 minute)
READ_ONLY float  __dist_to_viewer_old   = -1.0          (rl = 1 minute)

string        __death_anim                   = ""   (rl = 1 minute)
READ_ONLY int __death_anim_stage             = 0    (rl = 1 minute)
float         __death_anim_start             = -1.0 (rl = 1 minute)
float         __death_anim_end               = -1.0 (rl = 1 minute)
float         __death_anim_alpha_fade_alpha1 = 1.0  (rl = 1 minute)
float         __death_anim_alpha_fade_alpha2 = 1.0  (rl = 1 minute)
```

13.7 Explanation of all moving entity built-in vars

13.7.1 __disable_saving

```
static bool __disable_saving = false
```

For every variable, you can disable whether on not it is saved to file when it is changed. This is described in Section 16.2.5. The variable `__disable_saving`, when true, will make it so all variables are disabled from being saved. Indeed, when saving the game and exiting, there will be no trace left behind of a moving entity where `__disable_saving` is true.

13.7.2 __from_world_gen

```
READ_ONLY bool __from_world_gen = false
```

This variable is true iff the moving entity was created in procedural world generation code. This variable is saved to file in a unique way.

Moving entities where `__from_world_gen` is false are also called **roaming**. For a moving entity where `__from_world_gen` is true (a non-roaming entity), we break into two categories: **grounded** non-roaming entities and **non-grounded** (or moved) non-roaming entities. A non-roaming entity is called grounded iff it is still in the chunk where it was procedurally generated. Otherwise, it has moved from its original chunk.

13.7.3 __grounded

```
READ_ONLY bool __grounded
```

This is true iff the ment was created by procedural world generation and has not moved from its original chunk.

13.7.4 __grounded_offset

13.7.5 __grounded_offset_old

```
READ_ONLY Vector __grounded_offset
READ_ONLY Vector __grounded_offset_old
```

These are used internally by the engine.

13.7.6 __ttl, __ttl_grounded, __game_end_time

```
static float __ttl           = -1.0
static float __ttl_grounded  = 60*60 (1 hour)
float        __game_end_time = -1.0
```

The variable `__ttl` should be more precisely called "`__ttl_roaming`", but we call it `__ttl` because it is the most common type of ttl which is modified. The variable `__ttl` is the length of time (in seconds) a roaming entity exists after it is created.

Similarly, `__ttl_grounded` is the length of time (in seconds) a non-roaming moving entity exists (a moving entity created from procedural world generation) before it is despawned (which will reset all of its variables). To make life simple, it makes sense to leave `__ttl_grounded` as one hour: All changes to the moving entity will be reverted in one hour and it will be despawned.

If a moving entity is roaming, then let length = `__ttl`, and if a moving entity is non-roaming then let length = `__ttl_grounded`. The variable `__game_end_time` is set when a moving entity is created and it is set to

```
__game_end_time = current_game_time + length;
```

So if the moving entity is roaming and the `__game_end_time` is reached, the moving entity will be removed. On the other hand if it is non-roaming and the `__game_end_time` is reached, the moving entity will be despawned and it will evetually be respanwed. The exactly time of the respawning is a little complicated (see the section about `__respawn_length` for more).

Note that `__game_end_time` is in game time, not in a level's time.

Suppose we want to accomplish the following: we have a moving entity type called "goblin" and we want some goblins to be removed 5 minutes after their creation and others to be removed 7 minutes after their creation. Since `__ttl` and `__ttl_grounded` are both static, we cannot modify these to accomplish this task. Instead, we must manually set `__game_end_time`. This can be done as follows (inside a game Lua script):

```
ga_ment_start(level, pos, "goblin")

local game_time = ga_get_sys_f("game.time.total")
local len = 60*5
if( should_make_long ) then len = 60*7 end
ga_ment_init_set_f("__game_end_time", game_time + len)

ga_ment_end()
```

Once `__game_end_time` is set to a positive value for a moving entity, the engine ignores `__ttl` and `__ttl_grounded` for that moving entity. However, if you manually set `__game_end_time`, this must be done during the initialization phase as in the example above.

You can also set `__game_end_time` during chunk generation. To do this, you might find the chunk generation API function `get_game_time` to be useful.

Let us remind the reader that the engine currently has the following quirk: Chunks not in the active chunk tree cannot be modified. So, when a moving entity is created by procedural world generation and it moves from its starting chunk, a record will be placed in the starting chunk saying that the moving entity can be recreated again at the time of the game end time of the moving entity. There is currently no way to prematurely remove the moving entity that has moved and have it respawn immediately. We need to wait for the game end time for it to respawn.

13.7.7 __respawn_length

```
float __respawn_length = 60*60 (one hour)
```

The variable `__respawn_length` only applies to non-roaming entities (entities created by procedural world generation). Non-roaming entities get "reverted" at the game time `__game_end_time`. However it is possible for non-roaming moving entities to be "**removed**" beforehand.

Once we remove a non-roaming moving entity that is still in its starting chunk, then it will remain gone and will not respawn for `__respawn_length`

many seconds. After a non-roaming entity leaves from its starting chunk, we leave a record in the starting chunk which says that the moving entity has moved and it also says when it will be respawned in that chunk at the `__game_end_time` of the moving entity.

So if you kill a non-roaming troll that is in its original chunk (that has a one hour respawn time), walk away for a minute, then come back, then the troll will not be there (it will not be recreated by procedural world generation). However if you kill the troll, walk away for two hours and then return, then it will be there (it will have respawned).

13.7.8 __add_to_live_world_time

```
READ_ONLY float __add_to_live_world_time = -1.0
```

This is set to the current game time when the ment is added to the active chunk tree. Note that when the player loads a game, ments get re-added to the active chunk tree.

Also, if you load a chunk, move away so the chunk gets despawned, and then move back so the chunk gets created again, the ments that were saved as being in that chunk will be added to the active chunk tree again.

13.7.9 __extra_min_levels, __extra_max_levels

```
static int __extra_min_levels = 0
static int __extra_max_levels = 0
```

Both `__extra_min_levels` and `__extra_max_levels` should be non-negative integers. For every moving entity, there is a min and a max level where it can exist. The variable `__start_level` is set to the level of the starting chunk where the moving entity was created. Then immediately afterwards the `__min_level` and `__max_level` variables are set as follows:

```
__min_level = start_level - __extra_min_levels
__max_level = start_level + __extra_max_levels
```

13.7.10 __start_level, __min_level, __max_level

```
READ_ONLY int __start_level = -1
READ_ONLY int __min_level = -1
READ_ONLY int __max_level = -1
```

These are described in the previous subsection.

13.7.11 __level, __chunk_id

```
READ_ONLY int __level = -1
READ_ONLY int __chunk_id = -1
```

These describe the level that the moving entity is in together with the `chunk_id` of the chunk which contains the moving entity. These variables are not saved and loaded in the usual way.

13.7.12 __offset, __offset_old

```
READ_ONLY Vector __offset     = Vector(7.5, 7.5, 7.5)
READ_ONLY Vector __offset_old = Vector(7.5, 7.5, 7.5)
```

Let's say the moving entity is in chunk C. The variable `__offset` describes the position of the entity inside chunk C. The variable `__offset_old` describes the position of the entity during the last discrete update (but still with respect to chunk C). The reason we have both `__offset` and `__offset_old` is because during rendering, we render a moving entity as an interpolation between two updates. So even though the game has only 25 discrete updates per second, we can achieve a higher frame rate by interpolation.

These variables are not saved and loaded in the usual way. For example, the "**base chunk**" is the unique chunk that contains the moving entity that is on level `__min_level`. When we save a game we save a moving entity in the chunk file associated to the base chunk of the entity. The offset that is stored is relative to that base chunk.

13.7.13 __hide

When this is true, then 1) the ment is not rendered, 2) the ment's hard coded engine update is not processed, 3) the player and other moving entities cannot collide with it, 4) the ment does not process "alarms", and 5) the ment does not show up in moving entity range queries. However their Lua `__on_update` function is called, which is one way you can use to unhide them again.

13.7.14 __vel

```
Vector __vel = Vector(0.0, 0.0, 0.0)
```

This is the velocity of the moving entity. During each discrete update phase, the engine will attempt to move the moving entity according to this velocity.

```
bool __vel_disabled = false
```

When `__vel_disabled` is true, the ment will behave as if it has the zero vector for its velocity. The reason we have this variable is to easily pause the movement of a ment while being able to remember its velocity.

13.7.15 __mesh

```
string __mesh = ""
```

The `__mesh` variable determines the mesh name to be used for the moving entity. The corresponding mesh should be listed in "`Meshes/mesh_names.txt`". Note that the file `mesh_names.txt` associates each mesh name to a wavefront.obj file (for the mesh itself) as well as a texture name.

When the package is loaded, the `__mesh` variable for a moving entity type is set to the name of that moving entity. So for example, if the moving entity troll.lua does not define `__mesh`, then by default the `__mesh` variable for troll moving entities is "troll".

If mesh is the empty string, then the moving entity will be invisible.

13.7.16 __alpha

```
float __alpha = 1.0
```

If this is 1.0, the ment is rendered normally. Otherwise, if it < 1.0 and ≥ 0.0, then it is rendered partially transparent.

13.7.17 __tex_override

```
string __tex_override = ""
```

Recall that the mesh name `__mesh` of the moving entity is associated to a wavefront.obj file and a texture name. The file

`Meshes/mesh_names.txt`

defines these associations. When `__tex_override` is not the empty string, the same wavefront.obj is used but instead the string `__tex_override` will be used for the texture name. When `__tex_override` is the empty string, the original texture name associated to the mesh name is used.

This can be used for creating a freezing gun which, when it hits a monster, it sets the monster's `__tex_override` string to the name of an ice texture.

13.7.18 __force_no_lighting

```
bool __force_no_lighting = false
```

Recall that the user can choose to apply directional lighting to ments. However, when this variable is true, the ment is fully lit (assuming it does not have a custom render function).

13.7.19 __min_render_dist, __max_render_dist

```
float __min_render_dist = -1.0
float __max_render_dist = -1.0
```

If `__min_render_dist` is > 0.0, then the ment is not rendered if its distance to the player is less than that distance.

If `__min_render_dist` is > 0.0 AND `__max_render_dist` > 0.0 as well, then the ment is not rendered if its distance to the player is *greater* than that distance. The reason why we require `__min_render_dist > 0` here is for performance reasons.

13.7.20 __max_screen_size, __max_screen_size_time_len

```
float __max_screen_size          = -1.0
float __max_screen_size_time_len = -1.0
```

When `__max_screen_size` is > 0.0, the ment is not rendered if its apparent size of the screen is that much. For example, if this is set to 1.0, then the ment is not rendered if its length in screen space is the width of the screen (roughly).

This check only takes place for X seconds after the ment was added to the live world (the active chunk tree), where X is the value of the max screen size time length variable.

The typical use of these two functions is when the player launches a rocket. We do not want to render the rocket if it is too close to the camera.

13.7.21 __team_id_source, __team_id_target

```
int __team_id_source = 0
int __team_id_target = 0
```

Team 0 is neutral, team 1 is the player, team 2 is typical monsters. Typically for the hitter entity to attack a hittie entity, then 1) the hitter must have a non-zero source team id, 2) the hittie must have a non-zero target team id, and 3) the source team id and the target team id must be different.

Consider a monster's rocket projectile. It would have `__team_id_source` $= 2$. If that rocket has `__team_id_target` $= 0$, then the player cannot shoot the rocket down. On the other hand, if the rocket has `__team_id_target` $= 2$, then the player can shoot down the rocket.

Note that in terms of one moving entity hitting another in the usual collision detection system, it is up to you to enforce this convention about the team source id of the hitter and the team target id of the hittie. That is, the `__on_ment_hit` function of the hitter moving entity should look at these team ids and if there is a mismatch then the function should return false (specifying that the hit is not terminal).

13.7.22 __collides

```
bool __collides         = true
```

There are several collision related variables:

- `__collides`,
- `__solid_wrt_player`,
- `__point_block_correct`, and
- `__ment_correct`.

When the variable `__collides` is true, when we try to move the ment, we will see if it collides with other objects. Such collisions result in the callback functions being called, such as `__on_block_hit` and `__on_ment_hit` functions of the moving entity being called when there is a collision. That is, these collisions are detected by moving the moving entity while keeping everything else in the world still. If one of these callback functions returns true, which signifies that the collision is "terminal", then the ment will stop moving at the location of the collision. Note that during this collision detection, the ment is treated as a moving *point*, not a moving sphere.

So note that `__collides` being true does *not* mean the ment will be pushed away from blocks and other ments. For that, see `__point_block_correct` and `__ment_correct`.

13.7.23 __player_can_telefrag

```
bool __player_can_telefrag
```

When this is true, the player is allowed to telefrag the ment. One way to telefrag an ment is by growing right next to it. It also might be possible to telefrag by saving and loading. When this variable is false and the player attempts to telefrag the ment, the player will be teleported back to their last "safe position".

For the most part, all ments should probably have this variable set to true, except a handful of boss monsters to prevent cheesing.

13.7.24 __solid_wrt_player

```
bool __solid_wrt_player = false
```

The variable `__solid_wrt_player` is true iff the player cannot move through the moving entity. When this is true, the ment is physically modeled as a sphere with a certain radius (the radius given by the function `ga_ment_get_radius`). That is, when we try to move the player, a moving entity with this variable set to true is treated as a solid immovable sphere that the player cannot move through.

13.7.25 __point_block_correct and __ment_correct

```
bool __point_block_correct = false (rl = 1 minute)
bool __ment_correct        = false (rl = 1 minute)
```

If `__ment_correct` is set to true, then once every discrete update, we will push the moving entity away from all other moving entities, as if the other moving entities are immovable solid spheres.

For `__point_block_correct`, we cheat to make the math simpler. That is, when this variable is set to true, then the *center* of the moving entity is pushed away from solid blocks (once every discrete update). We try to move the ment so that its center is no longer inside of any solid block. However the sphere of the ment could still be intersecting blocks.

13.7.26 __radius, __radius_lvlinv

```
float __radius        = 1.0
bool  __radius_lvlinv = false
```

Every moving entity is modeled as a sphere from the point of view of the engine. The variable `__radius` and `__radius_lvlinv` determine the radius of this sphere.

Specifically, suppose `__radius_lvlinv` is true. Then no matter what level the moving entity is on, it will have radius `__radius`. lvlinv stands for "level invariance".

On the other hand suppose `__radius_lvlinv` is false. Then when the moving entity is on its starting level, it will have radius `__radius`. However when the moving entity moves either up or down in level, the radius will change accordingly. For example, when the moving entity moves from level 53 to 54, its radius will be scaled by a factor of 16.0. This way when the moving entity moves from one level to another, it will look smooth and the player will not notice any change.

13.7.27 __homing, etc

```
bool          __homing                 = false
float         __homing_speed           = 1.0
float         __homing_min_dist        = 0.0
READ_ONLY int __homing_target          = -1
bool          __homing_player_pathing  = false
bool          __homing_player_vis_test = false
bool          __homing_only_diff_level = false
```

When `__homing` is true, the moving entity will turn towards targets (but still with the same speed). The variables `__team_id_source` and `__team_id_target` are used for this. For example, when `__homing` is true and `__team_id_source` is 1 (1 represents the player), then the moving entity will be attracted to all moving entities whose `__team_id_target` variables are 2.

The variable `__homing_target` is used to track what moving entity the current moving entity is homing towards. If this is ≥ 0, the moving entity is homing towards the moving entity with that instance id. If `__homing_target` is -1, then

the moving entity has not yet tried to acquire a target. If `__homing_target` is -2, then the moving entity tried to acquire a target but failed.

The variable `__homing_speed` is the speed at which the ment moves when it is homing (it does not use the previous length of the `__vel` vector).

`__homing_min_dist` is the closest the ment can be to its target before it temporarily stops.

If `__homing_player_pathing` is true and the ment is homing towards the player, then the ment will go around corners to reach the player.

If `__homing_player_vis_test` is true, the ment must be visible to the player in order for the ment to move.

If `__homing_only_diff_level` is true, the ment only moves if it has a different level than its target.

13.7.28 __gas_cloud_period, etc

```
float           __gas_cloud_period       = -1.0
READ_ONLY float __gas_cloud_last_time    = -1.0
float           __gas_cloud_ttl          = 2.0
Vector          __gas_cloud_color        = Vector(1.0, 1.0, 0.0)
float           __gas_cloud_radius       = 2.0
float           __gas_cloud_trigger_dist = 48
```

When `__gas_cloud_period` is > 0.0, a cloud of particles is emitted by the ment whenever the player is near. The variable `__gas_cloud_last_time` is used internally. The variable `__gas_cloud_ttl` specifies how many seconds each particle lasts. The variable `__gas_cloud_color` specifies the color of the particles. The variable `__gas_cloud_radius` specifies how far away the particles are from the ment center. The variable `__gas_cloud_trigger_dist` specifies how close the player must be to the ment for it to emit particles.

13.7.29 __turn_speed, __turn_towards_player, __turning_disabled

```
float  __turn_speed               = 1.0
bool   __turning_disabled         = false
bool   __turn_towards_player      = false
Vector __turn_towards_player_axis = Vector(1,0,0)
bool   __turn_around_vel          = false
Vector __turn_around_vel_axis     = Vector(0,0,1)
```

When a moving entity is put in the world, it starts with a random orientation. Its "turning speed" is specified by `__turn_speed`. When `__turning_disabled` is true, turning is disabled.

If `__turn_towards_player` is true, then the ment will turn towards the player with a speed specified by `__turn_speed`. That is, it will rotate the "Axis" of the ment's frame towards the player, where "Axis" is the vector variable `__turn_towards_player_axis`. Note that `__turn_towards_player_axis`

is not in world coordinates, but rather in the coordinates of the mesh of the moving entity. For example, $(1, 0, 0)$ is on the x-axis of the mesh. This is overrided if `__turn_around_vel` is true, in which case the ment will turn about the axis of the velocity vector of the ment (with the given turn speed). Also when `__turn_around_vel` is true, the "Axis" of the ment's frame will point in the direction of the velocity vector (so the ment will be rotating about the "Axis" of its frame). Here "Axis" is the value of the vector variable `__turn_around_vel_axis`. Note that this axis is in the coordinate system of the mesh. It is $(0, 0, 1)$ by default. For example, when a rocket moves through the world, it points in the direction of its velocity and its mesh's z-axis points in the direction of the velocity.

13.7.30 __mesh_fixed_frame, __mesh_fixed_frame_vX

```
bool   __mesh_fixed_frame    = false
Vector __mesh_fixed_frame_v1 = Vector(1.0, 0.0, 0.0)
Vector __mesh_fixed_frame_v2 = Vector(0.0, 1.0, 0.0)
Vector __mesh_fixed_frame_v3 = Vector(0.0, 0.0, 1.0)
```

Use these functions to micromanage the orientation of the ment. When `__mesh_fixed_frame` is true, the orientation of the moving entity will be overridden. In this case, the orientation is specified by the three 'fixed frame" vectors (which should be orthogonal and have length one).

13.7.31 __towards_viewerXXX and __dist_to_viewerXXX

```
READ_ONLY bool   __towards_viewer_valid = false
READ_ONLY Vector __towards_viewer_vec   = Vector(0.0, 0.0, 0.0)
READ_ONLY Vector __towards_viewer_dir   = Vector(0.0, 0.0, 1.0)
READ_ONLY float  __dist_to_viewer       = -1.0
READ_ONLY float  __dist_to_viewer_old   = -1.0
```

Not only are these read only, but you should not even read from these. Instead you should use the following Game Lua-to-C API functions (but that may change in a later version of the game):

```
Vector ga_ment_get_var_special_vec_to_viewer(int inst_id);
float ga_ment_get_var_special_dist_to_viewer(int inst_id);
```

The variables `__towards_viewer_XXX` and `__dist_to_viewer` are used to cache values which these functions can return. The purpose of having both `__dist_to_viewer` and `__dist_to_viewer_old` is so that the `__on_closest` function can be called appropriately.

Note: none of these variables are saved to file.

13.7.32 __death_animXXX

```
string        __death_anim                   = ""
READ_ONLY int __death_anim_stage             = 0
float         __death_anim_start             = -1.0
float         __death_anim_end               = -1.0
float         __death_anim_alpha_fade_alpha1 = 1.0
float         __death_anim_alpha_fade_alpha2 = 1.0
```

When a moving entity "dies", it can optionally use a "death animation". Note: the rendering is done by the engine. Right now here the possible options for `__death_anim`:

- "`alpha_fade`"
- "`dark_hole`"

When `__death_anim` is set to "`dark_hole`", the moving entity will gradually shrink until it is a single point. The time `__death_anim_start` (in game time) specifies when the shrinking starts and the `__death_anim_end` specifies when the shrinking ends (and the entity has become a single point). The variable `__death_anim_stage` is used by the engine to track which stage of the death animation we are in.

When `__death_anim` is set to "`alpha_fade`", the ment will have its alpha be set to a value between

`__death_anim_alpha_fade_alpha1`

and

`__death_anim_alpha_fade_alpha2`

for each time that is between `__death_anim_start` and `__death_anim_end`.

Here is code for a "black hole bullet" moving entity which, when it hits another moving entity, it will cause that entity to shrink to a point.

```
bool function p.__on_ment_hit(
    hitter_inst_id, hittie_inst_id,
    level, lp, normal)
--
    local hittie_type = ga_ment_get_type(hittie_inst_id)
    if not ga_ment_var_exists(hittie_type, "health") then return end

    ga_ment_set_i(hittie_inst_id, "health", 0) --Killing the hittie.

    --Special dark hole death animation.
    --The hittie entity will shrink down to a point
    --for the next 2 seconds.
    ga_ment_set_s(hittie_inst_id, "__death_anim", "dark_hole")
    local game_time = ga_get_game_time()
```

```
    ga_ment_set_f(hittie_inst_id, "__death_anim_start", game_time)
    ga_ment_set_f(hittie_inst_id, "__death_anim_end", game_time + 2.0)

    --It is a terminal hit
    --(the bullet will stop now).
    return true
end
```

Chapter 14

Window Lua Scripts

14.1 Introduction

There are two types of windows:

- Game windows
- HUD windows

All window Lua scripts are put in the `Windows` directory of the package.

14.1.1 Window IDs (WIDs)

Every window has a window id (a `wid`). Consider a game window, for example. The first time a game window associated to a given window Lua script is pushed onto the game window stack, that window will be assigned a wid.

Note that a wid is associated to the window Lua script itself, not to the instance of the window.

So if we add a window to the game window stack, then pop it off, then push it on again, it should be assigned the same wid in the end.

14.1.2 Stacks vs Sets

The game windows are put into the "window stack". Only the top window in this stack is rendered, and only the top window in this stack is given user input. Windows can be "pushed" onto or "popped" from this stack.

HUD windows, on the other hand, exist in a "set". All HUD windows in the set are rendered when the player is in the game (and there are no windows in the main menu window stack or the game window stack). Care must be taken to specify in which order the HUD windows are rendered (one is rendered on top of another).

14.2 More on HUD Windows

HUD windows are similar to game windows, except they do not have `__start` and `__end` functions. Instead, they have the analogous functions `__hud_add` and `__hud_remove`. They do have the functions `__process_input` and `__render` functions. The render function is only called when the window stack is empty. The process input function is normally only called when there are no windows on the window stack (because normally the game is paused when the window stack is non-empty, although you can configure this).

14.3 Window Script Functions

When you are in a game, if you go to MAIN MENU → OPTIONS → PACKAGE TOP MENU you will be able to access a special window that the current package defines, called `main_menu.lua` (in the `Windows` directory of the package).

Here are the window lua script module functions (that are called in the Game Lua state):

```
//-----------------------------------------------
//              Window Script Functions
//-----------------------------------------------
string __get_name(int wid);
void __start(int wid);                  //Game windows only.
void __end(int wid);                    //Game windows only.
void __pop(int wid);                    //Game windows only.
void __pushed_upon(int wid);            //Game windows only.
void __hud_add(int wid);                //HUD windows only.
void __hud_remove(int wid);             //HUD windows only.
void __process_input(int wid);
void __render(int wid);
void __update_always(int wid);
void __update_on_stack(int wid);        //Game windows only.
void __update_on_stack_top(int wid);    //Game windows only.
void __update_on_hud(int wid);          //HUD windows only.
```

The follwing module functions should NOT be used:

```
//-----------------------------------------------
//     Deprecated Windows Module Functions
//-----------------------------------------------
void __on_start(int wid); //Renamed to __start.
void __on_end(int wid);   //Renamed to __end.
```

14.3.1 p.__get_name

```
string __get_name(int wid);
```

The official name of a window is simply the name of the Lua file that defines it, without the `.lua` at the end.

However the function `__get_name` should return a human readable and friendly version of the window name. This is used, for example, when displaying the list of HUD Side Displays.

14.3.2 p.__start

```
void __start(int wid);
```

Consider the lua script `my_window.lua`. When a window of this type is pushed onto the window stack, then this `__start` function will be called. This function is passed the window id (wid) of the window.

Note that it is intended that there is at most one instance of a window for each window Lua script.

Here is what the `p.__start` function of `my_window.lua` might look like:

```
function p.__start(wid)
    ga_play_sound_menu("chimes_sound")
end
```

Here the `__start` function is playing the sound `chimes_sound`. There are two ways to play sounds (that are not "music"): 1) `ga_play_sound` and 2) `ga_play_sound_menu`. The `ga_play_sound` functions plays a "game" sound whereas the `ga_play_sound_menu` functions plays a "menu" sound. When the player is in a menu, all game sounds are paused (and so only menu sounds can be played).

14.3.3 p.__end

```
void __end(int wid);
```

When a window is popped from the window stack, the `__end` function of that popped window will be called (immediately before it is actually popped).

There is another situation in which `__end` is called. If a window W is on top of the window stack and another window is pushed on top of it, then the `__end` function of W will be called.

Here is what the `p.__end` function of `my_window.lua` might look like:

```
function p.__on_end(wid)
    ga_play_sound_menu("sad_sound")
end
```

See also `__pop` and `__pushed_upon`.

14.3.4 p.__pop and p.__pushed_upon

```
void __pop(int wid);
void __pushed_upon(int wid);
```

Recall that `__end` is called in two situations. The functions `__pop` and `__pushed_upon` are called in these two situations individually. That is, the `__pop` function of a window is called when that window is popped from the window stack (or more precisely immediately before it is popped from the stack). On the other hand, the `__pushed_upon` function of a window W is called immediately before another window is pushed on top of the window W.

14.3.5 p.__hud_add and p.__hud_remove

```
void __hud_add(int wid);       //HUD windows only.
void __hud_remove(int wid);    //HUD windows only.
```

These functions are only called for HUD windows. The `__hud_add` function is called when the window is added to the HUD set and the `__hud_remove` function is called when it is removed from that set.

14.3.6 p.__process_input

```
void __process_input(int wid);
```

The `__process_input` function of a main menu window is called when it is time to process user input (keyboard and mouse). It is called *only* if the window is on TOP of the main menu window stack.

Here is possible code from the lua script `my_window.lua`:

```
function p.__process_input(wid)
    if ga_win_key_pressed(wid, "ESC") then
        ga_window_pop_all()
        return
    end

    if ga_win_key_pressed(wid, "X") then
        ga_exit() --Exit the program.
    end
end
```

Here, when the `my_window.lua` is on top of the main menu window stack, if the player pressed the ESCAPE key then all windows on the main menu window stack will be popped and the player will return to the game. On the other hand, if the player pressed the X key, then the program will exit.

14.3.7 p.__render

```
void __render(int wid);
```

The render function of a main menu window is called when it is time to render the window. It is called *only* if the window is on TOP of the window stack (or is in the HUD set).

Here is possible code from the lua script `my_window.lua`:

```
function p.__render(wid)
    ga_win_set_char_size(wid, 0.04, 0.08)
    ga_win_txt_center(wid, 0.6, "PRESS ESCAPE TO GO BACK TO THE GAME")
    ga_win_txt_center(wid, 0.3, "PRESS X TO EXIT THE GAME")
end
```

Here the `ga_win_set_char_size` function sets the text character size to be such that letters have width 0.04 and height 0.08 (1.0 is the width of the screen and 1.0 is the height of the screen).

The `ga_win_txt_center` renders text which is left-to-right centered in the middle of the screen, and it has the minimum y coordinate as specified.

14.3.8 p.__update_always

```
void __update_always(int wid);
```

This function on each window is called, even if the game is paused.

This function is called every cycle (which happens more often than every *discrete update*).

Here is an example:

```
function p.__update_always(wid)
    if ga_get_game_paused() then return end
    ga_print("Doing some updating.")
end
```

If you want more finely grained functions like `__update`, `__update_passive`, `__update_discrete_pre`, and `__update_discrete_post`, then put those functions in a Game Lua Script (a script in the Game directory) and have those functions call functions in your window script.

14.3.9 p.__update_on_stack

```
void __update_on_stack(int wid);
```

This is called when the window is on the window stack (but not necessarily on the top of the stack). This is called every cycle (which happens more often than every *discrete update*).

14.3.10 p.__update_on_stack_top

```
void __update_on_stack_top(int wid);
```

This is called when the window is on *top* the window stack. This is called every cycle (which happens more often than every *discrete update*).

14.3.11 p.__update_on_hud

This is called when the window is in the HUD set. This is called every cycle (which happens more often than every *discrete update*).

14.3.12 An Example

This is what the `Windows/main_menu.lua` window script might look like:

```
function p.__on_start(wid)
    local min_y = 0.25
    local max_y = 0.75
    local char_w = 0.03
    local char_h = 0.06
    local color = std.vec(0.0, 0.5, 0.5) --RGB.
    options = {
        "GET FREE GOLD",
        "LIST OF CHEAT CODES }
    ga_win_widget_small_list_start(
        wid, min_y, max_y, char_w, char_h,
        color, options)
end

function p.__on_end(wid)
    --Nothing to do.
end

function p.__process_input(wid)
    local selection = ga_win_widget_small_list_process_input(wid)
    local selection_str = ga_win_widget_small_list_entry(wid, selection)
    if( selection_str == "GET FREE GOLD" ) then
        ga_window_push("win_get_free_gold")
        return
    end
    if( selection_str == "LIST OF CHEAT CODES" )
        ga_window_push("win_list_of_cheatcodes")
        return
    end
    if ga_win_key_pressed(wid, "ESC") then
        --Popping this window from the main menu window stack.
```

```
        ga_window_pop()
        return
    end
end

function p.__render(wid)
    ga_win_set_char_size(wid, 0.04, 0.08)
    ga_win_txt_center(wid, 0.85, "MAIN MENU")

    --The small list widget will automatically be rendered.
end
```

You can read about the small list widget in Chapter 18.

Chapter 15

Game Lua Scripts

15.1 Introduction

Recall that packages have the folder called "Game". One purpose of this folder is to define helper Lua function for use in other scripts.

The second purpose is to contain the file

"top.lua".

This top.lua script contains functions that are called by the engine at various points in time. In this chapter we will describe these functions. Functions in scripts other than top.lua are called by the engine also.

15.2 All top.lua Module Functions

```
//-----------------------------------------------
//             Game/top.lua Functions
//-----------------------------------------------
void top.__new_game();
void top.__load_game();
bool top___reboot_game();
void top.__update();
void top.__update_passive();
void top.__update_discrete_pre();
void top.__update_discrete_post();
string top.__game_input(string cmd_str);
string top.__game_input_get_all_cmds();
string top.__game_input_get_help_str(string cmd_name);
void top.__killed_player();
void top.__respawn_player(string param);
Vector top.__get_level_color(int level, string side);
```

```
//-----------------------------------------------
//     Other "Game" Script Functions
//-----------------------------------------------
void __load_game_early();
void __load_game();
void __update();
void __update_passive();
void __update_discrete_pre();
void __update_discrete_post();
void __render_augmented(int level);
```

15.3 top.__new_game

```
void top.__new_game();
```

This function is called when the player first creates a game. That is, this function is only called once. When the player then loads the game later, the __load_game function will be called (NOT the __new_game function). Something you probably want to do in the new_game function is search the world for a suitable starting position for the player.

Here is what the __new_game function might look like (in Game/top.lua):

```
function p.__new_game()
    --Setting the heath is not needed as long
    --as the globals.txt file sets it accordingly.
    ga_set_i("health", 200)

    --Setting body to "fly" mode
    --and the camera mode to use true up.
    local trans = std.vec(0.0, 0.0, 0.0)
    local radius = 0.3
    local use_true_up = true
    ga_move_set_body_fly(trans, radius, use_true_up)
end
```

15.4 top.__load_game

```
void top.load_game();
```

The __load_game function is called each time the player loads the game (just after the engine finished the load). Here is an example of what the load function might look like:

```
function p.__load_game()
    --Adding windows to the hud.
```

```
    --This is needed because when the game is loaded,
    --the game and window stacks and the hud window set
    --are initially empty.
    ga_hud_window_add("win_hud", 0)
end
```

15.5 top.__reboot_game

This is described in Section 17.38.

When the player "reboots" their game, the function `top.__reboot_game` will be called over and over again until it return false. The purpose of the function `reboot_game` is to specify which dynamic variables should be saved. Once `__reboot_game` returns false, the function `top.__new_game` is called.

15.6 top.__update

```
void top.__update();
```

When the player is in normal game play, the game calls an update function every cycle. This could be called 60 or perhaps 100 or more times per second. On the other hand, there are also "discrete updates" which occur exactly 25 times per second. The functions top.update_discrete_pre and top.update_discrete_post before and after the engine performs this discrete update.

15.7 top.__update_passive

```
void top.__update_passive();
```

This function is called when the player is in either a game menu or a main menu. Note that when the player is in a menu, the game time is "frozen" (the world should stand still, for the most part).

15.8 top.__update_discrete_pre

```
void top.__update_discrete_pre();
```

This function is called just before the engine does a discrete update. Note that there should be exactly 25 discrete updates per second (when the player is in normal game play).

15.9 top.__update_discrete_post

```
void top.__update_discrete_post();
```

This function is called just after the engine does a discrete update. Here is what the update discrete functions might do in Game/top.lua:

```
function p.__update_discrete_pre()
    --Nothing to do.
end

function p.__update_discrete_post()
    --Moving the player.
    local travel = std.vec(0.0, 0.0, 0.0)
    --Set the travel function depending
    --on what keys are pressed...
    ga_move_set_desired_travel(travel)
end
```

15.10 top.__game_input

```
string top.__game_input(string str);
```

The primary way that the engine gives commands to the package is via the game_input function. This function returns output in the form of a string.

The system command

game_input str

causes top.game_input to be called with the input string str. This can be used for binding key and mouse events to game actions. For example, in the file binds.txt we can have the following line:

```
PACKAGE_MOVE_JUMP     SPACE.downup   "" "game_input jump" ""
```

Then if the player pressed the space bar during normal game play, then the command "game_input jump" will be executed, which causes the top.game_input function to be called with the input string "jump". It is up to top.game_input on how to interpret this command.

There are some commands that are called by the engine in certain circumstances. These command strings start and end with double underscores. For example, __game_input will be passed the following strings by the engine in the appropriate circumstances:

```
__game_saved__
__spiral_of_death__
__screenshot__
__screenshot_failed__
```

The expected response of top.__game_input to being passed these strings is to display a message on the HUD to the user. "Spiral of death" refers to the situation when it takes too long to process a discrete game update, so the engine tries to perform several updates simultaneously to make up time.

15.11 top.__game_input_get_all_cmds

```
string top.__game_input_get_all_cmds();
```

This function should return a list of all the "commands" supported by the package. These should be separated by semicolons (without a semicolon at the end). For example, it might return the following:

fly;run;shoot

15.12 top.__game_input_get_help_str

```
string top.__game_input_get_help_str(string cmd_name);
```

Returns a string documenting the specified command. Here is an example:

```
function p.__game_input_get_help_str(cmd_name)
    if( cmd_name == "blue" ) then
        return
                "Usage: blue\n\n"
            .. "Cheat code that causes the player "
            .. "to teleport as if they used blue rings. "
    end
    -- ...
end
```

15.13 top.__killed_player

```
int top.__killed_player();
```

To (try to) kill the player, call the ga_kill_player Game Lua-to-C API function. If game.player.alive is false, nothing will happen. If god mode is on, nothing will happen. Otherwise game.player.alive will be set to false and top.killed_player will be called.

15.14 top.__respawn_player

```
int top.__respawn_player(string param);
```

Once the player is dead, to respawn the player must call either the system command

`respawn passive`

or the system command

`respawn force`.

The respawn command optionally takes a second string argument param, which is passed to `top.__respawn_player`. Then the engine will respawn the player (which includes placing them at their respawn point). After all this, the engine will call the function `top.__respawn_player`. Here is an example of what `top.respawn_player` might look like:

```
function p.respawn_player(str_param)
    ga_set_i("health", 100)
    ga_set_i("bullets", 0)
    ga_set_i("shells", 0)
    ga_set_i("rockets", 0)
end
```

15.15 top.__get_level_color

```
Vector top.__get_level_color(int level, string side);
```

This is called when the engine needs to determine how to shade blocks from a given level. The `side` string should be one of the following: `x_pos`, `x_neg`, `y_pos`, `y_neg`, `z_pos`, `z_neg`.

15.16 other.__load_game_early

```
void other.__load_game_early();
```

If a game script (other than Game/top.lua) has this function, it will be called when the game is loaded, *before top.load_game*.

15.17 other.__load_game

```
void other.__load_game();
```

If a game script (other than Game/top.lua) has this function, it will be called when the game is loaded, *after top.load_game*.

15.18 The order in which load_game functions are called

To summarize, this is the order in which load functions are called:

1) The __load_game_early functions in all game scripts other than "Game/top.lua".

2) The function __load_game in "Game/top.lua".

3) The __load_game functions in all game scripts other than "Game/top.lua".

15.19 other.__update

```
void other.__update();
```

This is just like top.__update, except it is for game scripts other than top. That is, it is called every frame when the game is unpaused.

15.20 other.__update_passive

```
void other.__update_passive();
```

This is just like top.__update_passive, except it is for game scripts other than top. That is, it is called every frame when the game is paused.

15.21 other.__update_discrete_pre and post

```
void __update_discrete_pre();
void __update_discrete_post();
```

These are just like their "top" versions, except they are for the game scripts other than top. Here is the order in which discrete update functions get performed:

1) __update_discrete_pre gets called for every game script other than top.lua.

2) top.__update_discrete_pre gets called.

3) The engine performs its internal discrete update.

4) top.__update_discrete_post gets called.

5) __update_discrete_post gets called for every game script other than top.lua.

There are about 25 discrete updates per second. These only occur when the game is unpaused.

15.22 other.__render_augmented

```
void __render_augmented(int level);
```

This function is called when the finest level of detail is being rendered. Right now, the argument "level" is the same as the viewer level. Specifically, it is used when opaque objects are being rendered on the level of the player. Within this function you can call functions like the following to render objects:

- ga_render_push_matrix,
- ga_render_pop_matrix,

- ga_render_matrix_translated, and
- ga_render_mesh.

This is all in the coordinate system of the level of the player.

So for example, if you "push the matrix", translate by (13.5, 7.5, 9.5), render a sphere mesh (whose model is centered at the origin), then "pop the matrix", what will happen is a sphere will be rendered at the block position (13,7,9).

In a later version of the engine we may allow rendering on multiple levels. In that case, the level argument to the function would be important.

Chapter 16

The Initialization Lua-to-C API

When the package is loaded, certain functions in lua scripts are called to initialize various things. For example, consider a script "MovingEnts/bullet.lua". This has a function p.type_init which is called when the package is initialized. This function should in turn call functions that are part of the Initialization Lua-to-C API to initialize various aspects of bullet type moving entities.

Functions in the Initialization Lua-to-C API can only be called at certain times. One time is when the game calls bullet.type_init, etc.

Note: **ia_** stands for the "Initialization API".

16.1 The Full Initialization Lua-to-C API

```
//-----------------------------------------------
//       Initializing Moving Entity Types
//-----------------------------------------------

void ia_ment_new_var_b(int tid, string var, bool default_value, float revert_length);
void ia_ment_new_var_i(int tid, string var, int default_value, float revert_length);
void ia_ment_new_var_f(int tid, string var, float default_value, float revert_length);
void ia_ment_new_var_v(int tid, string var, Vector default_value, float revert_length);
void ia_ment_new_var_s(int tid, string var, string default_value, float revert_length);

void ia_ment_new_var_b_perm(int tid, string var, bool default_value);
void ia_ment_new_var_i_perm(int tid, string var, int default_value);
void ia_ment_new_var_f_perm(int tid, string var, float default_value);
void ia_ment_new_var_v_perm(int tid, string var, Vector default_value);
void ia_ment_new_var_s_perm(int tid, string var, string default_value);

void ia_ment_new_static_var_b(int tid, string var, bool value);
```

```
void ia_ment_new_static_var_i(int tid, string var, int value);
void ia_ment_new_static_var_f(int tid, string var, float value);
void ia_ment_new_static_var_v(int tid, string var, Vector value);
void ia_ment_new_static_var_s(int tid, string var, string value);

void ia_ment_set_builtin_var_b(int tid, string var, bool value);
void ia_ment_set_builtin_var_i(int tid, string var, int value);
void ia_ment_set_builtin_var_f(int tid, string var, float value);
void ia_ment_set_builtin_var_v(int tid, string var, Vector value);
void ia_ment_set_builtin_var_s(int tid, string var, string value);

void ia_ment_set_var_saving(int tid, string var, bool value);

void ia_ment_set_var_rl_only(int tid, string var, float revert_length);

void ia_ment_set_var_changed_cb(int tid, string var, string func);

//-----------------------------------------------
//          Initializing Block Types
//-----------------------------------------------

void ia_block_new_static_var_b(int tid, string var, bool value);
void ia_block_new_static_var_i(int tid, string var, int value);
void ia_block_new_static_var_f(int tid, string var, float value);
void ia_block_new_static_var_v(int tid, string var, Vector value);
void ia_block_new_static_var_s(int tid, string var, string value);

void ia_block_set_builtin_var_i(int tid, string var, int value);
void ia_block_set_builtin_var_s(int tid, string var, string value);

void ia_block_new_var_b(int tid, string var, bool value);
void ia_block_new_var_i(int tid, string var, int value);
void ia_block_new_var_f(int tid, string var, float value);
void ia_block_new_var_v(int tid, string var, Vector value);
void ia_block_new_var_s(int tid, string var, string value);

void ia_block_make_var_eph(int tid, string var, int rl);
void ia_block_make_var_not_eph(int tid, string var);
```

16.2 Moving Entity (Type) Initialization Functions

These functions are intended to be called from the type_init function of each Moving Entity Lua Script. These functions all involve variables associated to a

moving entity.

There are five types of variables for moving entities: bools (b), ints (i), floats(f), Vector(v), and strings (s). A vector is an (x,y,z) triple of floats.

The variables associated to a moving entity type must be defined via these functions before the main game begins.

Variables are either "static" or not. If a variable is static, then there is a single variable for the moving entity type which has a value. This value is associated to the type, and not the instance. For example, if the troll moving entity has the static integer variable "max_health" which is set to 200, then all trolls have their max_health variables set to 200 (and these variables cannot be changed). On the other hand, if the troll moving entity has the non-static integer variable "health" which is initially set to 200, then all trolls initially have their health variable set to 200, however this variable can change for each troll.

Consider an instance of a troll moving entity. If its (non-static) health variable is changed, then it will remain changed for a certain amount of time, called the **revert length**. After the revert length amount of time has passed, the health variable will be reset to its default value.

16.2.1 ia_ment_new_var_XXX

```
void ia_ment_new_var_b(int tid, string var, bool default_value, float revert_length);
void ia_ment_new_var_i(int tid, string var, int default_value, float revert_length);
void ia_ment_new_var_f(int tid, string var, float default_value, float revert_length);
void ia_ment_new_var_v(int tid, string var, Vector default_value, float revert_length);
void ia_ment_new_var_s(int tid, string var, string default_value, float revert_length);
```

You can use these functions to create a new (non-static) variable associated to a moving entity type.

For example, the following code in the MovingEnts/troll.lua will create the variable "health" in the troll moving entity type.

```
function p.__type_init(tid)
    ia_ment_new_var_i(tid, "health", 200, 60.0 * 60.0)
end
```

The tid is an integer id for the moving entity type. Here the health variable is set to have the default value of 200 (so new trolls that are created during game play will initially have health 200). The revert time of the variable is one hour (60*60 = 3600 seconds). Thus, if the player damages a troll (but does not kill it), then the troll's health will change and it will remain changed for one hour. After one hour, the troll's health will revert back to the default value of 200.

These functions can be called multiple times. For example, the following is valid in the troll.lua file:

```
function p.__type_init(tid)
    ia_ment_new_var_i(tid, "health", 199, 60.0 * 60.0)
```

```
    ia_ment_new_var_i(tid, "health", 200, 60.0 * 60.0)
end
```

The result of this will be that the moving entity has an integer variable called "health" which is set to the initial value 200 for each troll.

16.2.2 ia_ment_new_var_XXX_perm

```
void ia_ment_new_var_b_perm(int tid, string var, bool default_value);
void ia_ment_new_var_i_perm(int tid, string var, int default_value);
void ia_ment_new_var_f_perm(int tid, string var, float default_value);
void ia_ment_new_var_v_perm(int tid, string var, Vector default_value);
void ia_ment_new_var_s_perm(int tid, string var, string default_value);
```

These are helper functions. Each function is equivalent to calling the corresponding ia_ment_new_var_XXX function but with 100 thousand hours as a revert length.

Be careful about having revert lengths be too long, because if so then the saved game files will be filled with crap that is not needed. A simple system is to have ments have a time to live (`ttl`) of at most one hour. When an ment is removed, its variables are not saved.

16.2.3 ia_ment_new_static_var_XXX

```
void ia_ment_new_static_var_b(int tid, string var, bool value);
void ia_ment_new_static_var_i(int tid, string var, int value);
void ia_ment_new_static_var_f(int tid, string var, float value);
void ia_ment_new_static_var_v(int tid, string var, Vector value);
void ia_ment_new_static_var_s(int tid, string var, string value);
```

These functions are used to create new static variables associated to moving entities. Recall that static vars are associated to the moving entity type, not individual moving entity instances.

Consider the following type_init function of a troll moving entity Lua script:

```
function p.__type_init(tid)
    ia_ment_new_static_var_i(tid, "max_health", 200)
    ia_ment_new_var_i(tid, "health", 200, 60.0 * 60.0)
end
```

This causes moving entities of type troll to have both a max_health and a health variable (which are both integers). However although each troll has its own health value, all the trolls share the same max_health value (which is 200).

These new_static_var functions can be called multiple times, just like their non-static versions.

16.2.4 ia_ment_set_builtin_var_XXX

```
void ia_ment_set_builtin_var_b(int tid, string var, bool value);
void ia_ment_set_builtin_var_i(int tid, string var, int value);
void ia_ment_set_builtin_var_f(int tid, string var, float value);
void ia_ment_set_builtin_var_v(int tid, string var, Vector value);
void ia_ment_set_builtin_var_s(int tid, string var, string value);
```

The engine automatically creates certain variables for each moving entity. The names of all of these start with a double underscore (so you should not create your own variable starting with double underscores). See Section 13.6 for a list of all the built-in moving entity variables, and see Section 13.7 for an explanation of what these variables do.

16.2.5 ia_ment_set_var_saving

```
void ia_ment_set_var_saving(int tid, string var, bool value);
```

When in the game a change is made to the world, this needs at some point to be saved to a file. For example, if we damage a troll, that will modify its health variable and so that needs to be changed. However certain variables are not very important in the long term and so they do not need to be saved. For every moving entity variable you can change whether or not it needs to be saved when it is changed. Let's say the trolls have a variable called last_scream_time that we want to make so it is not saved to file.

Then the troll.lua file might contain the following:

```
function p.__type_init(tid)
    --The health variable (default valid = 200, revert length = one hour).
    --When the health of a troll changes, this var will be flagged
    --for saving (so during the next save it will be saved).
    ia_ment_new_var_i(tid, "health", 200, 60.0 * 60.0)

    --The last_scream_time variable.
    --The next time the game is saved,
    --this variable (for each moving entity) will NOT be saved.
    ia_ment_new_var_f(tid, "last_scream_time", 0.0, 60.0)
    ia_ment_set_var_saving(tid, "last_scream_time", false)
end
```

Static variables are not saved to file.

Also, every moving entity type has a built-in (static) variable called `__disable_saving`. When this is true, then NO variables for the moving entity type will be saved to file. Indeed, when this is true, moving entities of that type are not saved in any way to file.

16.2.6 ia_ment_set_var_rl_only

```
void ia_ment_set_var_rl_only(int tid, string var, float revert_length);
```

Use this to set the revert length of an ment variable. This can even be used for built-in variables.

16.2.7 ia_ment_set_var_changed_cb

```
void ia_ment_set_var_changed_cb(int tid, string var, string func);
```

This registers a Lua callback function to be called whenever the specified variable changes. The format of the `func` variable should be "mod.func". Let us give an example. Suppose a certain ment type has an integer variable called `health`. You can call this function setting func to be `health_logic.callback`. Then, in a Lua script called `health_logic.lua` in the Game directory of your package, you can have a function that looks like the following:

```
function p.callback(inst_id, var, old_value, new_value)
    --inst_id is the inst_id of the ment whose "health" variable has changed.
    --var is the string "health".
    --old_value was the value of the health variable before it was changed.
    --new_value is the value of the health variable after it changed.
    if( new_value <= 0 ) then
        ga_print("We need to kill the monster!")
    end
end
```

The Lua callback function actually does not get called right away. It gets called at the end of the frame in which the variable changed.

16.3 Block (Type) Initialization Functions

These functions are intended to be called from the type_init function of each Block Lua Script. These functions all involve variables associated to a moving entity.

There are five types of variables for moving entities: bools (b), ints (i), floats(f), Vector(v), and strings (s). A vector is an (x,y,z) triple of floats.

The variables associated to a block type must be defined via these functions before the main game begins.

Variables are either "static" or not. If a variable is static, then there is a single variable for the moving entity type which has a value. This value is associated to the type, and not the instance.

16.3.1 ia_block_new_var_XXX

```
void ia_block_new_var_b(int tid, string var, bool value)
void ia_block_new_var_i(int tid, string var, int value)
void ia_block_new_var_f(int tid, string var, float value)
void ia_block_new_var_v(int tid, string var, Vector value)
void ia_block_new_var_s(int tid, string var, string value)
```

Use the functions to set variables associated to a block type. The tid is the type id, which is passed as an argument to each block script's type_init function. For example, here is what the type_init function in the file block_soda_machine.lua might look like:

```
function p.__type_init(tid)
    --Initially the machine has 10 sodas.
    ia_block_new_var_i(tid, "num_sodas", 10)

    --A vector representing the location where sodas as spawned
    --when the player uses the block.
    ia_block_new_var_v(tid, "spawning_location", std.vec(1.0, 2.0, 3.0))
end
```

16.3.2 ia_block_set_builtin_var_XXX

```
void ia_block_set_builtin_var_i(int tid, string var, int value);
void ia_block_set_builtin_var_s(int tid, string var, string value);
```

Use these functions to set block variables that are "built-in". You can tell if a variable is built-in because it starts with a double underscore. Here is an example:

```
function p.__type_init(tid)
    ia_block_set_builtin_var_s(
        tid, "__special_collision_type", "ONE_WAY_X_POS")
end
```

Currently here are all the built-in block variables:

```
//-----------------------------------------------
//             Block Builtin Vars
//-----------------------------------------------
static int    __revert_length_bottom   = 60*60 (one_hour)
static int    __revert_length_default  = 60*60 (one_hour)
       string __special_collision_type = ""
       int    __which_update           = -1
```

If a block in the world has its original block type (its block type has not been changed), then if a variable in the block is modified, it will revert back to its default value in `__revert_length_bottom` many seconds.

If a block type is changed to have new block type, then the block will revert to its old block type in bt's `__revert_length_default` many seconds (unless we specify the revert time in the change block type function).

The variable `__which_update` is used for internal purposes and should not be modified.

16.3.3 ia_block_new_static_var_XXX

```
void ia_block_new_static_var_b(int tid, string var, bool value)
void ia_block_new_static_var_i(int tid, string var, int value)
void ia_block_new_static_var_f(int tid, string var, float value)
void ia_block_new_static_var_v(int tid, string var, Vector value)
void ia_block_new_static_var_s(int tid, string var, string value)
```

You use these functions just like their non-static versions. Use such a function to create a variable associated to a block type where ALL blocks of that type have the same value for this variable. For example, this is what the type init function might look like for block_wood.lua:

```
function p.__type_init(tid)
    --Creating a new static var called "description".
    ia_block_new_static_var_s(tid, "description", "This is a wood block")
end
```

All wood blocks now have the immutable value "This is a wood block" assigned to the static variable "description".

16.4 Block Stacks

Before we say anything more about blocks, it is important to understand the concept of a "block stack". Fractal Block World does not just store a single block at any given block location. Instead, it stores a "stack of blocks".

The bottom block in the stack is the original block that was created by procedural world generation. If the player then modifies that location by either digging to create an empty block or by creating a solid block there, this actually just pushes a block onto the stack. The original block from procedural world generation is still stored in the block stack (at the bottom of the stack). Every block that is added to the stack has a "revert time". Once the game time reaches this time, the block is popped from the stack.

For example, suppose the original block at a location is an air block, and then the player creates a brick block at that location. Suppose the revert time of the brick block is one hour in the future. Then in one hour, the brick block will be removed and the original air block will occupy that location.

Every block on a block stack stores variables. All of these variable need to have been registered with the block type with the ia_block_new_var_XXX or ia_block_new_static_var_XXX functions. Each one of these variables lasts the

entire lifetime the block (on a block stack) that stores it. However there is one exception. The exception is that some block variables can be set to be "ephemeral".

If a variable is ephemeral *and it is at the bottom of a block stack*, then the variable has a revert time. Once that game time is reached, the variable is reverted back to the default value of the variable.

Note also that the block at the bottom of a block stack has a revert time. When this time is reached, all variables for the block are reverted. The idea is that the revert time of a ephemeral variable should be less than the revert time of the bottom block of a block stack.

16.4.1 Ephemeral block variables

```
void ia_block_make_var_eph(int tid, string var, int rl)
void ia_block_make_var_not_eph(int tid, string var)
```

Use these function in a type_init function of a block script to make a variable either ephemeral or not ephemeral. These functions must be called after the variable is created. By default, every variable is NOT ephemeral. The number rl is the "revert length": when the variable is changed at time T, it will be reverted at time T + rl. Here is an example:

```
function p.type_init(tid)
    ia_block_new_var_i(tid, "cooldown", 10)

    --One minute revert length.
    ia_block_make_var_eph(tid, "cooldown", 60)
```

Chapter 17

The Game Lua-to-C API

In this chapter we will discuss the "Game Lua-to-C API". The Game Lua API is an API which certain Lua scripts are able to access. It provides "game" related features, such as shrinking the player, etc. The API functions are implemented in the C++ engine of the program. Here are the scripts that can use the API:

- Environment Rects (in EnvRects/)
- Basic Entites (in BasicEnts/)
- Game Lua Modules (in Game/)
- Moving Entities (in MovingEnts/)
- Windows (in Windows/)

Note: the Chunk Generation Scripts in WorldNodes/ cannot access the Game Lua API. This is because the Chunk Generation Scripts are run in separate threads. The "Game Lua modules" will be discussed in a later chapter.

17.1 The 6 Directions and 3 Axes

In the game there are 6 directions: front, back, left, right, up, down. When creating the world there are also 6 directions: x_pos, x_neg, y_pos, y_neg, z_pos, z_neg. Here are the integers associated to these:

```
x_pos -> 0
x_neg -> 1
y_pos -> 2
y_neg -> 3
z_pos -> 4
z_neg -> 5
```

Here is how to translate between these:

```
x_pos = right
x_neg = left
y_pos = front
y_neg = back
z_pos = up
z_neg = down
```

That is, the world uses a right-handed coordinate system.

You can convert between side integers and side strings using the functions

```
string std.side_int_to_str(int side_int)
int std.side_str_to_int(string side_str)
```

These are defined in the base package in the file "base/Game/std.lua".

So if a function wants a block side as an integer and we pass it the integer 4, then this represents the positive z direction.

When the user faces one of the six directions, the HUD tells the user which direction they are facing.

There are 3 axes: x,y, and z. When a function requires an "axis string", one of the following strings should be specified: "x", "y", "z".

17.2 The Full Game Lua-to-C API

```
//-----------------------------------------------
//            Program Level Functions
//-----------------------------------------------

string ga_command(string command);

void ga_save(bool play_sound);
void ga_load();
void ga_exit();
void ga_exit_with_error();

void ga_print(string line);
void ga_flush();

void ga_console_print(string line);

void ga_dump_lua_env();

void ga_debug_push(string frame);
void ga_debug_pop(string frame);
void ga_debug_line(string line);

void ga_load_user_script(string mod_name);
```

```
//-----------------------------------------------
//        Returning Values From a Function
//-----------------------------------------------

void ga_return_b(string var, bool value);

//-----------------------------------------------
//                    Time
//-----------------------------------------------

float ga_get_game_time();
float ga_get_level_time(int level);
int   ga_get_sys_time();

bool ga_get_game_paused();

int ga_get_high_precision_timer();

//-----------------------------------------------
//           Pseudo Random Functions
//-----------------------------------------------

void ga_srand(int seed);
int ga_rand();
float ga_randf();
float ga_randf_range(float min_f, float max_f);
int ga_randi(int min_i, int max_i);

int ga_chunk_id_to_seed(int chunk_id);
int ga_path_to_seed(string path);
int ga_chunk_id_and_lbp_to_seed(int chunk_id, LocalBlockPos lbp);
int ga_bp_to_seed(int level, BlockPos bp);

int ga_chunk_seed(int chunk_id);                 //Deprecated.
int ga_lbp_seed_pos(int chunk_id, int lbp_hash); //Deprecated.
int ga_block_seed_pos(int level, BlockPos bp);   //Deprecated.

//-----------------------------------------------
//               Env Vars : Globals
//-----------------------------------------------

//Setting if var exists.
bool ga_exists(string var);

//Getting the type of a var.
```

```
string ga_get_var_type(string var);

//Env var creating 1.
void ga_create_b(string var);
void ga_create_i(string var);
void ga_create_f(string var);
void ga_create_v(string var);
void ga_create_s(string var);

//Env var creating 2.
void ga_init_b(string var, bool value);
void ga_init_i(string var, int value);
void ga_init_f(string var, float value);
void ga_init_v(string var, Vector value);
void ga_init_s(string var, string value);

//Env var getting.
bool   ga_get_b(string var);
int    ga_get_i(string var);
float  ga_get_f(string var);
Vector ga_get_v(string var);
string ga_get_s(string var);

//Env var getting helpers.
bool ga_b_exists_and_true(string var);

//Env var setting.
void ga_set_b(string var, bool value);
void ga_set_i(string var, int value);
void ga_set_f(string var, float value);
void ga_set_v(string var, Vector value);
void ga_set_s(string var, string value);

//Additional env var setting.
void ga_toggle_b(string var);
void ga_set_i_by_delta(string var, int delta);
void ga_set_f_by_delta(string var, float delta);
void ga_set_v_by_delta(string var, Vector delta);

//-----------------------------------------------
//              Env Vars : System Vars
//-----------------------------------------------

//Testing which variables exist.
bool ga_exists_sys_for_get(string var);
bool ga_exists_sys_for_set(string var);
```

```
//System var getting.
bool   ga_get_sys_b(string var);
int    ga_get_sys_i(string var);
float  ga_get_sys_f(string var);
Vector ga_get_sys_v(string var);
string ga_get_sys_s(string var);

//System var setting.
void ga_set_sys_b(string var, bool value);
void ga_set_sys_i(string var, int value);
void ga_set_sys_f(string var, float value);
void ga_set_sys_v(string var, Vector value);
void ga_set_sys_s(string var, string value);

//-----------------------------------------------
//            Package State Vars
//-----------------------------------------------

//Functions for the current package.

bool ga_package_var_exists(string var);

void ga_package_init_b(string var, bool value);
void ga_package_init_i(string var, int value);
void ga_package_init_f(string var, float value);
void ga_package_init_v(string var, Vector value);
void ga_package_init_s(string var, string value);

bool   ga_package_get_b(string var);
int    ga_package_get_i(string var);
float  ga_package_get_f(string var);
Vector ga_package_get_v(string var);
string ga_package_get_s(string var);

void ga_package_set_b(string var, bool value);
void ga_package_set_i(string var, int value);
void ga_package_set_f(string var, float value);
void ga_package_set_v(string var, Vector value);
void ga_package_set_s(string var, string value);

void package_var_remove(string var);

//Functions for any package.

bool ga_package2_var_exists(string package, string var);
```

```
void ga_package2_init_b(string package, string var, bool value);
void ga_package2_init_i(string package, string var, int value);
void ga_package2_init_f(string package, string var, float value);
void ga_package2_init_v(string package, string var, Vector value);
void ga_package2_init_s(string package, string var, string value);

bool   ga_package2_get_b(string package, string var);
int    ga_package2_get_i(string package, string var);
float  ga_package2_get_f(string package, string var);
Vector ga_package2_get_v(string package, string var);
string ga_package2_get_s(string package, string var);

void ga_package2_set_b(string package, string var, bool value);
void ga_package2_set_i(string package, string var, int value);
void ga_package2_set_f(string package, string var, float value);
void ga_package2_set_v(string package, string var, Vector value);
void ga_package2_set_s(string package, string var, string value);

void ga_package2_var_remove(string package, string var);

//-----------------------------------------------
//               Dynamic Vars
//-----------------------------------------------

bool   ga_dyn_exists(string var);
string ga_dyn_get_var_type(string var);

void ga_dyn_create_b(string var);
void ga_dyn_create_i(string var);
void ga_dyn_create_f(string var);
void ga_dyn_create_v(string var);
void ga_dyn_create_s(string var);

void ga_dyn_init_b(string var, bool value);
void ga_dyn_init_i(string var, int value);
void ga_dyn_init_f(string var, float value);
void ga_dyn_init_v(string var, Vector value);
void ga_dyn_init_s(string var, string value);

bool   ga_dyn_get_b(string var);
int    ga_dyn_get_i(string var);
float  ga_dyn_get_f(string var);
Vector ga_dyn_get_v(string var);
string ga_dyn_get_s(string var);
```

```
bool   ga_dyn_get_b_maybe(string var, bool default_value);
int    ga_dyn_get_i_maybe(string var, int default_value);
float  ga_dyn_get_f_maybe(string var, float default_value);
Vector ga_dyn_get_v_maybe(string var, Vector default_value);
string ga_dyn_get_s_maybe(string var, string default_value);

bool ga_dyn_b_exists_and_true(string var);

void ga_dyn_set_b(string var, bool value);
void ga_dyn_set_i(string var, int value);
void ga_dyn_set_f(string var, float value);
void ga_dyn_set_v(string var, Vector value);
void ga_dyn_set_s(string var, string value);

void ga_dyn_toggle_b(string var);
void ga_dyn_set_i_by_delta(string var, int delta);
void ga_dyn_set_f_by_delta(string var, float delta);
void ga_dyn_set_v_by_delta(string var, Vector delta);

void ga_dyn_remove(string var);

void   ga_dyn_node_itr_start(string prefix);
string ga_dyn_node_itr_next();

void   ga_dyn_leaf_itr_start(string prefix);
string ga_dyn_leaf_itr_next();

void ga_dyn_dump();

//-----------------------------------------------
//                 Textures
//-----------------------------------------------

void ga_tex_keep_alive(string tex_name);
LIST ga_get_tex_names_with_prefix(string prefix);

//-----------------------------------------------
//                   Sounds
//-----------------------------------------------

string ga_sound_name_to_fn(string sound);
LIST ga_get_sound_names_with_prefix(string prefix);

void ga_play_sound(string sound);
void ga_play_sound_menu(string sound);
```

```
void ga_play_music(string sound);
void ga_stop_music();
string ga_get_current_music_fn();

void ga_play_playlist(string playlist);
bool ga_playlist_exists(string playlist);

//-----------------------------------------
//                Input Binds
//-----------------------------------------

void ga_enable_non_escape_binds(bool value);
void ga_enable_escape_bind(bool value);
string ga_what_binds_to_action(string action);

//-----------------------------------------------
//                  Meshes
//-----------------------------------------------

string ga_mesh_get_tex(string mesh_name);
float  ga_mesh_get_radius(string mesh_name);
float  ga_mesh_get_inv_radius(string mesh_name);

//-----------------------------------------------
//                 Game Related
//-----------------------------------------------

string ga_get_package_name();
LIST ga_get_current_packages();

bool ga_is_cheating_enabled();

bool ga_get_hardcore_mode();
void ga_set_hardcore_mode(bool value);

//Life and death (game stuff).
bool ga_kill_player();

//-----------------------------------------------
//              Use and Look Objects
//-----------------------------------------------

bool   ga_use_object_exists();
string ga_use_object_get_type();

bool          ga_look_object_bent_exists();
```

```
int           ga_look_object_bent_get_chunk_id();
LocalBlockPos ga_look_object_bent_get_lbp();

bool ga_look_object_ment_exists();
int  ga_look_object_ment_inst_id();

bool          ga_look_object_block_exists();
int           ga_look_object_block_get_chunk_id();
LocalBlockPos ga_look_object_block_get_lbp();
int           ga_look_object_block_get_normal_side();
Vector        ga_look_object_block_get_lp();

//------------------------------------------------
//               System HUD Related
//------------------------------------------------

void ga_hud_msg(string msg, float duration);

void ga_hud_reg_damage_from_dir(int damage, Vector dir);
void ga_hud_reg_damage_from_dir_color(int damage, Vector dir, Vector color);
void ga_hud_reg_dir_tex(string name, string tex, Vector dir, float duration);

//------------------------------------------------
//       Moving The Player Through Chunk Tree
//------------------------------------------------

void ga_shrink();
void ga_shrink2(Vector lp);
void ga_grow();
void ga_grow2(Vector lp);

void ga_tele(string path, Vector offset);
void ga_set_tele_back(string path, Vector offset);
void ga_tele_back();

void ga_tele_same_level(Vector lp);

//------------------------------------------------
//                 Exploration
//------------------------------------------------

int  ga_get_fertile_radius(int level_delta);
void ga_set_fertile_radius(int level_delta, int radius);

int  ga_get_level_radius(int level_delta);
void ga_set_level_radius(int level_delta, int radius);
```

```
void ga_explore_while(string func, string win);

//-----------------------------------------------
//                 Worldgen
//-----------------------------------------------

void ga_worldgen_main_dummy(string bt, string arg);
void ga_worldgen_refresh_state_var(string var);
void ga_worldgen_refresh_state_all();

//-----------------------------------------------
//               Windows (Part 1)
//-----------------------------------------------

//Window related.
void ga_window_push(string win_name);
void ga_window_pop();
void ga_window_pop_all();
void ga_hud_window_add(string win_name, int priority);
void ga_hud_window_remove(string win_name);

//These 3 functions are deprecated.
void ga_main_menu_push(string win_name);
void ga_main_menu_pop();
void ga_main_menu_pop_all(bool return_to_game);

LIST ga_get_window_stack_display_names();
LIST ga_main_menu_get_stack_display_names(); //Deprecated.
void ga_open_title_menu();
void ga_open_console();
void ga_close_console();
bool ga_get_program_has_focus();
void ga_set_game_paused_while_on_win_stack(bool value);

//-----------------------------------------------
//               Viewer Queries
//-----------------------------------------------

//Viewer queries.
int ga_get_viewer_chunk_id();
int ga_get_viewer_ancestor_chunk_id(int level);
string ga_get_viewer_chunk_bt();
int ga_get_viewer_level();
Vector ga_get_viewer_offset();
Vector ga_get_viewer_lp(int level);
```

```
BlockPos ga_get_viewer_bp(int level);
string ga_get_viewer_path();
string ga_get_viewer_path_ext();
Vector ga_get_vec_to_viewer(int level, Vector lp);
float ga_lbp_dist_to_viewer(int chunk_id, int lbp_hash);
float ga_block_dist_to_viewer(int level, BlockPos bp);

//Cached ment variables.
Vector ga_ment_get_var_special_vec_to_viewer(int inst_id);
float ga_ment_get_var_special_dist_to_viewer(int inst_id);

//-----------------------------------------------
//               Basic Entities
//-----------------------------------------------

string ga_bent_get_type(int level, BlockPos bp);

void ga_bent_add(int level, BlockPos bp, string type, float rl);
void ga_bent_add_i(int level, BlockPos bp, string type, int param, float rl);
void ga_bent_add_s(int level, BlockPos bp, string type, string param, float rl);

void ga_bent_set_param_i(int level, BlockPos bp, int value, float rl);
void ga_bent_set_param_s(int level, BlockPos bp, string value, float rl);
int    ga_bent_get_param_i(int level, BlockPos bp);
string ga_bent_get_param_s(int level, BlockPos bp);

void ga_bent_remove_temp(int level, BlockPos bp, int num_sec);
void ga_bent_remove_perm(int level, BlockPos bp);

LIST ga_bent_sphere_query(int level, Vector lp, float radius);
CLASS ga_search_for_bent_in_chunk(int chunk_id, string bent_type);

LIST ga_get_bent_names_with_prefix(string prefix);

//-----------------------------------------------
//             Moving Entities (type)
//-----------------------------------------------

bool ga_ment_type_var_exists(string type, string var);

bool   ga_ment_get_static_b(string type, string var);
int    ga_ment_get_static_i(string type, string var);
float  ga_ment_get_static_f(string type, string var);
Vector ga_ment_get_static_v(string type, string var);
string ga_ment_get_static_s(string type, string var);
```

```
bool ga_ment_static_b_exists_and_true(string type, string var);

//-----------------------------------------------
//           Moving Entities (instance)
//-----------------------------------------------

void ga_ment_start(int level, Vector lp, string type);
void ga_ment_end();
void ga_ment_init_set_b(string key, bool value);
void ga_ment_init_set_i(string key, int value);
void ga_ment_init_set_f(string key, float value);
void ga_ment_init_set_v(string key, Vector value);
void ga_ment_init_set_s(string key, string value);

bool   ga_ment_var_exists(int inst_id, string var);
string ga_ment_get_var_type(int inst_id, string var);

bool   ga_ment_b_exists_and_true(int inst_id, string var);

bool   ga_ment_get_b(int inst_id, string var);
int    ga_ment_get_i(int inst_id, string var);
float  ga_ment_get_f(int inst_id, string var);
Vector ga_ment_get_v(int inst_id, string var);
string ga_ment_get_s(int inst_id, string var);

void ga_ment_set_var_rl_only(int inst_id, string var, float rl);

void ga_ment_set_b(int inst_id, string var, bool value);
void ga_ment_set_i(int inst_id, string var, int value);
void ga_ment_set_f(int inst_id, string var, float value);
void ga_ment_set_v(int inst_id, string var, Vector value);
void ga_ment_set_s(int inst_id, string var, string value);

void ga_ment_toggle_b(int inst_id, string var);
void ga_ment_set_i_by_delta(int inst_id, string var, int delta);
void ga_ment_set_f_by_delta(int inst_id, string var, float delta);
void ga_ment_set_v_by_delta(int inst_id, string var, Vector delta);

int ga_ment_inst_id_to_code_id(int inst_id);
int ga_ment_code_id_to_inst_id(int code_id);

bool ga_ment_exists(int inst_id);

ga_ment_respawn(int inst_id);
ga_ment_remove(int inst_id);
ga_ment_remove_perm(int inst_id);
```

```
ga_ment_remove_with_respawn_length(int inst_id, float respawn_length);

ga_ment_respawn_now(int inst_id);
ga_ment_remove_now(int inst_id);
ga_ment_remove_perm_now(int inst_id);
ga_ment_remove_with_respawn_length_now(int inst_id, float respawn_length);

string ga_ment_get_type(int inst_id);
Vector ga_ment_get_lp(int inst_id);
Vector ga_ment_get_sllp(int inst_id);
int    ga_ment_get_start_level(int inst_id);
int    ga_ment_get_level(int inst_id);
int    ga_ment_get_chunk_id(int inst_id);
float  ga_ment_get_radius(int inst_id);
void ga_ment_dump(int inst_id);

void ga_ment_tele(int inst_id, int chunk_id, Vector offset);

LIST ga_ment_sphere_query(
    int level, int min_level, int max_level,
    Vector lp, float radius);

void ga_ment_set_alarm(
    int inst_id, float alarm_game_time, string alarm_name);
void ga_ment_set_alarm_on_level(
    int inst_id, int level, float alarm_level_time, string alarm_name);

void ga_ment_all_dump();

void ga_ment_vec_to_another_ment(int inst_id_1, int inst_id_2);

LIST ga_get_ment_names_with_prefix(string prefix);

//-----------------------------------------------
//                  Particles
//-----------------------------------------------

void ga_particle_add(CLASS args);
void ga_particle_explosion(CLASS args);
void ga_particle_trail(CLASS args);
void ga_particle_ring(CLASS args);

//-----------------------------------------------
//                 Blocks (type)
//-----------------------------------------------
```

```
bool ga_bt_exists(string bt);
bool ga_bt_var_exists(string bt, string var);
bool ga_bt_get_physically_solid(string bt);

LIST ga_get_block_names_with_prefix(string prefix);

//-----------------------------------------------
//             Blocks (instance)
//-----------------------------------------------

//Old block type getting functions.
string ga_block_get(int level, BlockPos bp);
string ga_get_cocoon_block_of_chunk(int level, BlockPos vcp);

//New block type getting functions.
string ga_bp_to_bt(int level, BlockPos bp);
string ga_chunk_id_and_lbp_to_bt(int chunk_id, LocalBlockPos lbp);
string ga_chunk_id_to_bt(int chunk_id);
string ga_vcp_to_bt(int level, ViewerCentricPos vcp);

void ga_block_change_rl(
    int level, BlockPos bp, string new_bt, float rl);
void ga_block_change_rl_default(
    int level, BlockPos bp, string new_bt);
void ga_block_change_perm(
    int level, BlockPos bp, string new_bt);

bool   ga_block_var_exists(int level, BlockPos bp, string var);
string ga_block_get_var_type(int level, BlockPos bp, string var);

bool   ga_block_get_b(int level, BlockPos bp, string var);
int    ga_block_get_i(int level, BlockPos bp, string var);
float  ga_block_get_f(int level, BlockPos bp, string var);
Vector ga_block_get_v(int level, BlockPos bp, string var);
string ga_block_get_s(int level, BlockPos bp, string var);

bool ga_block_b_exists_and_true(int level, BlockPos bp, string var);

void ga_block_set_b(int level, BlockPos bp, string var, bool value);
void ga_block_set_i(int level, BlockPos bp, string var, int value);
void ga_block_set_f(int level, BlockPos bp, string var, float value);
void ga_block_set_v(int level, BlockPos bp, string var, Vector value);
void ga_block_set_s(int level, BlockPos bp, string var, string value);

void ga_block_toggle_b(int level, BlockPos bp, string var);
void ga_block_set_i_by_delta(int level, BlockPos bp, string var, int delta);
```

```
void ga_block_set_f_by_delta(int level, BlockPos bp, string var, float delta);
void ga_block_set_v_by_delta(int level, BlockPos bp, string var, Vector delta);

string ga_get_most_common_bt_in_chunk(int chunk_id);

CLASS ga_search_for_bt_in_chunk(int chunk_id, string bt);
CLASS ga_search_for_bt_in_chunk_random(int chunk_id, string bt);
void ga_search_and_replace_bt_in_chunk_perm(int chunk_id string bt1, string bt2);

//Alternate API for block vars.
bool   ga_chunk_var_exists(int level, ViewerCentricPos vcp, string var);
bool   ga_chunk_get_b(int level, ViewerCentricPos vcp, string var);
int    ga_chunk_get_i(int level, ViewerCentricPos vcp, string var);
float  ga_chunk_get_f(int level, ViewerCentricPos vcp, string var);
Vector ga_chunk_get_v(int level, ViewerCentricPos vcp, string var);
string ga_chunk_get_s(int level, ViewerCentricPos vcp, string var);
void   ga_chunk_set_b(int level, ViewerCentricPos vcp, string var, bool value);
void   ga_chunk_set_i(int level, ViewerCentricPos vcp, string var, int value);
void   ga_chunk_set_f(int level, ViewerCentricPos vcp, string var, float value);
void   ga_chunk_set_v(int level, ViewerCentricPos vcp, string var, Vector value);
void   ga_chunk_set_s(int level, ViewerCentricPos vcp, string var, string value);

//-----------------------------------------------
//          Respawn Point and Waypoints
//-----------------------------------------------

string ga_get_respawn_path();
BlockPos ga_get_respawn_lbp();
void ga_set_respawn_point(string path, BlockPos lbp);
void ga_set_respawn_cb(string func, string win);

//The following are deprecated.
//Use Data/Packages/base/Game/game_base_wp_system.lua instead.
string ga_get_emergency_waypoint_path();
void   ga_add_waypoint_sloppy(string path, string name_override);
void   ga_add_waypoint_sloppy_in_only(string path, string name_override);

//-----------------------------------------------
//        Coordinates: Blocks and Chunks
//-----------------------------------------------

int              ga_chunk_id_to_level(int chunk_id);
ViewerCentricPos ga_chunk_id_to_vcp(int chunk_id);
BlockPos         ga_chunk_id_to_bp(int chunk_id);
string           ga_chunk_id_to_path(int chunk_id);
```

```
int ga_vcp_to_chunk_id(int level, BlockPos vcp);
int ga_path_to_chunk_id(string path);

BlockPos ga_chunk_id_and_lbp_to_bp(int chunk_id, BlockPos lbp);
BlockPos ga_lbp_to_bp(BlockPos vcp, LocalBlockPos lbp);

BlockPos         ga_vcp_to_bp(int level, ViewerCentricPos vcp);
ViewerCentricPos ga_bp_to_vcp(int level, BlockPos bp);

int ga_bp_to_chunk_id(int level, BlockPos bp);

int              ga_chunk_id_to_parent_chunk_id(int chunk_id);
BlockPos         ga_bp_to_parent_bp(int level, BlockPos bp);
ViewerCentricPos ga_bp_to_parent_vcp(BlockPos bp);
int              ga_bp_to_parent_chunk_id(int level, BlockPos bp);

BlockPos ga_bp_to_ancestor_bp(int source_level, BlockPos source_bp, int target_level);
int      ga_chunk_id_to_ancestor_chunk_id(int source_chunk_id, int target_level);

string        ga_bp_to_path(int level, BlockPos bp);
LocalBlockPos ga_bp_to_lbp(BlockPos bp);

//See base/Game/std.lua for
//lbp_to_bp, bp_to_vcp, bp_to_lbp, local_to_level_pos,
//level_to_local_pos, lp_to_vcp, lp_to_offset, bp, block_center,
//lbph_to_lbp, lbp_to_lbph, lp_to_bp, etc.

//-----------------------------------------------
//             Coordinates: Vectors
//-----------------------------------------------

Vector ga_chunk_id_and_offset_to_lp(int chunk_id, Vector offset);
Vector ga_offset_to_lp(BlockPos vcp, Vector offset);

CLASS ga_level_scale_factor(int source_level, int target_level);
CLASS ga_convert_lp(
    int source_level, int target_level, Vector source_lp);

CLASS ga_finest_chunk_containing_point(int level, Vector lp);

//-----------------------------------------------
//                   Math
//-----------------------------------------------

//Much of this is in "base/Game/std.lua".
```

```
CLASS ga_path_diff(string path1, string path2);

//-----------------------------------------------
//             Movement and Physics
//-----------------------------------------------

void ga_camera_set_look(Vector look, Vector up);
void ga_move_set_desired_travel(Vector travel);
void ga_move_set_roll(float roll);
bool ga_move_get_on_sure_footing();
void ga_move_set_ledge_guards(bool on);

void ga_move_set_body_spirit();
void ga_move_set_body_spirit_off();
bool ga_move_set_body_ground(
    Vector trans, float radius, float bot_to_eye, float eye_to_top);
bool ga_move_set_body_fly(
    Vector trans, float radius, bool use_true_up);

void ga_player_model_set_look();
void ga_player_model_q2md2_set_cmd(string cmd);
void ga_player_model_q2md2_set_state(string state);

//-----------------------------------------------
//                 Visibility
//-----------------------------------------------

bool ga_vis_test_level(int level, Vector lp_start, Vector lp_end);

CLASS ga_ray_cast(
    int level, Vector lp_start, Vector dir,
    int min_level, int max_level);

//-------------------------------------------
//                Rendering
//-------------------------------------------

void ga_render_skip_next_frame();
void ga_render_push_matrix();
void ga_render_pop_matrix();
void ga_render_matrix_load_identity();
void ga_render_matrix_row_major(
    float m11, float m12, float m13, float m14,
    float m21, float m22, float m23, float m24,
    float m31, float m32, float m33, float m34,
    float m41, float m42, float m43, float m44);
```

```
void ga_render_matrix_translated(float trans_x, float trans_y, float trans_z);
void ga_render_matrix_scaled(float scale_x, float scale_y, float scale_z);
void ga_render_matrix_rotated(
    float angle,
    Vector axis);
void ga_render_matrix_frame(
    Vector look,
    Vector up,
    Vector left);
void ga_render_matrix_frame_from_ment(int inst_id);

void ga_render_ment_typical(int inst_id);
void ga_render_mesh(string mesh_name);
void ga_render_mesh_with_tex(
    string mesh_name,
    string tex_name);
void ga_render_mesh_with_tex_alpha(
    string mesh_name,
    string tex_name,
    float alpha);
void ga_render_mesh_with_tex_no_lighting(
    string mesh_name,
    string tex_name);
void ga_render_mesh_with_tex_alpha_no_lighting(
    string mesh_name,
    string tex_name,
    float alpha);

void ga_render_line(
    Vector v1,
    Vector v2);
void ga_render_line_thick(
    Vector v1,
    Vector v2,
    float thickness);
void ga_render_triangle(
    Vector v1,
    Vector v2,
    Vector v3,
    float u1, float v1,
    float u2, float v2,
    float u3, float v3,
    string tex);
Vector ga_render_get_color();
void ga_render_color(
    Vector color);
```

```
void ga_render_set_depth_test_enabled(bool value);
void ga_render_clear_depth_buffer();

//-----------------------------------------------
//              Windows (Part 2)
//-----------------------------------------------

//These are described in another chapter.
//These are in addition to the previous "windows" functions.

//-----------------------------------------------
//             Rebooting the Game
//-----------------------------------------------

string ga_reboot_dyn_itr_get();
void ga_reboot_dyn_itr_next();
bool ga_reboot_dyn_itr_at_end();
void ga_reboot_dyn_itr_save();

//-----------------------------------------------
//                  File IO
//-----------------------------------------------

int    ga_open_file_for_writing(string file_name);
int    ga_open_named_pipe(string name);
void   ga_write(int handle, string str);
string ga_read(int handle);
void   ga_close_file(int handle);

//-----------------------------------------------
//               Accessibility
//-----------------------------------------------

bool ga_get_is_colorblind();
Vector ga_get_colorblind_closest(Vector color);
Vector ga_get_colorblind_bynum(int num);
void ga_set_colorblind_bynum(int num, Vector color);

//-----------------------------------------------
//               Text and Strings
//-----------------------------------------------

Vector ga_color_code_to_vec(string code);
string ga_color_vec_to_code(Vector color);
string ga_txt_strip_esc_seq(string input);
```

```
//-----------------------------------------------
//               Windows Clipboard
//-----------------------------------------------

void ga_copy_to_clipboard(string str);
string ga_paste_from_clipboard();
```

We will now describe all of these functions.

17.3 Game API: Program Level Functions

```
string ga_command(string command);

void ga_save(bool play_sound);
void ga_load();
void ga_exit();
void ga_exit_with_error();

void ga_print(string line);
void ga_flush();

void ga_console_print(string line);

void ga_dump_lua_env();
```

The engine has a console in which the user can enter commands. The function ga_command causes the specified command to be executed. This should not be used for typical game play functions. It returns the result of the command converted to a string. For example, the following is how you can get what is bound to the W key:

```
local str = ga_command("bind W.downup get")
```

The function `ga_save` saves the game. You specify whether the "saving game sound" is played.

The function `ga_load` loads the game. That is, each player has exactly one saved game slot. The load function will load that saved game.

The function `ga_exit` exists the program (without saving).

The function `ga_exit_with_error` is just like `ga_exit`, except it prints that there was an error and dumps the "debug stack." Note that this command exits the program *without saving*.

Note that some of these functions work by making a request which is fulfilled later.

The function `ga_print` prints the given string to the stdout.txt file as a line. No newline character is required. The `ga_print` function works by writing

to a buffer. The function `ga_flush` flushes this buffer. Note that exiting the program via `ga_exit` will flush the buffer also.

The user can open the console by pressing the tilde key. This console displays a list of lines, including commands that the user entered into the console as well as output from commands. The function `ga_console_print` adds the given string to the console as a line (no newline character is required).

17.3.1 Pushing and popping the debug stack

```
void ga_debug_push(string frame);
void ga_debug_pop(string frame);
```

The program uses its own internal debug stack. This stack is printed to the file stdout.txt if the program exits due to an error. You can push and pop to the stack via these functions.

The string at the top of the stack must be the string that you are passing to the pop function, otherwise the program will exit.

You can use these functions to figure out where the program is crashing if it is crashing while executing your Lua code. Here is an example:

```
function p.top_function()
    ga_debug_push("top")
    --Do some stuff.
    p.bad_function()
    --Do more stuff.
    ga_debug_pop("top")
end

function p.bad_function()
    ga_debug_push("i_am_worried_about_this")
    local gold = ga_get_i("goldddddddd")
    ga_debug_pop("i_am_worried_about_this")
end
```

When the program runs, it will crash because there is no variable called `goldddddddd`. When you open the `stdout.txt` file, you will see that the top of the debug stack is `i_am_worried_about_this`.

```
void ga_debug_line(string line);
```

You can also call the function `ga_debug_line` to print information between the push and pop, assuming the program crashes. Here is an example:

```
function p.function 1()
    ga_debug_push("function_1")
    p.good_function()
    ga_debug_line("checkpoint 1A")
```

```
    p.happy_function()
    ga_debug_line("checkpoint 1B")
    p.function_2()
    ga_debug_line("checkpoint 1C")
    p.funny_function()
    ga_debug_pop("function_1")
end

function p.function_2()
    ga_debug_push("function_2")
    p.function_that_will_crash()
    ga_debug_pop("function_2")
end
```

When we call `function_1`, the program will ultimately crash. In the stdout.txt file, it will show the debug stack. The last frame on the stack will be `function_2`. The previous frame will be `function_1`, and it will show that the program got to checkpoint 1A and 1B, but not 1C.

```
void ga_load_user_script(string mod_name);
```

Note that all Lua scripts in the current package's directory are "loading into the Game Lua state". You can also load individual scripts from the directory Input/Scripts. For example, if you want to load `foo.lua` into the Game Lua state, then make the call `ga_load_user_script("foo")`.

17.4 Game API: Returning Values From a Function

```
void ga_return_b(string var, bool value);
```

Normally when the C++ part of the program calls a Lua function, the Lua function can return some value that the C++ part of the program can process. However sometimes several values are needed to be returned.

The function `ga_return_b` serves to return an extra bool value. This function specifies the name of the variable and its value. This should be called before the actual return statement of the Lua function.

This can be used, for example, by the `on_ment_hit` function of a moving entity script.

17.5 Game API: Time

```
float ga_get_game_time();
float ga_get_level_time(int level);
int   ga_get_sys_time();
```

There are two types of time in the game: game time and level time. The game time starts at 0.0 when the player starts a new game. The game time records the number of seconds that have passed since the start of the game. However there is an exception: there are places in the game where the player can "sleep". This will advance the game time. In this way the player can easily sleep for an hour causing many entities to respawn.

The other kind of time is "level time". Each level (level 0, level 1, etc) has its own time system. On the level containing the player, the level time advances at the normal rate. However levels that are coarser than the player have their time advanced at a slower rate. For example, if the player is on level L, then time on level L-1 advances at 1/16 the speed as normal. The time on level L-2 advances at (1/256) the speed as normal.

If the player is on level L but moves to level L-1, then level L is destroyed. Then if the player shrinks from level L-1 back into level L, then the time of level L will reset at "0.0".

The function `ga_get_sys_time` returns the "operating system time". Under the hood this returns `time(NULL)` (where `time` is the function defined in the C++ programming language). So, `get_sys_time` returns the number of seconds elapsed since the Unix epoch (midnight January 1, 1970).

```
bool ga_get_game_paused();
```

The function `ga_get_game_paused` returns whether or not the game is paused. For example, when the player opens the console, the game is paused.

For debugging you can use the Windows high precision timer:

```
int ga_get_high_precision_timer();
```

That function returns the result the Windows API function QueryPerformanceCounter.

17.6 Game API: Pseudo Random Functions

17.6.1 Core random functions

```
void ga_srand(int seed);
int ga_rand();
float ga_randf();
float ga_randf_range(float min_f, float max_f);
int ga_randi(int min_i, int max_i);
```

Note:

```
FBW_RAND_MAX = 32767
```

The function `ga_srand` sets the pseudo random number generator seed.

The function `ga_rand` pseudo randomly generates an integer (using the pseudo random seed) where the integer is between 0 and `FBW_RAND_MAX-1` inclusive. Each subsequent call to `ga_rand` will generate a new number. The caller should try to avoid using this function and should instead use `ga_randf`, `ga_randf_range`, and `ga_randi`.

The function `ga_randf` pseudo randomly generates a floating point number in the range [0.0, 1.0].

The function `ga_randf_range` generates a pseudo random float in the given range. The function `ga_randi` generates a pseudo random integer in the given range (and the range is inclusive, so that largest integer that can be generated is max_i).

17.6.2 Seeds associated to chunks

```
int ga_chunk_id_to_seed(int chunk_id);
```

This function gets a seed (for pseudo random number generation) that is determined by the path of the given chunk. Since this uses a chunk id, the chunk must be in the active chunk tree.

```
int ga_path_to_seed(string path);
```

This gets the seed for the specified chunk which may or may not be in the active chunk tree. If it is in the active chunk tree, the seed agrees with what is returned by `ga_chunk_id_to_seed`. However, the function `ga_chunk_id_to_seed` is much faster than `ga_path_to_seed` because it uses a cached seed.

```
int ga_chunk_id_and_lbp_to_seed(int chunk_id, LocalBlockPos lbp);
```

This function gets a seed associated to a block. This might not be the same as the seed associated to the chunk that is the block. This function requires the chunk to be in the active chunk tree (the one named by `chunk_id`).

```
int ga_bp_to_seed(int level, BlockPos bp);
```

This function also gets a seed associated to a block. The seed is the same value returned by `ga_chunk_id_and_lbp_to_seed`. Again, the chunk containing the block needs to be in the active chunk tree.

```
int ga_chunk_seed(int chunk_id);                  //Deprecated.
int ga_lbp_seed_pos(int chunk_id, int lbp_hash); //Deprecated.
int ga_block_seed_pos(int level, BlockPos bp);   //Deprecated.
```

Do not use these functions because they are deprecated.

17.7 Game API: Env Vars: Globals

The engine has a "variable store" (an "environment"). This holds variables with one of several types: bool, int, float, Vector, and string. A vector is a class with three float members: x, y, and z. A package's Lua scripts are only able to access two types of environment variables: 1) "global variables", and 2) a select few other variables which we call "system variables". The function ga_get_i gets the value of a global environment variable whose type is an integer, and ga_get_sys_i gets the value of a system environment variable whose type is an integer. Global variables must be declared in the file globals.txt in the package's top directory. You can read about this in Section 19.3.

17.7.1 Getting env globals

```
bool ga_exists(string var);
```

Use this `ga_exists` functions to determine if the given *global* variable exists.

```
string ga_get_var_type(string var);
```

The function `ga_get_var_type` returns one of the following strings depending on the type of the var: `"b"`, `"i"`, `"f"`, `"v"`, `"s"`.

Note that if there is an (integer) global variable called `num_rockets`, then the call to `ga_exists("num_rockets")` will return true. Note that the variable `num_rockets` actually corresponds to the environment variable with the full name

`game.globals.num_rockets`.

If you open up the game's console and type the command

```
ls game.globals.num_rockets
```

it will tell you the value of the variable. However to get the value of this variable from a Lua script, you would use the command

```
ga_get_i("num_rockets")
```

as we will describe later.

We recommend prefixing all your global variables with a short string indicating the name of your package. For example if your package is called "Tree-Cutter", then it would be better to have the global variable `tc.num_rockets` instead of just `num_rockets`.

```
void ga_create_b(string var);
void ga_create_i(string var);
void ga_create_f(string var);
void ga_create_v(string var);
void ga_create_s(string var);
```

You can use these to create global variables. If the global variable already exists, then nothing will happen. If the variable does NOT exist (in particular, it is not listed in globals.txt), then it will be created (and initialized to a default value) for use by the game. However it will not be saved when the game is saved. Thus, these `ga_create_XXX` functions should be used for the creation of *temporary variables* only.

```
void ga_init_b(string var, bool value);
void ga_init_i(string var, int value);
void ga_init_f(string var, float value);
void ga_init_v(string var, Vector value);
void ga_init_s(string var, string value);
```

These are very similar to the `ga_create_XXX` functions. Both the `ga_create_XXX` and `ga_init_XXX` functions will do nothing if the (global) variable already exists. However if the variable does NOT exist, then the `ga_init_XXX` function will set the varaible to the given value (as opposed to the `ga_create_XXX` function which sets it to a default value).

```
bool   ga_get_b(string var);
int    ga_get_i(string var);
float  ga_get_f(string var);
Vector ga_get_v(string var);
string ga_get_s(string var);
```

Use these `ga_get_XXX` functions to get the value of a global environment variable. If the variable does not exist, the program will exit. If the type of the variable is wrong, then the program will exit.

```
bool ga_b_exists_and_true(string var);
```

The `ga_b_exists_and_true` function returns true if and only if the (bool) variable exists AND is true. If the variable exists but is not a bool, the program will exit.

17.7.2 Setting env globals

```
void ga_set_b(string var, bool value);
void ga_set_i(string var, int value);
void ga_set_f(string var, float value);
void ga_set_v(string var, Vector value);
void ga_set_s(string var, string value);
```

Use these ga_set_XXX functions to set the value of a global environment variable. If the variable does not exist, the program will exit. If the type of the variable is wrong, then the program will exit.

```
void ga_toggle_b(string var);
void ga_set_i_by_delta(string var, int delta);
void ga_set_f_by_delta(string var, float delta);
void ga_set_v_by_delta(string var, Vector delta);
```

The function `ga_toggle_b` will toggle the value of the specified bool. The "set by delta" functions will replace the value of the variable with its variable plus delta. If the variable does not exist, the program will exit.

17.8 Game API: Env Vars: System Vars

```
bool ga_exists_sys_for_get(string var);
bool ga_exists_sys_for_set(string var);

bool   ga_get_sys_b(string var);
int    ga_get_sys_i(string var);
float  ga_get_sys_f(string var);
Vector ga_get_sys_v(string var);
string ga_get_sys_s(string var);

void ga_set_sys_b(string var, bool value);
void ga_set_sys_i(string var, int value);
void ga_set_sys_f(string var, float value);
void ga_set_sys_v(string var, Vector value);
void ga_set_sys_s(string var, string value);
```

Use these functions `ga_get_sys_XXX` and `ga_set_sys_XXX` functions for getting and setting "system" environment variables.

Some system variables are read only. Others you can read and write to. You can use the function `ga_exists_sys_for_get` to see if you can read from a system variable, and you can use `ga_exists_sys_for_set` to see if you can write to it.

To get a list of all system variables you can access (and whether or not they are read only), open up the console in the game and run the command `gendoc`. This will generate the file

`Output/Documentation/system_vars.txt`

For example, one line in this file is

```
game.player.camera.look                          (READ ONLY)
```

in the section listing vector variables. So you can write the following Lua code:

```
local look_vector = ga_get_sys_v("game.player.camera.look")
```

The reason why not all environment variables are exposed as system variables is because we do not want packages to depend on parts of the engine that are volatile. We want to avoid packages becoming broken if variable names change.

17.9 Game API: Package State Vars

Recall that the program has variables in what is called the "environment". Some of these environment variables can be accessed as "system variables". Others of these environment are called "global vars", which can be accessed by the current package. For example, if the package has a global variable called game.total, then this is saved as

```
game.globals.gold.total
```

in the environment.

When the player saves their game, all their global vars will be saved in the save folder for the player.

On the other hand, **package state variables** are variables not associated to a player but instead are associated to the package itself. So if the user has two players that are both for the xar package, if one of the player sets a package state variable, the other player will see that new value.

You would want to use package variables for "user preferences", not "player attributes". For example, if you allow the player to choose whether or not they move with 6 degrees of freedom, this would be a good candiate for a package variable. For another example, the format for the HUD would be a good package variable.

You can only interact with a subset of package variables. In the following functions, var is appended to a prefix to get a longer varible name. For example, if var = "gold.total", then the full var would be

```
package.state.globals.gold.total
```

in the environment. Here are the functions for package variables for the current package:

```
bool ga_package_var_exists(string var);

void ga_package_init_b(string var, bool value);
void ga_package_init_i(string var, int value);
void ga_package_init_f(string var, float value);
void ga_package_init_v(string var, Vector value);
void ga_package_init_s(string var, string value);

bool   ga_package_get_b(string var);
int    ga_package_get_i(string var);
float  ga_package_get_f(string var);
Vector ga_package_get_v(string var);
string ga_package_get_s(string var);

void ga_package_set_b(string var, bool value);
void ga_package_set_i(string var, int value);
void ga_package_set_f(string var, float value);
```

```
void ga_package_set_v(string var, Vector value);
void ga_package_set_s(string var, string value);

void package_var_remove(string var);
```

The behavior of these functions is analogous to that of the env var functions in Section 17.8.

It is also possible to change package state varibles not just for the current package, but for *any* package. You can do this via the following functions:

```
bool ga_package2_var_exists(string package, string var);

void ga_package2_init_b(string package, string var, bool value);
void ga_package2_init_i(string package, string var, int value);
void ga_package2_init_f(string package, string var, float value);
void ga_package2_init_v(string package, string var, Vector value);
void ga_package2_init_s(string package, string var, string value);

bool   ga_package2_get_b(string package, string var);
int    ga_package2_get_i(string package, string var);
float  ga_package2_get_f(string package, string var);
Vector ga_package2_get_v(string package, string var);
string ga_package2_get_s(string package, string var);

void ga_package2_set_b(string package, string var, bool value);
void ga_package2_set_i(string package, string var, int value);
void ga_package2_set_f(string package, string var, float value);
void ga_package2_set_v(string package, string var, Vector value);
void ga_package2_set_s(string package, string var, string value);

void ga_package2_var_remove(string package, string var);
```

17.10 Game API: Dynamic Vars

Dynamic variables are similar to global variables, but there are some key differences. Dynamic variables can be created and destroyed at any time, and they do not need to be listed ahead of time in any file. They are stored in a file called dyn_vars.txt (in the player's save folder). Note that dynamic variables are not automatically saved when the player *reboots* the game. They must be saved manually (see the functions ga_reboot_dyn_XXX).

all dynamic variable names must start with "dyn."

17.10.1 Testing if a dynamic variable exists

```
bool ga_dyn_exists(string var);
```

Use the function above to determine if a dynamic variable exists.

17.10.2 Getting the type of a dynamic variable

```
string ga_dyn_get_var_type(string var);
```

The function `ga_dyn_get_var_type` returns one of the following strings depending on the type of the var: `"b"`, `"i"`, `"f"`, `"v"`, `"s"`.

17.10.3 Creating dynamic variables

```
void ga_dyn_create_b(string var);
void ga_dyn_create_i(string var);
void ga_dyn_create_f(string var);
void ga_dyn_create_v(string var);
void ga_dyn_create_s(string var);
```

These functions will create the variable if it does not exist. It will also set the var to an initial value, regardless of whether or not the var already exists.

```
void ga_dyn_init_b(string var, bool value);
void ga_dyn_init_i(string var, int value);
void ga_dyn_init_f(string var, float value);
void ga_dyn_init_v(string var, Vector value);
void ga_dyn_init_s(string var, string value);
```

These functions do nothing if the variable already exists. If the variable does not exist, it is set to the specified value.

17.10.4 Getting dynamic variables

```
bool   ga_dyn_get_b(string var);
int    ga_dyn_get_i(string var);
float  ga_dyn_get_f(string var);
Vector ga_dyn_get_v(string var);
string ga_dyn_get_s(string var);

bool ga_dyn_b_exists_and_true(string var);
```

These functions get the value of a dynamic variable. The function

`ga_dyn_b_exists_and_true`

returns true if and only if the variable exists AND is true. If the variable exists but is not a bool, the program will exit.

```
bool   ga_dyn_get_b_maybe(string var, bool default_value);
int    ga_dyn_get_i_maybe(string var, int default_value);
float  ga_dyn_get_f_maybe(string var, float default_value);
Vector ga_dyn_get_v_maybe(string var, Vector default_value);
string ga_dyn_get_s_maybe(string var, string default_value);
```

The "get maybe" functions will return the value of the variable if it exists, and otherwise they will return the default value that is passed to the function.

17.10.5 Setting dynamic variables

```
void ga_dyn_set_b(string var, bool value);
void ga_dyn_set_i(string var, int value);
void ga_dyn_set_f(string var, float value);
void ga_dyn_set_v(string var, Vector value);
void ga_dyn_set_s(string var, string value);
```

The functions above set the value of a dynamic variable. The variable must already exist.

```
void ga_dyn_toggle_b(string var);
void ga_dyn_set_i_by_delta(string var, int delta);
void ga_dyn_set_f_by_delta(string var, float delta);
void ga_dyn_set_v_by_delta(string var, Vector delta);
```

The functions above are analogous to the functions for global variables (see `ga_toggle_b` and `ga_set_i_by_delta`).

17.10.6 Removing dynamic variables

```
void ga_dyn_remove(string var);
```

This function removes a dynamic variable.

17.10.7 Iterating over dynamic variables

```
void   ga_dyn_node_itr_start(string prefix);
string ga_dyn_node_itr_next();

void   ga_dyn_leaf_itr_start(string prefix);
string ga_dyn_leaf_itr_next();
```

These variables are for iterating over leaf nodes and non-leaf nodes of dynamic variables. It is best if we show an example:

```
ga_dyn_init_s("dyn.a.b.c",      "apple")
ga_dyn_init_s("dyn.a.b.c.d",    "orange")
ga_dyn_init_s("dyn.a.b.c.e",    "peach")
```

```
ga_dyn_init_s("dyn.a.b.f",      "pear")
ga_dyn_init_s("dyn.a.b.g.h",    "lemon")
--
ga_console_print("Internal nodes of dyn.a.b:")
ga_dyn_node_itr_start("dyn.a.b")
while(true) do
    local name = ga_dyn_node_itr_next()
    if name == "" then break end
    ga_console_print(name)
end
--
ga_console_print("Leaf nodes of dyn.a.b:")
ga_dyn_leaf_itr_start("dyn.a.b")
while(true) do
    local name = ga_dyn_leaf_itr_next()
    if name == "" then break end
    ga_console_print(name)
end
```

When the code above is run, it will print the following:

```
Internal nodes of dyn.a.b:
g
c
Leaf nodes of dyn.a.b:
f
c
```

This, dyn.a.b leads to the *internal* nodes g and c, and dyn.a.b leads to the *leaf* nodes f and c. So c is both an internal and a leaf node of "dyn.a.b". The result need not be in alphabetical order.

17.10.8 Dumping dynamic variables

Finally,

```
ga_dyn_dump()
```

is used for debugging the dynamic variable system.

17.11 Game API: Textures

```
void ga_tex_keep_alive(string tex_name);
LIST ga_get_tex_names_with_prefix(string prefix);
```

If a texture is not used for a long time (perhaps a minute, but this can be configured by the user), the texture is unloaded. This keep alive function will

(make a request to) load the texture if it has not yet been loaded. If it has been loaded, it will simulate that the texture has just been used. So by calling this function periodically, you can insure that the texture does not get unloaded.

If the game attempts to use a texture that has not yet been loaded, it may appear black for a few frames.

Use the function `ga_get_tex_names_with_prefix` to get an array consisting of all names of textures that start with the given prefix. Technically it returns an array of tables, where each table has a member called "name" (the string name of the texture).

17.12 Game API: Sounds

```
string ga_sound_name_to_fn(string sound);
LIST ga_get_sound_names_with_prefix(string prefix);

void ga_play_sound(string sound);
void ga_play_sound_with_volume(string sound, float volume);
void ga_play_sound_menu(string sound);
void ga_play_sound_menu_with_volume(string sound, float volume);

void ga_play_music(string sound);
void ga_stop_music();
string ga_get_current_music_fn();

void ga_play_playlist(string playlist);
bool ga_playlist_exists(string playlist);
```

There are tree types of sounds for the program: 1) game sounds, 2) menu sounds, and 3) music. Game sounds are paused when the player opens a menu (including the main menu). Menu sounds are not paused (menu sounds are designed to be used while inside a menu). Music is similar to menu sounds in that it is also played while the user is in a menu. The difference is that only one music sound can be playing. Music sounds can be several minutes long, but game and menu sounds should be relatively short.

The functions `ga_play_sound` and `ga_play_sound_menu` play game sounds and menu sounds respectively. The functions `ga_play_sound_with_volume` and `ga_play_sound_menu_with_volume` are like their counterparts except you specify a volume. The volume should be between 0.0 and 1.0 inclusive.

Once a game or menu sound is played it cannot be stopped. Music, on the other hand, can be stopped at any time.

The sound name string that is specified should be listed in

"Sounds/sound_names.txt".

The function `ga_sound_name_to_fn` converts the given sound name to the filename of the sound. The function `ga_get_sound_names_with_prefix` lists all sound names that start with the given string prefix.

The function `ga_get_current_music_fn` gets the current filename of the music song that is playing.

The function `ga_play_playlist` plays the specified playlist. See the game's website for how the playlist system works exactly. That is, follow this link

http://danthemanhathaway.com/ComputerGames/FractalBlockWorld/ReleaseMisc/Packages

and look for the music guide. Basically every folder in the Music directory of the package is a playlist. The top Music directory itself is not a playlist, and instead the "top level" playlist should have its songs be placed in the folder "Music/default". Playlist folders can be nested. That is, we could have the following playlists:

```
Data/Packages/xar/Music/default
Data/Packages/xar/Music/ying_world
Data/Packages/xar/Music/ying_world/small_yellow_flower
```

If `ga_play_playlist("small_yellow_flower")` is called, the engine will try to play music from the `small_yellow_flower` playlist. However if that folder is empty, ift will try the `ying_world` folder. If that is empty, it will try the default folder.

The function `ga_playlist_exists` returns whether the given playlist folder can be found, not whether or not it is empty.

17.13 Game API: Input Binds

```
void ga_enable_non_escape_binds(bool value);
void ga_enable_escape_bind(bool value);
string ga_what_binds_to_action(string action);
```

By default, non-escape binds are enabled. The function

`ga_enable_non_escape_binds`

specifies whether non-escape binds are enabled. When they are disabled, only the binds for opening the main menu and the console are executed. You might want to disable non-escape binds if you have a HUD window which when open processes all input. Note that on the other hand, when a game window or main menu window is open, then binds are not executed.

The function

`ga_enable_escape_bind`

sets whether the escape key bind is enabled. Strictly speaking, it is not the escape key. Rather, the escape bind is the bind associated to the action

`PACKAGE_OPEN_MAIN_MENU`

(which is normally bound to the escape key). You would want to set this to false if you want a HUD window to process input from the user and you want the escape key to cause the window to close instead of exiting to the main menu.

In terms of geting information about input binds, we recommend using `ga_command` in combination with the command `bind`. For example, this is how you can get a semicolon separated list of all actions which bind to a particular action:

```
local inputs_str = ga_command("bind get_inputs PACKAGE_MOVE_FORWARD")
ga_print(inputs_str)
```

That code will print "W.downup" (without quotes) in the default configuration of the xar package. However, we understand that the typical case is that only one action binds to an input event, which is a keyboard key, and we want to strip the ".downup" part. For this, you can use the function `ga_what_binds_to_action`:

```
local key_str = ga_what_binds_to_action("PACKAGE_MOVE_FORWARD")
ga_print(key_str)
```

In the same situation as the previous example, the last example will print "W" (without quotes). If the action string is not found, the function will return "ERROR" (without quotes). If the action exists but no input event binds to it, the function will return the empty string. If more than one input event binds to the action, it will just return the first one.

17.14 Game API: Meshes

```
string ga_mesh_get_tex(string mesh_name);
float  ga_mesh_get_radius(string mesh_name);
float  ga_mesh_get_inv_radius(string mesh_name);
```

Recall that mesh names are associated to mesh files and textures in

`Meshes/mesh_names.txt`.

The "radius" of a mesh is the largest distance of any vertex in the mesh from the origin (of the mesh).

The function `ga_mesh_get_tex` gets the texture name associated to the given mesh.

The function `ga_mesh_get_radius` returns the radius of the specified mesh.

The function `ga_mesh_get_inv_radius` returns 1.0 divided by this radius. This is provided because it might be slightly faster (it is faster because these values are cached).

17.15 Game API: Game Related

17.15.1 ga_ga_get_package_name

```
string ga_get_package_name();
```

This returns the name of the current package. For example if the player is using the Data/Packages/xar package, this function will return "xar".

17.15.2 ga_get_current_packages

```
LIST ga_get_current_packages();
```

This gets the list of all packages that are currently being used. So, you can use this to see what mods are being used. Here is an example:

```
local list = ga_get_current_packages()
for _,v in ipairs(list) do
    ga_print("using package: " .. v.name)
end
```

17.15.3 ga_ga_is_cheating_enabled

```
bool ga_is_cheating_enabled();
```

The function `ga_is_cheating_enabled` returns whether or not cheating is enabled.

17.15.4 Hardcore mode

```
bool ga_get_hardcore_mode();
void ga_set_hardcore_mode(bool value);
```

Every player has a hardcore bool. If this bool is true, then if they try to exit the game or load a game, it will forcibly save the game. Also, on hardcore mode the respawn command will not respawn the player. Ultimately the player is on their honor not to make a hack around this.

17.15.5 ga_kill_player

```
bool ga_kill_player();
```

The function ga_kill_player first checks if metagame.cheat.god is true. If so, the function returns false and nothing else happens.

Next, the function ga_kill_player checks if the variable game.player.alive is true. If it is false, the function returns false and nothing else happens. If it is true, then it sets it to false and calls the function top.killed_player (in the script Game/top.lua).

For completeness, let us explain more of the life/death process. We already mentioned that calling the C-API function ga_kill_player will set an internal variable and will in turn call the Lua function top.killed_player. So how does the player respawn? The player respawns by calling the system command "respawn force" or "respawn passive". For example, there can be a death window that gets pushed onto the game window stack. When the correct button is pushed, the window can execute the command

ga_command("respawn passive")

When the engine respawns the player and is finished, the engine calls the function top.respawn_player().

17.16 Game API: Use and Look Objects

```
bool   ga_use_object_exists();
string ga_use_object_get_type();

bool          ga_look_object_bent_exists();
int           ga_look_object_bent_get_chunk_id();
LocalBlockPos ga_look_object_bent_get_lbp();

bool ga_look_object_ment_exists();
int  ga_look_object_ment_inst_id();

bool          ga_look_object_block_exists();
int           ga_look_object_block_get_chunk_id();
LocalBlockPos ga_look_object_block_get_lbp();
int           ga_look_object_block_get_normal_side();
Vector        ga_look_object_block_get_lp();
```

The closest basic entity you are looking at is called your "**look bent**". The function `ga_look_object_bent_exists` will tell you if there is a basic entity in your line of sight. If there is one, then the functions

`ga_look_object_bent_get_chunk_id`

and

`ga_look_object_bent_get_lbp`

will tell you information about it. With the current version of the engine, to have a look bent, it cannot be too far away.

Similarly, you have a "**look ment**" and a "**look block**". See the functions above for getting information about these objects.

Now assume that you have either a look bent, look ment, or look block object. Let U be the closest of the three to the player. If the object U can be "used", then U is called the "use object" of the player. Note that if you

turn your head, you can easily not have a use object anymore. The function `ga_use_object_exists` tells you whether or not you have a use object. If you do have one, then as we said before, it will either be your look bent, look ment, or look block object. To know which type of object your use object is, call the function `ga_use_object_get_type`. This will return either "bent", "ment", or "block" (without quotes).

The "use" command causes the engine to call the corresponding `__use` Lua function on the current use object.

Similarly, the command "use 2" causes the engine to call the corresponding `__use2` Lua function on the current use object.

17.17 Game API: System HUD Related

```
void ga_hud_msg(string msg, float duration);
```

This puts a message close to the center of the screen. It is displayed for duration many seconds, unless another message is displayed in place of it.

```
void ga_hud_reg_damage_from_dir(int damage, Vector dir);
```

This puts a pink (or whatever color) solid circle near the center of the screen which indications that an attack was made to the player from a certain direction. The larger the circle, the more damage was dealt to the player. The dir points from the player to where the damage comes from.

```
void ga_hud_reg_damage_from_dir_color(
    int damage, Vector dir, Vector color);
```

This is just like the previous function, except it is rendered using the specified color.

```
void ga_hud_reg_dir_tex(
    string name, string tex,
    Vector dir, float duration);
```

This puts a textured square near the center of the screen in the same area that attacks to the player are displayed. The exact location of the square indicates "what direction the picture refers to". It stays on the screen for duration many seconds.

17.18 Game API: Moving The Player Through Chunk Tree

```
void ga_shrink();
bool ga_shrink2(Vector lp);
void ga_grow();
```

```
void ga_grow2(Vector lp);

void ga_tele(string path, Vector offset);
void ga_set_tele_back(string path, Vector offset);
void ga_tele_back();

void ga_tele_same_level(Vector lp);
```

For these functions here that return bools, they return true iff the operation was successful.

The function `ga_shrink` will shrink the player one level (at their current location). The function `ga_shrink2` first teleports the player to the specified position ("level position" lp) on the same level, then the player will shrink from that location. If the player cannot teleport there, they will just shrink at their current location.

The functions `ga_grow` and `ga_grow2` are just like `ga_shrink` and `ga_shrink2`, except with growing one level instead of shrinking one level.

The function `ga_tele` will teleport the player to the given chunk with the specified offset within that chunk. This can be done even if the target chunk is not in the active chunk tree.

A common situation is we want to teleport the player to a location, but if there is a problem then we want to teleport them back to where they came from. Call the function `ga_set_tele_back` to set the location of where the player needs to be teleported back to. Call the function `ga_tele_back` to actaully teleport the player back to that point.

If you do not call `ga_set_tele_back` but call `ga_tele_back`, then the engine will send the player back to the position of where they were when they called `ga_tele`. So this is important: If you choose to call `ga_set_tele_back`, you must call it immediately *after* the call to `ga_tele`. You might find that there is no need to call `ga_set_tele_back`. See the Custom Teleportation guide on the game's website for more on this.

The function `ga_tele_same_level` will teleport the player to the specified location within the same level. If the target chunk is not in the active chunk tree, the teleportation will not happen.

17.19 Game API: Exploration

Exploration is what we call the process of chunks being loaded as the player moves through the world.

```
int  ga_get_fertile_radius(int level_delta);
void ga_set_fertile_radius(int level_delta, int radius);

int  ga_get_level_radius(int level_delta);
void ga_set_level_radius(int level_delta, int radius);
```

Every level L has a level radius and a fertile radius. These numbers are determined by the delta from L to the viewer's level.

Basically, the "level radius" is the distance (in chunks) in which blocks get loaded. The "fertile radius" is the distance (in chunks) in which other entities are loaded, such as bents and ments.

The `level_delta` in these functions is the difference between the target level and the viewer's level (it should be a non-negative integer).

Here is an example of how to set the fertile and level radius to be 1 for the 5 finest levels (this is a very small radius):

```
local radius = 1
ga_set_fertile_radius(0, radius)
ga_set_fertile_radius(1, radius)
ga_set_fertile_radius(2, radius)
ga_set_fertile_radius(3, radius)
ga_set_fertile_radius(4, radius)
ga_set_level_radius(0, radius)
ga_set_level_radius(1, radius)
ga_set_level_radius(2, radius)
ga_set_level_radius(3, radius)
ga_set_level_radius(4, radius)
```

So let L be the level of the viewer. Then levels L, $L-1$, $L-2$, $L-3$, and $L-4$ all have a fertile and level radius of 1.

Note: even though we are providing these functions, if you are making a package for other people you should not abuse these functions. A reasonable use for them is to make a keybinding for the player to quickly change their level and fertile radii. This is more something for the end user than someone making a package.

```
void ga_explore_while(string func, string win);
```

The concept of an "explore while loop" is a little complicated. Basically, a common task is for the engine to teleport the player through the chunk tree, but the Lua code of the package needs to guide the engine through this teleportation process. As a special case, when the engine teleports the player from chunk A to chunk B, the package may need to check if chunk B is really safe to teleport to. The function `ga_explore_while` specifies a function to be called every frame. As long as this function returns true, the engine will continue "exploring" (loading chunks around the player's chunk) and will not call any other Lua functions. Once that function returns false, the explore while loop "is over" and the engine resumes calling normal game Lua functions. The function `ga_explore_while` also specifies a window script to be used for rendering and input for the duration of the explore while loop. Note that you might want to call `ga_render_skip_next_frame` during the explore while loop to suppress rendering (to make every frame faster). See the Blue Rings guide on the game's website for more on explore while loops. See also the Custom Teleportation guide in the same place.

17.20 Game API: Worldgen

```
void ga_worldgen_main_dummy(string bt, string arg);
void ga_worldgen_refresh_state_var(string var);
void ga_worldgen_refresh_state_all();
```

See Section 7.4 for what the `ga_worldgen_main_dummy` function does. Basically, the string argument you pass to it is sent to the `__main_dummy` function of the specified block script.

The refresh state var and refresh state all functions flush global variables that start with `worldgen.state` to the worldgen engine. The function

`ga_worldgen_refresh_state_var`

flushes a single variable, whereas

`ga_worldgen_refresh_state_all`

flushes all such variables (but is slower).

17.21 Game API: Windows (Part 1)

```
void ga_window_push(string win_name);
void ga_window_pop();
void ga_window_pop_all();
void ga_hud_window_add(string win_name, int priority);
void ga_hud_window_remove(string win_name);

//These next 3 functions are deprecated.
void ga_main_menu_push(string win_name);
void ga_main_menu_pop();
void ga_main_menu_pop_all(bool return_to_game);
```

There are two types of windows: game windows and hud windows. A window cannot be in more than one category. The game windows are put into a stack (the window stack). However the HUD windows are put into a set.

The functions `ga_window_push` and `ga_window_pop` push windows on and off of the game window stack. Only the top window is rendered and only the top window gets user input. The function `ga_window_pop_all` pops ALL windows off of the game window stack.

The functions `ga_hud_window_add` and `ga_hud_window_remove` will add and remove windows from the HUD window set. These windows should be mostly transparent, so the order in which these windows are rendered is important. Windows are rendered with lower priority numbers first. So priority 1 would be rendered before priority 2, making 2 “on top”.

The functions `ga_main_menu_push`, `ga_main_menu_pop`, and `ga_main_menu_pop_all` are deprecated (the game window stack and main menu window stack were merged into one stack).

```
LIST ga_get_window_stack_display_names();
LIST ga_main_menu_get_stack_display_names(); //Deprecated.
void ga_open_title_menu();
void ga_open_console();
void ga_close_console();
bool ga_get_program_has_focus();
void ga_set_game_paused_while_on_win_stack(bool value);
```

The function `ga_get_window_stack_display_names` returns the list (in order) of the "display names" of all windows on the window stack. Do not use `ga_main_menu_get_stack_display_names` because it has been deprecated (the game window stack was merged with the main menu window stack). Technically it returns an array of tables, where each table has a string member called "name". The display name of a window is specified by the `__get_path_display_name` function of that window script.

The function `ga_open_title_menu` opens the title menu. The title menu is the screen that says "new game", "load game", "save game", "options", and "quit".

The functions `ga_open_console` and `ga_close_console` open and close the console.

The function `ga_get_program_has_focus` returns whether or not the game "has the focus". It might be useful to call this in the `__update_always` function of a window script.

By default, the game is paused when there is at least one window on either the game window stack of the main menu window stack. However, you can call `ga_set_game_paused_while_on_win_stack` to specify whether the game should be paused in this situation. If you want the game to be paused when some windows are on top of the stack but not others, it is up to you to make this work. For example, you could have the update and update passive functions of top.lua call a function which sees which window is at the top of the game or main menu window stack, and that function could call `ga_set_game_paused_while_on_win_stack` accordingly.

17.22 Game API: Viewer Queries

```
int ga_get_viewer_chunk_id();
int ga_get_viewer_ancestor_chunk_id(int level);
string ga_get_viewer_chunk_bt();
int ga_get_viewer_level();
Vector ga_get_viewer_offset();
Vector ga_get_viewer_lp(int level);
BlockPos ga_get_viewer_bp(int level);
string ga_get_viewer_path();
string ga_get_viewer_path_ext();
Vector ga_get_vec_to_viewer(int level, Vector lp);
```

```
float ga_lbp_dist_to_viewer(int chunk_id, int lbp_hash);
float ga_block_dist_to_viewer(int level, BlockPos bp);

//Cached ment variables.
Vector ga_ment_get_var_special_vec_to_viewer(int inst_id);
float ga_ment_get_var_special_dist_to_viewer(int inst_id);
```

The official position of the player is the camera position, which we call the viewer position.

The function `ga_get_viewer_chunk_id` gets the chunk id of the viewer. The function `ga_get_viewer_ancestor_chunk_id` gets the chunk id of a chunk on the chunk path of the viewer's chunk.

The function `ga_get_viewer_chunk_bt` is a helper function to get the block type of the chunk of the viewer. Note that another way to get this is by first get the chunk id of the chunk of the viewer, then use that to get the block position of that chunk, and then calling ga_block_get.

The function `ga_get_viewer_level` gets the level that the viewer is on (the level of the viewer chunk). Note that the viewer chunk is the finest chunk which contains the viewer's position.

`ga_get_viewer_chunk_bt` returns the block type of the chunk containing the viewer. Although you can figure this out using other functions, we provide this shortcut out of convenience.

`ga_get_viewer_offset` gets the offset of the viewer relative to the chunk that contains the viewer. So the viewer offset should be a vector between `(0.0, 0.0, 0.0)` and `(16.0, 16.0, 16.0)`.

`ga_get_viewer_lp` gets the *level position* (`lp`) of the viewer. This is the position of the viewer on the specified level. Note that on the viewer level, the `ga_get_viewer_lp` should return the same as `ga_get_viewer_offset`.

`ga_get_viewer_bp` gets the position of the block (on the specified level) which contains the viewer.

`ga_get_viewer_path` gets the path of the chunk which contains the player. The function `ga_get_viewer_path_ext` gets the path of the block which contains the player (so that path is one longer than that of `ga_get_viewer_path`. Note that functions like this that get paths might be slow if the paths are really long.

`ga_get_vec_to_viewer` returns the difference between the viewer's level position and the specified vector. The result will point from the specified vector to the viewer.

`ga_lbp_dist_to_viewer` returns the distance from the center of the given block to the viewer (on the level of the block).

`ga_block_dist_to_viewer` does the same.

`ga_ment_get_var_special_vec_to_viewer` returns the vector from the specified moving entity to the viewer (on the level of the moving entity).

`ga_ment_get_var_special_dist_to_viewer` returns the distance from the specified moving entity to the viewer (on the level of the moving entity).

17.23 Game API: Basic Entities

17.23.1 Getting and setting basic entities

```
string ga_bent_get_type(int level, BlockPos bp);
```

This ga_bent_get_type function returns the type of the basic entity at the given location. If there is no basic entity there, it will return the empty string.

```
void ga_bent_add(int level, BlockPos bp, string type, float rl);
void ga_bent_add_i(int level, BlockPos bp, string type, int param, float rl);
void ga_bent_add_s(int level, BlockPos bp, string type, string param, float rl);
```

Basic entities (BEnts) have a string parameter and an integer parameter. These three functions will create a new basic entity at the specified level and block position with the specified revert length (rl). The function ga_bent_add_i creates a basic entity with a specified integer parameter. The function ga_bent_add_s similarly creates a basic entity with a specified string parameter. To set both the integer and string parameters, use the ga_bet_set_param_XXX functions.

```
void ga_bent_set_param_i(int level, BlockPos bp, int value, float rl);
void ga_bent_set_param_s(int level, BlockPos bp, string value, float rl);
```

These two functions will set the integer and string parameters of the basic entity at the given block position.

```
int    ga_bent_get_param_i(int level, BlockPos bp);
string ga_bent_get_param_s(int level, BlockPos bp);
```

These two functions will get the integer and string parameters of the basic entity at the given block position.

```
void ga_bent_remove_temp(int level, BlockPos bp, int num_sec);
void ga_bent_remove_perm(int level, BlockPos bp);
```

The ga_bent_remove_temp will remove the basic entity for a given number of seconds. Note that this will remove ANY basic entity from that location. The function ga_bent_remove_perm will *permanently* remove the basic entity (and any basic entity) from the given location.

17.23.2 ga_bent_sphere_query

```
LIST ga_bent_sphere_query(int level, Vector lp, float radius);
```

The ga_bent_sphere_query returns a list of all the basic entities that are within radius distance of lp on the given level. Here is an example:

```
local level = 5
local lp = std.vec(18.2, 19.7, 20.6)
local radius = 17.4
local list = ga_bent_sphere_query(
    level, lp, radius)
for k,v in pairs(list) do
    local dist = v.dist --Distance of bp center to lp.
    local bp = v.bp     --Block position of bent.
    --Do something with dist and bp!
end
```

The list is ordered by dist (closer bents come first).

17.23.3 ga_search_for_bent_in_chunk

```
CLASS ga_search_for_bent_in_chunk(int chunk_id, string bent_type);
```

The function above searches the given chunk for a basic entity of the given type. Let us show an example:

```
local data = ga_search_for_bent_in_chunk(chunk_id, "bent_gold_500")
if( data.is_valid ) then
    --We have found one such entity.
    --This is the local block position of the found bent:
    local lbp = data.value
else
    --There is no such entity in the chunk.
end
```

17.23.4 ga_get_bent_names_with_prefix

```
LIST ga_get_bent_names_with_prefix(string prefix);
```

Use the function `ga_get_bent_names_with_prefix` to get an array consisting of all names of basic entities that start with the given prefix. Technically it returns an array of tables, where each table has a member called "name" (the string name of the basic entity).

17.24 Game API: Moving Entities (type)

```
bool ga_ment_type_var_exists(string type, string var);
```

The function ga_ment_var_exists returns whether or not a moving entity (type) has a given variable.

```
bool   ga_ment_get_static_b(string type, string var);
int    ga_ment_get_static_i(string type, string var);
float  ga_ment_get_static_f(string type, string var);
Vector ga_ment_get_static_v(string type, string var);
string ga_ment_get_static_s(string type, string var);
```

These functions get the values of static variables for moving entities. Note: if a variable is NOT static, then calling one of these functions will get the *default value* of that variable. Note that the only way to change a static variable (or the default value of a non-static variable) is during the package initialization phase. See the Lua-to-C Initialization API.

```
bool ga_ment_static_b_exists_and_true(string type, string var);
```

The function ga_ment_static_b_exists_and_true is a helper function. It returns whether or not the given moving entity variable exists AND is true.

17.25 Game API: Moving Entities (instance)

17.25.1 Creating a moving entity

```
void ga_ment_start(int level, Vector lp, string type);
void ga_ment_end();
void ga_ment_init_set_b(string key, bool value);
void ga_ment_init_set_i(string key, int value);
void ga_ment_init_set_f(string key, float value);
void ga_ment_init_set_v(string key, Vector value);
void ga_ment_init_set_s(string key, string value);
```

To create a moving entity, you call ga_ment_start and then ga_ment_end. In between you call functions ga_ment_init_set_XXX to set variables of the moving entity. The ga_ment_start requires the level of the moving entity as well as the level position and the type name of the moving entity.

17.25.2 Getting moving entity variables

```
bool   ga_ment_get_b(int inst_id, string var);
int    ga_ment_get_i(int inst_id, string var);
float  ga_ment_get_f(int inst_id, string var);
Vector ga_ment_get_v(int inst_id, string var);
string ga_ment_get_s(int inst_id, string var);
```

Use these ga_ment_get_XXX functions to get the variables of a moving entity. If the variable does not exist, the program will exit.

17.25.3 Testing if a variable exists

```
bool ga_ment_var_exists(int inst_id, string var);
```

Returns whether or not the given var exists. Although this can be accomplished with `ga_ment_var_exists` using the *type* of the ment, we proivde this shortcut for convenience. Note: This function used to be called

`ga_ment_var_exists2`,

but we renamed it to `ga_ment_var_exists`. The engine currently still accepts `ga_ment_var_exists2`, but eventually it will not.

The old `ga_ment_var_exists` function was renamed to

`ga_ment_type_var_exists`.

17.25.4 Getting the type of a variable

```
string ga_ment_get_var_type(int inst_id, string var);
```

The function `ga_ment_get_var_type` returns one of the following strings depending on the type of the var: `"b"`, `"i"`, `"f"`, `"v"`, `"s"`.

17.25.5 Checking if an ment bool exists and is true

```
bool ga_ment_b_exists_and_true(int inst_id, string var);
```

The ga_ment_b_exists_and_true is a helper function which returns true iff the moving entity has the bool variable AND the variable is true.

17.25.6 Changing the revert length of a variable

```
void ga_ment_set_var_rl_only(int inst_id, string var, float rl);
```

Use the `ga_ment_set_var_rl_only` function for changing the revert length of a variable for (this instance of) a moving entity. This does NOT change the value of the variable. Note that you can also set the revert length of a variable in the initialization API with `ia_ment_set_var_rl_only`.

17.25.7 Setting moving entity variables

```
void ga_ment_set_b(int inst_id, string var, bool value);
void ga_ment_set_i(int inst_id, string var, int value);
void ga_ment_set_f(int inst_id, string var, float value);
void ga_ment_set_v(int inst_id, string var, Vector value);
void ga_ment_set_s(int inst_id, string var, string value);
```

Use these ga_ment_get_XXX functions to set the variables of a moving entity. If the variable does not exist, the program will exit. The revert time is already specified (it is associated to the variable).

```
void ga_ment_toggle_b(int inst_id, string var);
void ga_ment_set_i_by_delta(int inst_id, string var, int delta);
void ga_ment_set_f_by_delta(int inst_id, string var, float delta);
void ga_ment_set_v_by_delta(int inst_id, string var, Vector delta);
```

These are helper functions for modifying moving entity variables. These are analogous to the functions `ga_toggle_b` and `ga_set_XXX_by_delta` for environment variables.

17.25.8 Inst ID and code ID

```
int ga_ment_inst_id_to_code_id(int inst_id);
int ga_ment_code_id_to_inst_id(int code_id);
```

Every moving entity has an instance ID and a code ID. The instance ID is only valid until the player either exists the program or loads a game. The code ID, on the other hand, is persistent. Use these functions to convert to and from these two types of IDs.

If the code id cannot be found by ga_ment_code_id_to_inst_id, it returns -1. If the inst id cannot be found by ga_ment_inst_id_to_code_id, it returns -1.

A code ID ≥ 0 indicates that the moving entity is "roaming". A code ID < -1 indicates that the moving entity is not "roaming" (and was therefore originally created by procedural world generation).

17.25.9 Testing if a moving entity exists

```
bool ga_ment_exists(int inst_id);
```

The ga_ment_exists returns whether or not the given (instance of a) moving entity exists.

17.25.10 Removing a moving entity

```
ga_ment_respawn(int inst_id);
ga_ment_remove(int inst_id);
ga_ment_remove_perm(int inst_id);
ga_ment_remove_with_respawn_length(int inst_id, float respawn_length);
```

These functions will make a request to do something. The request will be fulfilled slightly later (at most 0.04 seconds later). The delay is for safety. There are also `_now` versions of these functions which are fulfilled immediately.

The `ga_ment_respawn` function will remove the given moving entity (instance) and it will respawn within the next few seconds.

The `ga_ment_remove` function will remove the giving moving entity and it will be respawn RL seconds later, where RL is the value of the moving entity variable `__respawn_length` of the moving entity.

The `ga_ment_remove_perm` will remove the moving entity and it will not respawn.

The `ga_ment_remove_with_respawn_length` function removes the moving entity and it will respawn in the specified number of seconds later.

Note: Only moving entities created from procedural world generation will respawn. Roaming entities, on the other hand, will be completely removed when you remove them with any one of these functions (and they will not respawn).

```
ga_ment_respawn_now(int inst_id);
ga_ment_remove_now(int inst_id);
ga_ment_remove_perm_now(int inst_id);
ga_ment_remove_with_respawn_length_now(int inst_id, float respawn_length);
```

These remove the moving entity immediately, but they are more error prone. Unless you have a good reason, use the normal versions instead of the `_now` versions.

17.25.11 Getting the type string of a moving entity

```
string ga_ment_get_type(int inst_id);
```

The function ga_ment_get_type gets the moving entity type of the moving entity instance.

17.25.12 Getting the level position

```
Vector ga_ment_get_lp(int inst_id);
```

The function ga_ment_get_lp gets the position of the moving entity on its level (its "level position"). Note: to get the level of the moving entity, call ga_ment_get_i(inst_id, "__level").

17.25.13 Getting the starting level level position

```
Vector ga_ment_get_sllp(int inst_id);
```

The starting level of a moving entity is the level of the first chunk that the entity spawned into. It is very common to convert the level position (lp) of a moving entity to the starting level. We call this the starting level level position (sllp) of the moving entity. The function `ga_ment_get_sllp` returns just that.

17.25.14 Getting the starting level position

```
int ga_ment_get_start_level(int inst_id);
```

This just returns the `__start_level` variable of the ment. When an ment is added to the world, this value is set.

17.25.15 Getting the level

```
int ga_ment_get_level(int inst_id);
```

This gets the level that the ment is on. This may not be the same as the "index level" of the ment. The index level is used for collision detection. Basically, the index level is the finest level where the radius of the ment on that level is less than the chunk width.

17.25.16 Getting the level

```
int ga_ment_get_chunk_id(int inst_id);
```

This gets the `chunk_id` of the chunk which "contains the ment". This will be the finest chunk in the active chunk tree which contains the center of the ment.

17.25.17 Getting the radius

```
float ga_ment_get_radius(int inst_id);
```

Gets the radius of the ment, on the level returned by ga_ment_get_level.

17.25.18 Dumping a moving entity

```
void ga_ment_dump(int inst_id);
```

The function `ga_ment_dump` prints to stdout.txt relevant information about the given moving entity.

17.25.19 Teleporting a moving entity

```
void ga_ment_tele(int inst_id, int chunk_id, Vector offset);
```

This function makes a request to teleport the given moving entity to the specified chunk. This request will be fulfilled at the end of the current discrete update cycle. If the chunk with the given chunk id exists, it will move the moving entity there and will use the given offset.

17.25.20 Sphere query

```
LIST ga_ment_sphere_query(
    int level, int min_level, int max_level,
    Vector lp, float radius);
```

The ga_ment_sphere_query returns a list of all the moving entities that are within radius distance of lp on level level. Also we consider moving entities that are on the levels between min_level and max_level inclusive. Here is an example:

```
local level = 5
local min_level = 4
local max_level = 6
local lp = std.vec(18.0, 19.0, 20.0)
local radius = 17.4
local list = ga_ment_sphere_query(
    level, min_level, max_level,
    lp, radius)
for k,v in pairs(list) do
    local ment_inst_id = v.inst_id
    local dist_to_ment = v.dist
    --Do something with ment_inst_id and dist_to_ment!
end
```

The list is ordered by dist (closer ments come first).

17.25.21 Alarms

```
void ga_ment_set_alarm(
    int inst_id, float alarm_game_time, string alarm_name);
void ga_ment_set_alarm_on_level(
    int inst_id, int level, float alarm_level_time, string alarm_name);
```

An alarm is maintained by the engine (but is NOT saved when the game is saved). A moving entity can set an alarm. The alarm is associated to the moving entity instance (the inst_id) and the alarm also has a name. When the time comes, the alarm goes off and the moving entity is called back (the function on_alarm of the moving entity is called). There are two types of alarms: normal (game) and level. A normal (game) type alarm goes off at the given game time. A level type alarm goes off when the specified level time occurs.

17.25.22 Dumping all moving entities

```
void ga_ment_all_dump();
```

This will dump information about ALL moving entities that exist in the active chunk tree.

17.25.23 The vector from one ment to antoher

```
void ga_ment_vec_to_another_ment(int inst_id_1, int inst_id_2);
```

Let v_1 be the position of the ment with the first instance id and the v_2 be the position of the ment with the second instance id. This function returns $v_2 - v_1$, calculated on the level of the first moving entity. So, the returned vector points from v_1 to v_2.

17.25.24 Listing moving entity types

```
LIST ga_get_ment_names_with_prefix(string prefix);
```

The function above returns an array of all ment type names that start with the given prefix. Technically it returns an array of tables, where each table has a string member called `name`.

17.26 Game API: Particles

A particle is a point like entity used for rendering only. They are not saved when the game is saved.

17.26.1 Adding a single particle

```
void ga_particle_add(CLASS args);
```

This function ga_particle_add adds a single particle. There are many parameters which are passed as a single class to the function. The following example illustrates this:

```
local args = {}
args.level = start_level
args.pos = std.vec(4.0, 5.0, 6.0)
args.ttl = 1.0
args.size = 0.2
args.color = std.vec(1.0, 1.0, 1.0)
args.fade_time = 0.5
args.vel = std.vec(0.0, 0.0, 10.0)
args.tex = "particle_2"
args.use_min_dist = false
ga_particle_add(args)
```

The use_min_dist determines whether or not particles will be removed if they are too close to the viewer. If use_min_dist is true, then this will be the case.

17.26.2 Adding a spherical explosion of particles

```
void ga_particle_explosion(CLASS args);
```

This adds a spherical explosion of particles. See the following example:

```
local args = {}
args.level = 10
args.pos = std.vec(6.0, 7.0, 8.0)
args.ttl_min = 10.0
args.ttl_max = 20.0
```

```
args.size_min = 0.2
args.size_max = 1.0
args.color = col
args.fade_time_min = 10.0
args.fade_time_max = 10.0
args.speed_min = 0.5
args.speed_max = 0.5
args.tex = "particle_2"
args.radius_min = 4.0
args.radius_max = 4.0
args.num = 200
args.use_min_dist = false
ga_particle_explosion(args)
```

Radius_XXX is the distance of each particle from the center.

17.26.3 Adding a line of particles

```
void ga_particle_trail(CLASS args);
```

This adds a line of particles. See the following example:

```
local args = {}
args.level = 10
args.pos_start = std.vec(2.0, 2.0, 2.0)
args.pos_end = std.vec(12.0, 13.0, 14.0)
args.ttl_min = 0.5
args.ttl_max = 0.5
args.size_min = 0.1
args.size_max = 0.1
args.color = std.vec(1.0, 1.0, 1.0)
args.fade_time_min = 0.5
args.fade_time_max = 0.5
args.speed_min = 0.0
args.speed_max = 0.0
args.tex = "particle_2"
args.radius_min = 0.0
args.radius_max = 0.0
args.avg_len = 1.0
args.use_min_dist = false
ga_particle_trail(args)
```

The parameters radius_XXX specify the distance of the particle from the line between pos_start and pos_end.

17.26.4 Adding a ring of particles

```
void ga_particle_ring(CLASS args);
```

This adds a ring of particles. See the following example:

```
local args = {}
args.level = 10
args.pos = std.vec(6.0, 7.0, 8.0)
args.normal = std.vec(0.0, 0.0, 1.0)
args.ttl_min = 1.0
args.ttl_max = 2.0
args.size_min = 0.4
args.size_max = 0.6
args.color = std.vec(0.0, 0.0, 1.0)
args.fade_time_min = 1.0
args.fade_time_max = 1.0
args.tex = "particle_2"
args.radius = 1.0
args.speed = 4.0
args.num = 100
args.use_min_dist = false
ga_particle_ring(args)
```

The parameter radius is the distance of the particles from the center position. The particles move away from the center position with the given speed.

17.27 Game API: Blocks (type)

17.27.1 Getting information about a block type

```
bool ga_bt_exists(string bt);
bool ga_bt_var_exists(string bt, string var);
bool ga_bt_get_physically_solid(string bt);
```

In this document, bt stands for "block type". Recall that block type strings must either start with `block_` (if they correspond to a Lua script) or `XAR_` (if it is built into the engine).

The function `ga_bt_exists` returns whether or not there exists a block type of that name. For example,

```
ga_bt_exists("XAR_SMALL_YELLOW_FLOWER")
```

returns true.

The function `ga_bt_var_exists` returns whether or not the given block type has a block variable of the specified name.

The function `ga_bt_get_physically_solid` returns whether or not the block (of the given type) is physically solid.

Note that there is a way you can query information about your own block Lua scripts. For example, suppose each of your block Lua scripts has a function called `is_funky` which returns a bool. Then to determine if a block type is funky, you can run the following code:

```
function p.is_bt_funky(bt)
    local mod_name = bt
    local func_name = "is_funky"
    if( _G[mod_name] and
        _G[mod_name][func_name] )
    then
        --Calling the function in the block lua script.
        return _G[mod_name][func_name]()
    else
        return false --It is not funky.
    end
```

17.27.2 Listing block types

```
LIST ga_get_block_names_with_prefix(string prefix);
```

The function above returns an array of all block types that start with the given prefix. Here is an example:

```
local bt_array = ga_get_block_names_with_prefix("XAR_")
for k,v in ipairs(bt_array) do
    local bt = v.name --The bt is actually in a member called "name".
    ga_print(bt)
end
```

17.28 Game API: Blocks (instance)

17.28.1 Old functions to get a block type

```
string ga_block_get(int level, BlockPos bp);
string ga_get_cocoon_block_of_chunk(int level, BlockPos vcp);
```

We recommend NOT using these functions because their names do not indicate what the arguments should be. Use the new functions instead. These old functions might be deprecated at some point.

The function `ga_block_get` returns the block type (string) of the block with the given position. If it is a "xar" block (built into the program), it will start with "`XAR_`". On the other hand, all blocks defined by Lua scripts will start with "`block_`".

The function `ga_get_cocoon_block_of_chunk` returns the block type of the block which occupies the same volume as the given chunk. The chunk is specified by its level and vcp (viewer centric position).

17.28.2 New functions to get a block type

```
string ga_bp_to_bt(int level, BlockPos bp);
string ga_chunk_id_and_lbp_to_bt(int chunk_id, LocalBlockPos lbp);
string ga_chunk_id_to_bt(int chunk_id);
string ga_vcp_to_bt(int level, ViewerCentricPos vcp);
```

Use these functions to get the block type (string) of any block. Note that you can also get the block type of chunks (because every chunk is itself a block, except for the root chunk of the world).

The functions `ga_bp_to_bt` and `ga_chunk_id_and_lbp_to_bt` get the block type of the specified *block*. The functions `ga_chunk_id_to_bt` and `ga_vcp_to_bt` get the block type of the specified *chunk*.

17.28.3 Changing a block

```
void ga_block_change_rl(
    int level, BlockPos bp, string new_bt, float rl);
void ga_block_change_rl_default(
    int level, BlockPos bp, string new_bt);
void ga_block_change_perm(
    int level, BlockPos bp, string new_bt);
```

The function `ga_block_change_rl` changes the type of a block. A "revert length" is specified (in seconds). After this amount of time, the block is reverted to its previous state.

The function `ga_block_change_rl_default` changes the type of a block but the revert length is specified by the static variable "`__revert_length_default`" associated to the block type.

The function `ga_block_change_perm` permanently changes the type of a block. Specifically, this is the same as calling `ga_block_change_rl` but with a fixed very large revert length.

Note that in Fractal Block World we actually store a "stack of blocks" at each block position, not a single block. This concept is explained more in Section 16.4. So calling one of these "block_change" functions will push a block onto a block stack.

17.28.4 Getting block variables

```
bool   ga_block_var_exists(int level, BlockPos bp, string var);
string ga_block_get_var_type(int level, BlockPos bp, string var);

bool   ga_block_get_b(int level, BlockPos bp, string var);
int    ga_block_get_i(int level, BlockPos bp, string var);
float  ga_block_get_f(int level, BlockPos bp, string var);
Vector ga_block_get_v(int level, BlockPos bp, string var);
string ga_block_get_s(int level, BlockPos bp, string var);
```

```
bool ga_block_b_exists_and_true(int level, BlockPos bp, string var);
```

Use the function `ga_block_var_exists` to see if the block has the specified block variable. The "get" functions are used to get the values of the block variables. If the variable does not exist, the program will exit. The function `ga_block_var_exists_and_true` is a helper function which returns if the (bool) variable exists AND is true.

The function `ga_block_get_var_type` returns one of the following strings depending on the type of the var: `"b"`, `"i"`, `"f"`, `"v"`, `"s"`.

17.29 Setting block variables

```
void ga_block_set_b(int level, BlockPos bp, string var, bool value);
void ga_block_set_i(int level, BlockPos bp, string var, int value);
void ga_block_set_f(int level, BlockPos bp, string var, float value);
void ga_block_set_v(int level, BlockPos bp, string var, Vector value);
void ga_block_set_s(int level, BlockPos bp, string var, string value);
```

Use the above functions to set block variables. If the variable does not exist, the problem will exit.

The concept of the "revert time" of a block variable is discussed in Section 16.4.

```
void ga_block_toggle_b(int level, BlockPos bp, string var);
void ga_block_set_i_by_delta(int level, BlockPos bp, string var, int delta);
void ga_block_set_f_by_delta(int level, BlockPos bp, string var, float delta);
void ga_block_set_v_by_delta(int level, BlockPos bp, string var, Vector delta);
```

The by_delta functions take the specified variable and add to them the value delta.

17.29.1 Block variables example

Here is a complete example of a soda_machine block which the player can use.

```
function p.get_is_solid() return true end
function p.get_tex() return "block_soda_machine" end
function p.main() set_default_block("s") end

function p.type_init(id)
    ia_block_new_var_i(id, "num_sodas", 10)
end

function p.get_can_use(level, bp)
    return true
end
```

```
function p.get_use_msg(level, bp)
    local num_sodas = ga_block_get_i(level, bp, "num_soads")
    return "Sodas left: " .. tostring(num_sodas)
end

--Drinking a soda gives the player 5 health.
function p.on_use(level, bp)
    local num_sodas = ga_block_get_i(level, bp, "num_sodas")
    if( num_sodas <= 0 ) then return end
    local old_player_health = ga_get_i("var.health")
    local new_player_health = old_player_health + 5
    ga_set_i("var.health", new_player_health)
    num_sodas = num_sodas - 1
    ga_block_set_i(level, bp, "num_sodas", num_sodas)
end
```

Here is how we can rewrite the on_use function to use "set by delta" functions:

```
function p.on_use(level, bp)
    if( ga_block_get_i(level, bp, "num_sodas") <= 0 ) then return end
    ga_set_i_by_delta("var.health", 5)
    ga_block_set_i_by_delta(level, bp, "num_sodas", -1)
end
```

17.29.2 The most common block type

```
string ga_get_most_common_bt_in_chunk(int chunk_id);
```

Use the function above to get the block type which occurs most frequently within the specified chunk. This function is slow.

17.29.3 Searching for blocks

```
CLASS ga_search_for_bt_in_chunk(int chunk_id, string bt);
CLASS ga_search_for_bt_in_chunk_random(int chunk_id, string bt);
```

These will search for a block with the specified type in the given chunk. It will return the first match. The random version is the same, but it searches for the block in a (pseudo) random order. Both of these functions are slow. Here is an example:

```
local chunk_id = ga_get_viewer_chunk_id()
local data = ga_search_for_bt_in_chunk(chunk_id, "XAR_SMALL_YELLOW_FLOWER")
if( data.is_valid ) then
    local pos = data.value --Value is a "block position".
    ga_console_print("found pos = " .. std.vec_to_str(pos))
end
```

17.29.4 Searching and replacing blocks

```
void ga_search_and_replace_bt_in_chunk_perm(
    int chunk_id string bt1, string bt2);
```

Use the above function to replace every block of type `bt1` with a block a type `bt2`. Effectively, the engine calls `ga_block_change_perm` to replace the blocks.

17.29.5 Alternate API for block variables

```
bool   ga_chunk_var_exists(int level, ViewerCentricPos vcp, string var);
bool   ga_chunk_get_b(int level, ViewerCentricPos vcp, string var);
int    ga_chunk_get_i(int level, ViewerCentricPos vcp, string var);
float  ga_chunk_get_f(int level, ViewerCentricPos vcp, string var);
Vector ga_chunk_get_v(int level, ViewerCentricPos vcp, string var);
string ga_chunk_get_s(int level, ViewerCentricPos vcp, string var);
void   ga_chunk_set_b(int level, ViewerCentricPos vcp, string var, bool value);
void   ga_chunk_set_i(int level, ViewerCentricPos vcp, string var, int value);
void   ga_chunk_set_f(int level, ViewerCentricPos vcp, string var, float value);
void   ga_chunk_set_v(int level, ViewerCentricPos vcp, string var, Vector value);
void   ga_chunk_set_s(int level, ViewerCentricPos vcp, string var, string value);
```

These are alternate functions for getting and setting block variables. With these functions, you refer to the chunk that is the block using the chunk's level and VCP.

17.30 Game API: Respawn Point and Waypoints

17.30.1 Respawn point

```
string ga_get_respawn_path();
BlockPos ga_get_respawn_lbp();
void ga_set_respawn_point(string path, BlockPos lbp);
void ga_set_respawn_cb(string func, string win);
```

Use the function `ga_get_respawn_path` to get the path of the chunk where the player will be spawned when they respawn. The function `ga_get_respawn_lbp` returns the local block position of the respawn point within the respawn chunk. To be clear, the "eyes" of the player will be teleported to the center of the respawn block position.

Use the function `ga_set_respawn_point` to set the game's current respawn point. When the player dies, he will respawn there. Note that to respawn, the player should enter the following system command "respawn passive". The lbp should be a local block position, specifying a block between (0,0,0) and (15,15,15) inclusive.

Normally when the engine respawns the player, it puts the player at their respawn point (and the player is in their normal body mode). On the other hand, if you call `ga_set_respawn_cb` you can specify an explore while loop function to be called when the engine respawns the player. Note: cb stands for "call back". That is, the engine will move the player to their respawn point but it will put the player in spirit body mode. It will call the explore while loop function until it returns false. Your explore while loop function should check if the player's chunk is safe to respawn in. If so, you should call `ga_move_set_body_spirit_off` to turn off spirit mode. If the chunk is NOT safe to respawn in, you need to recover accordingly. For example, you could teleport the player to an emergency location.

17.30.2 Waypoints (Deprecated)

```
string ga_get_emergency_waypoint_path();
void ga_add_waypoint_sloppy(string path, string name_override);
void ga_add_waypoint_sloppy_in_only(string path, string name_override);
```

Do not use these functions. Use the correspoding functions in

`Data/Packages/base/Game/game_base_wp_system.lua`

instead. That is, the engine itself no longer has any concept of what a waypoint is.

To make a package with a slightly modified waypoint system, you can override the file

`Data/Packages/base/Game/game_base_wp_system_aux.lua`.

That is, you create a file called `game_base_wp_system_aux.lua` in the `Game` directory of your package which overrides whatever functions you want to define differently. For example, the function

`search_for_target_wp_in_chunk(int chunk_id)`

is intended to search for a waypoint within the given chunk. By redefining this function, you can change what counts as a waypoint entity.

17.31 Game API: Coordinates: Blocks and Chunks

To review, here are what various abbreviations stand for:

```
-- lbp    = "local block position" (a vector of ints)
-- lbph   = "local block position hashcode" (an integer)
-- bp     = "block position" (a vector of ints)
-- offset = "position of a point in a chunk's coordinate system" (a vector of floats)
-- lp     = "level position (of a point in a level)" (a vector of floats)
-- vcp    = "viewer centric position" (a vector of ints)
```

Recall that every chunk is itself a block (except for the root chunk of the world). Every chunk is on a *level*. The root chunk is on level 0. Also, every *block* is on a level. **The level of a block is the same as the level of the chunk which contains the block.**

Thus if C is a chunk on level L, then the level of the "block version of C" is $L-1$. Do not get confused by this.

Every chunk (that is in the active chunk tree) has a *chunk id*, which is a non-negative integer.

Every chunk has what is called its *path* from the root of the chunk tree (represented by a string, such as `777_6e8`).

Finally every chunk (in the active tree) has what is called its *viewer centric position* (VCP), which is the position of the chunk relative to the chunk on the same level which contains the player.

To summarize, we have 4 ways of referring to a chunk:

1) By chunk id,

2) By level + viewer centric position,

3) By level + block position,

4) By path.

Note that if (L, VCP) is used as the VCP representation of the chunk, then its block position representation will be of the form $(L-1, \text{BP})$.

17.31.1 base/Game/std.lua

Many coordinate functions are provided in the Lua in the file base/Game/std.lua. Here are some such functions:

```
vec,
bp,
lbp_to_bp
bp_to_parent_vcp
bp_to_lbp
local_to_level_pos,
level_to_local_pos
lp_to_vcp
lp_to_offset,
block_center,
lbph_to_lbp,
lbp_to_lbph,
lp_to_bp,
side_int_to_str,
side_str_to_int,
side_int_to_vec,
get_adj_bp
```

Some of these functions we provide as part of the Game API as well, for convenience.

17.31.2 From chunk id

```
int              ga_chunk_id_to_level(int chunk_id);
ViewerCentricPos ga_chunk_id_to_vcp(int chunk_id);
BlockPos         ga_chunk_id_to_bp(int chunk_id);
string           ga_chunk_id_to_path(int chunk_id);
```

The function `ga_chunk_id_to_level` returns the level of the given chunk. The function `ga_chunk_id_to_vcp` returns the vcp (in the chunk's level) of the given chunk. The function `ga_chunk_id_to_bp` returns the block position of the given chunk. Note that the level of that block is the level of the chunk *minus one*. The function `ga_chunk_id_to_path` returns the chunk path of the given chunk.

17.31.3 To chunk id

```
int ga_vcp_to_chunk_id(int level, BlockPos vcp);
int ga_path_to_chunk_id(string path);
```

The function `ga_vcp_to_chunk_id` returns the chunk id of a chunk given its level and viewer centric position.

The function `ga_path_to_chunk_id` returns the chunk id of a chunk given its chunk path.

17.31.4 Converting from lbp to bp

```
BlockPos ga_chunk_id_and_lbp_to_bp(int chunk_id, BlockPos lbp);
BlockPos ga_lbp_to_bp(BlockPos vcp, BlockPos lbp);
```

Use these functions for converting from a local block position (which describes the position of a block within a chunk) to a block position. Note that the vcp version can easily be implemented in Lua as a math function. Both versions are provided for convenience.

17.31.5 Converting from bp to chunk id

```
int ga_bp_to_chunk_id(int level, BlockPos bp);
```

This returns the chunk ID of the chunk that is the specified block. Note that the level of the chunk is $level + 1$.

17.31.6 Converting between vcp and bp

```
BlockPos         ga_vcp_to_bp(int level, ViewerCentricPos vcp);
ViewerCentricPos ga_bp_to_vcp(int level, BlockPos bp);
```

These are perhaps the most subtle coordinate functions. It would be a little tricky to try to implement these yourself because they rely on where the viewer is located in the chunk tree. You should ask yourself if you really need to use these functions, especially `ga_bp_to_vcp` which you might never use. We are providing `ga_bp_to_vcp` for completeness, not because it is extremely useful. On the other hand `ga_bp_to_parent_vcp` is a quite common and useful function.

For both of these functions, there needs to be at least one chunk in the active chunk tree on the same level as the chunk in question. If this is not true, the program will exit.

The function `ga_vcp_to_bp` takes the VCP representation (L, vcp) of a chunk C and returns the block position bp such that $(L-1, \text{bp})$ is the block representation of the block form of C.

The function `ga_bp_to_vcp` takes the block position representation (L, bp) of a chunk C and returns vcp such that $(L+1, \text{vcp})$ is the VCP representation of C.

17.31.7 Chunk id to parent chunk id

```
int ga_chunk_id_to_parent_chunk_id(int chunk_id);
```

This takes the chunk id of a chunk and returns the chunk id of the parent of that chunk. If the original chunk does not exist, or the parent does not exist, it will return a negative number.

17.31.8 Block position to parent block position

```
BlockPos ga_bp_to_parent_bp(int level, BlockPos bp);
```

This takes a block and returns the block representation of the chunk which contains the block.

For example, if this function is passed (L, bp_1), then it will return bp_2 such that (L-1,bp_2) is the block representation of the chunk which contains the original block.

17.31.9 Block position to parent vcp

```
ViewerCentricPos ga_bp_to_parent_vcp(BlockPos bp);
```

As said before, this is a quite common function, This takes a block and returns the VCP representation of the chunk which contains the block. **Note that this function does not need to know what level the block is on.**

For example, if this function is passed (L, bp), then it will return vcp such that (L,vcp) is the block representation of the chunk which contains the original block.

This is similar to `ga_bp_to_parent_bp`. However, unlike that function, this function is trivial to implement in Lua as a math function.

17.31.10 Block position to parent chunk id

```
int ga_bp_to_parent_chunk_id(int level, BlockPos bp);
```

Similar to previous functions, this takes in a block position and returns the chunk id of the chunk which contains the block.

17.31.11 Block position to ancestor block position

```
BlockPos ga_bp_to_ancestor_bp(
    int source_level, BlockPos source_bp, int target_level);
```

This takes the source block and returns the position of the block that contains it on level `target_level`. It must be that `target_level` $\leq$ `source_level`.

Here is an example. Consider our source block B. Let C be its parent chunk (the chunk which contains B). The chunk C is on level `source_level`. If we call this function with `target_level` = `source_level - 1`, then the function will return the block position of the block that is C.

17.31.12 Chunk to ancestor chunk

```
int ga_chunk_id_to_ancestor_chunk_id(int source_chunk_id, int target_level);
```

This takes a chunk on a level (call it the source level) and returns the ancestor chunk on level `target_level`. It must be that `target_level` is $\leq$ the source level.

This function is not as powerful as `ga_bp_to_ancestor_bp`, but you might find `ga_chunk_id_to_ancestor_chunk_id` less confusing than `ga_bp_to_ancestor_bp`.

17.31.13 Block position to path

```
string ga_bp_to_path(int level, BlockPos bp);
```

The function `ga_bp_to_path` returns the path of the chunk that occupies the specified block's position. The chunk containing bp needs to be in the active chunk tree. However the chunk occupying the same space as the block need not be in the active chunk tree.

17.31.14 Block position to lbp

```
LocalBlockPos ga_bp_to_lbp(BlockPos bp);
```

This takes a block position and returns the local block position of the block within the chunk that contains the block. This is trivial to implement in Lua as a math function.

17.31.15 Block coordinates example

You can have the following be the code for the `__on_use` function of a block script:

```
function p.__on_use(level, bp1)
    local lbp         = ga_bp_to_lbp(bp1)
    local chunk_id    = ga_bp_to_parent_chunk_id(level, bp1)
    local bp2         = ga_chunk_id_and_lbp_to_bp(chunk_id, lbp)
    local parent_vcp1 = ga_bp_to_parent_vcp(bp1)
    local bp3         = ga_lbp_to_bp(parent_vcp1, lbp)
    local parent_bp1  = ga_bp_to_parent_bp(level, bp1)
    local parent_bp2  = ga_vcp_to_bp(level, parent_vcp1)
    local parent_vcp2 = ga_bp_to_vcp(level-1, parent_bp2)
    local parent_bp3  = ga_vcp_to_bp(level, parent_vcp2)

    --bp1, 2, and 3 should be the same.
    --parent_bp1, 2, and 3 should be the same.
    --parent_vcp1 and 2 should be the same.
    ga_print("lbp = " .. std.vec_to_str(lbp))
    ga_print("bp1 = " .. std.vec_to_str(bp1))
    ga_print("bp2 = " .. std.vec_to_str(bp2))
    ga_print("bp3 = " .. std.vec_to_str(bp3))
    ga_print("parent_bp1  = " .. std.vec_to_str(parent_bp1))
    ga_print("parent_bp2  = " .. std.vec_to_str(parent_bp2))
    ga_print("parent_bp3  = " .. std.vec_to_str(parent_bp3))
    ga_print("parent_vcp1 = " .. std.vec_to_str(parent_vcp1))
    ga_print("parent_vcp2 = " .. std.vec_to_str(parent_vcp2))
end
```

17.32 Game API: Coordinates: Vectors

17.32.1 To level position

```
Vector ga_chunk_id_and_offset_to_lp(int chunk_id, Vector offset);
Vector ga_offset_to_lp(BlockPos vcp, Vector offset);
```

The first function takes in a chunk id and an offset position within that chunk and returns the level position of the point. The second function is similar

but it uses the VCP of the chunk (this second version is trivial to implement in Lua as a math function). We provide both of these shortcuts for convenience.

17.32.2 Converting from one level to another

```
CLASS ga_level_scale_factor(int source_level, int target_level);
CLASS ga_convert_lp(
    int source_level, int target_level, Vector source_lp);
```

Every point in space is on every level. The function ga_convert_lp converts the coordinates of a point (seen on level source_level) to a point on level target_level.

The function ga_level_scale_factor returns how much scaling there is from level source_level to level target_level. For example, ga_level_scale_factor(10, 11) = 16. Also, ga_level_scale_factor(10, 8) = 1/256.

These functions return a table C. It has a bool member called is_valid. If that is true, then the member "value" of C is the correct result of the query. Here is an example:

```
local viewer_level = 15
local ment_level = 13
local data = ga_level_scale_factor(viewer_level, ment_level)
if data.is_valid then
    ga_print("The level scale factor is: " .. tostring(data.value))
end
```

17.32.3 Finest chunk containing point

```
CLASS ga_finest_chunk_containing_point(int level, Vector lp);
```

This function returns the finest chunk, in the active chunk tree, that contains the specified point. In other words, it returns the chunk on the level with the greatest number that contains the point. Actually, more information is returned: it returns a table C with the following members:

- is_valid (a bool)
- level (an int)
- chunk_id (an int)
- lp (a vector)

Here lp is the level position of the point on the level returned in C.

17.33 Game API: Math

The math functions to a large extent appear in the file

`base/Game/std.lua`

However here is a game API function that is not in that Lua script:

```
CLASS ga_path_diff(string path1, string path2);
```

Use the path diff function to get a vector which points from the first path to the second. Here is an example:

```
local path_1 = "777_564"
local path_2 = "777_8a3_857"
local result = ga_path_diff(path_1, path_2)
if( result.is_valid ) then
    --The float dist is the distance from the center of the chunk C1
    --at path_1 to the center C2 of the chunk at path_2,
    --calculated on the level of C1.
    local dist = result.dist

    --The vector dir points from the center of C1
    --to the center of C2.
    local dir = result.dir
end
```

You can use this to implement a "beacon" which the player can see on their screen, guiding them towards an object in the world that is in a chunk that has not yet been loaded.

17.34 Game API: Movement and Physics

17.34.1 Setting the camera position

```
void ga_camera_set_look(Vector look, Vector up);
```

Use the function above to forcibly turn the player's head. You must always pass the "up" vector, but it is only used if the game is in 6 degrees of freedom mode ("not using true up").

Note that you can get the orientation of the camera by getting the following (system) variables:

```
game.player.camera.look
game.player.camera.up
game.player.camera.left
```

17.34.2 Moving

```
void ga_move_set_desired_travel(Vector travel);
void ga_move_set_roll(float roll);
```

The way the player moves though the world is by specifying a move (travel) vector. The engine then tries to move the player along that vector as much as possible, doing collision detection in the process. The function

`ga_move_set_desired_travel`

specifies this travel vector. So, this should be called each discrete update.

The function `ga_move_set_roll` is only used in 6 degrees of freedom games. This is used to specify how much to rotate the viewer around the viewer's look vector. Again, this should be called each discrete update.

17.34.3 Gravity

```
bool ga_move_get_on_sure_footing();
void ga_move_set_ledge_guards(bool on);
```

In games with gravity, we except that there will be more friction when the player is just above a block surface (and there will be more movement acceleration). The function ga_move_get_on_sure_footing returns true if the player is just above a block surface so that he should be considered "on the ground". Note: when jumping up a staircase, the player will be able to "catch each ledge" and move quickly forward.

17.34.4 Setting the body type

```
void ga_move_set_body_spirit();
void ga_move_set_body_spirit_off();
bool ga_move_set_body_ground(
    Vector trans, float radius, float bot_to_eye, float eye_to_top);
bool ga_move_set_body_fly(
    Vector trans, float radius, bool use_true_up);
```

There are several player body types: spirit, ground, and fly. Use these functions to set the body type. These functions may fail (due to geometry in the world), in which case the player's body will remain the same.

When the player has the *spirit* body type, he is in a chunk but does not truly interact with anything in the world. This body type is used for traversing the chunk tree to find a suitable location. For example, this can be used to create the initial starting position of the player.

When the player's body is set to be of type spirit, the engine saves the player's old body type and parameters. Then, when the player is still in spirit mode, if you call `ga_move_set_body_spirit_off`, then the player's body type

and parameters will be reset to what they were the instant before it was changed to spirit mode.

The body type *ground* is intended for games with gravity. In this body type, the player is modeled as a cylinder. The eye of the player is exactly `bot_to_eye` many units from the bottom of the cylinder, and the eye of the player is exactly `eye_to_top` many units to the top of the cylinder. When the `ga_move_set_body_ground` is first called, the eyes of the player are first translated by the vector trans.

The body type *fly* is intended for space games. In this body type, the player is modeled as a sphere. Again, the eyes of the player are first translated by trans before the new body dimensions take place. The argument `use_true_up` argument specifies whether the top middle of the player's screen points in the positive Z direction. If this is set to false, the player can easily become upside down.

17.34.5 The character model

```
void ga_player_model_set_look();
void ga_player_model_q2md2_set_cmd(string cmd);
void ga_player_model_q2md2_set_state(string state);
```

These functions modify the player model of the player. The player model is a Quake 2 character model. The function ga_player_model_set_look causes the player model to face in the direction that the player is facing.

Use the function

`ga_player_model_q2md2_set_state`

to set the *state* of the player model. The following are the accepted versions of that string: the empty string (without quotes), `run`, `crouch` and `crouch_run`. The idea is that the player model is in a certain state, such as `run`, but it can preform commands like `wave` or `jump_up` which interrupt the run state.

Use the function

`ga_player_model_q2md2_set_cmd`

when a certain action (or *command*) needs to take place. Here are the acceptable commands: the empty string (without quotes), `stand`, `run`, `attack`, `pain`, `jump_up`, `jump_down`, `flip`, `salute`, `taunt`, `wave`, `point`, `crstand`, `crwalk`, `crattack`, `crattack`, `crpain`, `crdeath1`, `death1`, `death2`, `death3`.

The player can be doing at most one command at a time. When they are finished their command, they will perform their "state" action, which will continue on a loop.

17.35 Game API: Visibility

17.35.1 ga_vis_test_level

```
bool ga_vis_test_level(int level, Vector lp_start, Vector lp_end);
```

The function returns true iff the line segment from `lp_start` to `lp_end` does not intersect any solid blocks on the given level.

17.35.2 ga_ray_cast

```
CLASS ga_ray_cast(
    int level, Vector lp_start, Vector dir,
    int min_level, int max_level);
```

Use this function to cast a ray into the world. It returns a class describing what block was hit. See the following example:

```
function p.test()
    local start_level = ga_get_viewer_level()
    local lp_start = ga_get_viewer_offset()
    local dir = ga_get_sys_v("game.player.camera.look")
    local min_level = start_level - 5
    local max_level = start_level
    local data = ga_ray_cast(
        start_level, lp_start, dir,
        min_level, max_level)
    if( not data.block_hit ) then return end
    --
    local already_inside = data.already_inside --A bool.
    if( already_inside ) then return end --The ray started inside a block.
    local level = data.block_level --The level of the block hit.
    local bp = data.block_bp --The block position of the block hit.
    local lp = data.block_lp --The level position of the block hit (a vector).
    --
    --The normal side of the block hit (an integer):
    local normal_side = data.block_normal_side
    --The normal vector of the block hit:
    local normal = data.block_normal
    --
    ga_block_change_perm(level, bp, "block_e")
    --
    ga_ment_start(level, lp, "ment_laser")
    local speed = 100.0
    local vel = std.vec_scale(normal, speed)
    ga_ment_init_set_v("__vel", vel)
    ga_ment_end()
end
```

The `data` class also has the integer member `ment_inst_id`, which is the inst id of the first ment that was hit by the ray (if any). The member `ment_inst_id` is negative if no moving entity is hit. However, the ray collision stops when the ray hits a block.

You should be careful how many `ga_ray_cast` functions you call, because they are computationally expensive. Note that you can simply use the functions `ga_look_object_block_XXX` if you want to cast a ray in the direction that the player is looking (the engine casts such a ray each frame and you can query the intersection via these "look object" API functions).

17.36 Game API: Rendering

Render functions in the Game API can be called at certain times, such as in the `__render_augmented` functions of game scripts and the `__render` functions of bent and ment scripts.

Fractal Block World uses OpenGL for rendering. An important concept of that library is the "matrix stack". At any given point we can take the matrix M on the top of the stack. This is a 4x4 matrix which we can multiply on the right by a (column) vector v to get Mv. Here are functions which modify the program's matrix stack:

```
void ga_render_skip_next_frame();
void ga_render_push_matrix();
void ga_render_pop_matrix();
void ga_render_matrix_load_identity();
void ga_render_matrix_row_major(
    float m11, float m12, float m13, float m14,
    float m21, float m22, float m23, float m24,
    float m31, float m32, float m33, float m34,
    float m41, float m42, float m43, float m44);
void ga_render_matrix_translated(float trans_x, float trans_y, float trans_z);
void ga_render_matrix_scaled(float scale_x, float scale_y, float scale_z);
void ga_render_matrix_rotated(
    float angle,
    Vector axis);
void ga_render_matrix_frame(
    Vector look,
    Vector up,
    Vector left);
void ga_render_matrix_frame_from_ment(int inst_id);
```

Call the function `ga_render_skip_next_frame` to prevent the rendering of the next frame. However, normal processing will occur (other than rendering) in the next frame.

Let M be the matrix on the top of the matrix stack. The function

`ga_render_push_matrix`

pushes a copy of M to the top of the stack. That is, if before the stack had size n with M on top, then after the push the stack has size $n+1$ and the top two matrices are both M.

The function `ga_render_pop_matrix` pops the matrix on the top of the matrix stack (it removes it). So if before the stack had size n, after the pop it will have size $n-1$.

The function `ga_render_matrix_load_identity` replaces the matrix on the top of the matrix stack with the identity matrix.

The function `ga_render_matrix_row_major` takes the matrix N on the top of the matrix stack and multiplies it by the matrix M specified by the arguments of the function to get $N * M$. The arguments of the function determine M as follows:

$$M = \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}.$$

That is, the elements of M are specified in "row major order".

The function `ga_render_matrix_translated` corresponds to the OpenGL function glTranslated. It takes the matrix on the top of the matrix stack and multiplies it by the specified translation matrix.

The function `ga_render_matrix_scaled` corresponds to the OpenGL function glScaled. It takes the matrix on the top of the matrix stack and multiplies it by the specified scaling matrix.

The function `ga_render_matrix_rotated` takes the matrix on top of the matrix stack and multiplies it by a rotation matrix, specified by the given angle and axis.

The function `ga_render_matrix_frame` takes the matrix on the top of the matrix stack and multiples it by a certain matrix M. The matrix M is such that multiplication by it corresponds to the transformation which maps (1,0,0) to look, (0,1,0) to -left, and (0,0,1) to up. You can use this function to render a weapon that the player is holding. That is, you get the look, left, and up vectors of the camera and use those.

Every moving entity has its own "frame", which specifies its orientation. A frame is a triple of unit vectors which are perpendicular to each other. The function `ga_render_matrix_frame_from_ment` takes the matrix on the top of the matrix stack and multiplies it by the matrix which rotates to the orientation of that frame. It is intended that you use this in the `__render` function of a moving entity.

```
void ga_render_ment_typical(int inst_id);

void ga_render_mesh(string mesh_name);
void ga_render_mesh_with_tex(
    string mesh_name,
    string tex_name);
void ga_render_mesh_with_tex_alpha(
```

```
    string mesh_name,
    string tex_name,
    float alpha);
void ga_render_mesh_with_tex_no_lighting(
    string mesh_name,
    string tex_name);
void ga_render_mesh_with_tex_alpha_no_lighting(
    string mesh_name,
    string tex_name,
    float alpha);

void ga_render_line(
    Vector v1,
    Vector v2);
void ga_render_line_thick(
    Vector v1,
    Vector v2,
    float thickness);
void ga_render_triangle(
    Vector v1,
    Vector v2,
    Vector v3,
    float u1, float v1,
    float u2, float v2,
    float u3, float v3,
    string tex);
Vector ga_render_get_color();
void ga_render_color(
    Vector color);
```

The function `ga_render_ment_typical` renders an ment in the usual way (the way hardcoded by the engine). It is intended to be called inside an `__on_render` function of the ment. That is, before the `__on_render` function of an ment is called, a translation matrix is pushed onto the matrix stack. Then if `ga_render_ment_typical` is called inside that function, then that translation will not be applied again.

The function `ga_render_mesh` renders the mesh of the given name (using the default texture associated to that mesh).

The function `ga_render_mesh_with_tex` renders the mesh with the given name but using the specified texture as an override. The functions

`ga_render_mesh_with_tex_alpha`

and

`ga_render_mesh_with_tex_alpha_no_lighting`

are similar but they apply the specified alpha value to the mesh.

Recall that the user can choose to apply directional shading to meshes. However, it is possible to render meshes ignoring this lighting (they are fully lit). One such function is `ga_render_mesh_with_tex_no_lighting`.

The function `ga_render_line` renders a line in the world from vector $v1$ to the vector $v2$. The function ga_render_color can be called before this (or a sequence of these calls) to specify the color.

The function `ga_render_line_thick` is just like ga_render_line but you can specify the thickness of the line. Here line thickness is in terms of how OpenGL defines it.

The function `ga_render_triangle` renders a (textured) triangle with the specified texture coordiates and texture name. Note that you can call the function `ga_render_color` before this function to set the shading color.

The function `ga_render_color` can be called before certain render functions to specify the color. The function `ga_render_get_color` gets the current color that the system is using for rendering.

```
void ga_render_set_depth_test_enabled(bool value);
```

This turns on/off the depth test for rendering. When the depth test is enabled, the depth buffer is used.

```
void ga_render_clear_depth_buffer();
```

The function `ga_render_clear_depth_buffer` clears the OpenGL depth buffer. You probably do not want to call this function.

17.37 Game API: Windows (Part 2)

These functions are described in Chapter 18.

17.38 Game API: Rebooting the Game

```
string ga_reboot_dyn_itr_get()
void ga_reboot_dyn_itr_next()
bool ga_reboot_dyn_itr_at_end()
void ga_reboot_dyn_itr_save()
```

At any time the player can choose to "reboot" their saved game. This will delete all chunk files in that save game directory. It will keep all global vars stored in `env_vars.txt` in the same gave directory. What happens to the (dynamic) variables in `dyn_vars.txt` is more complicated.

By default, all dynamic variables will be deleted. However, the package can choose to save them one at a time. The functions described here iterate over all dynamic variables, and the function `ga_reboot_dyn_itr_save` marks the current var that we have iterated to to be saved.

This is all accomplished within the function `top.__reboot_game` in the file top.lua. This function is called by the engine when the game is rebooted, just before `top.__new_game` is called. The function `top.__reboot_game` will be called over and over again until it returns true.

Here is an example of what you might want the function `top.__reboot_game` to look like. This function only saves the variable called `dyn.test.my_favorite_book`.

```
--This is in the file "top.lua".
--Returns false when done.
function p.__reboot_game()
    ga_print("Here in top.__reboot_game")

    --Only checking ten thousand vars
    --during this function call.
    local countdown = 10000 --10 thousand.
    while true do
        if ga_reboot_dyn_itr_at_end() then
            --Done the rebooting process.
            return false --Done.
        end

        countdown = countdown - 1
        if( countdown <= 0 ) then
            --Too many vars to do now,
            --must do them later.
            return true --Not done yet.
        end

        local var = ga_reboot_dyn_itr_get()

        --Only saving the var called
        --"dyn.test.my_favorite_book".
        if var == "dyn.test.my_favorite_book" then
            ga_reboot_dyn_itr_save()
        end

        ga_reboot_dyn_itr_next()
    end

    --Will never reach here.
    return false --Done.
end
```

17.39 Game API: File IO

```
int    ga_open_file_for_writing(string file_name);
```

```
int    ga_open_named_pipe(string name);
void   ga_write(int handle, string str);
string ga_read(int handle);
void   ga_close_file(int handle);
```

You can create text files and write to them. However there are limitations due to security. The file name must be of the form X.txt or X.lua, where X is a non-empty string that only contains letters (capital and lowercase), numbers, and underscores.

Suppose the file name is `foo.txt`. The file will be created in the location `Output/FileOut/foo.txt` relative to the root directory of the program.

The function `ga_open_file_for_writing` opens the file (with the given name) for writing. It returns an integer *handle* which is what you use to refer to the file.

The function `ga_write` actually writes to the file. It appends the string str to the end (but note that the function `ga_open_file_for_writing` clears the file). The function `ga_write` does NOT insert extra newline characters.

The function `ga_close_file` closes the file.

The function `ga_open_named_pipe` opens a "named pipe" and returns an integer handle which you use to refer to that pipe. You can read and write from the pipe using `ga_read` and `ga_write`. See the guide about named pipes on the game's website for how to use named pipes.

Here is an example:

```
local handle = ga_open_file_for_writing("favorite_colors.txt")
ga_write(handle, "green\n")
ga_write(handle, "red\n")
ga_close_file(handle)
```

17.40 Game API: Accessibility

```
bool   ga_get_is_colorblind();
Vector ga_get_colorblind_closest(Vector color);
Vector ga_get_colorblind_bynum(int num);
void   ga_set_colorblind_bynum(int num, Vector color);
```

The way colorblind support works is the following: the player who is colorblind creates a small palate of colors (say 10 to 20) such that they can distinguist any two of the colors. Then, as the developer of your package, you call `ga_get_is_colorblind` and if this returns true, then as you see fit, when you are going to render a certain color you replace that color with the closest color in the pallete.

The function `ga_get_is_colorblind` returns whether the player is requesting colorblind mode. Note that a player might want to do this even if they are not colorblind.

The function `ga_get_colorblind_closest` returns the "closest color" in "the palette" to the one specified.

The function `ga_set_colorblind_bynum(n, color)` specifies the n-th color in the palette. The player who is colorblind would call this function. There is a catch: if there are n colors in the palette, then color n+1 must be set to the following invalid color:

`std.vec(-1.0, -1.0, -1.0)`.

This is how the system knows the palette is over.

Let us show how a player who is colorbind could set the palette. They can modify the file `Input/Scripts/game_startup.lua` and make it look something like this:

```
function p.__main() --Called every time a game is loaded.
    p.color_blind_init() --Defined below.
    -- ...
end

function p.color_blind_init()
    --First we enable color blind mode.
    --This will cause the function ga_get_is_colorblind
    --to return true.
    ga_command("set menu.colorblind.enable true")

    --Setting the palette.
    p.color_blind_init_helper(1, "^xcddde3")
    p.color_blind_init_helper(2, "^xf7ee00")
    p.color_blind_init_helper(3, "^x0021e1")
    p.color_blind_init_helper(4, "^x36db00")
    p.color_blind_init_helper(5, "^xed00ff")
    p.color_blind_init_helper(6, "^x0080ff")
    p.color_blind_init_helper(7, "^xff4242")
    p.color_blind_init_helper(8, "^x00c2ff")
    p.color_blind_init_helper(9, "^x7b0470")
    p.color_blind_init_helper(10, "^xff4692")
    p.color_blind_init_helper(11, "INVALID")
end

function p.color_blind_init_helper(i, code)
    if( code == "INVALID" ) then
        local invalid_color = std.vec(-1.0, -1.0, -1.0)
        ga_set_colorblind_bynum(i, invalid_color)
        return
    end
    local color = ga_color_code_to_vec(code)
    ga_set_colorblind_bynum(i, color)
```

```
end
```

If people in the community what their palettes to be included with the game, please send us your palettes and a description of the type of colorblindness the palette applies to. We will accept image files containing colored rectangles or the pallets to be given by code like that shown above.

17.41 Game API: Text and Strings

```
Vector ga_color_code_to_vec(string code);
string ga_color_vec_to_code(Vector color);
string ga_txt_strip_esc_seq(string input);
```

The function `ga_color_code_to_vec` takes in a color code string and returns a color in the form of a vector (x = red, y = green, z = blue). Here x,y,z are between 0 and 1. For example, if you pass `^x00ff00` to the function it will return the vector (0.0, 1.0, 0.0).

The function `ga_color_vec_to_code` works in the opposite direction. For example, if you pass it std.vec(0.0, 1.0, 0.0) it will return `^x00ff00`.

The function `ga_txt_strip_esc_seq` takes the input string and removes all color escape sequence codes from it. For example, if you pass the function the string

"`I like the color ^x00ff00RED^! the best`",

it will return the string

"`I like the color RED the best`".

17.42 Game API: Windows Clipboard

```
void ga_copy_to_clipboard(string str);
string ga_paste_from_clipboard();
```

Use these to read and write a string to and from the Windows clipboard.

Chapter 18

The Game Lua-to-C API: Windows

In Chapter 17 we talked about most of the Game Lua-to-C API. In this chapter we will discuss more of this API. Specifically, we will discuss functions that are intended to be called from Window Lua Scripts.

18.1 The API

```
//-----------------------------------------------
//                Window Managing
//-----------------------------------------------
string ga_win_wid_to_win_name(int wid);
int ga_win_win_name_to_wid(string win_name);
LIST ga_win_get_windows_on_hud();
LIST ga_win_get_windows_on_stack();
string ga_win_get_stack_top();
bool ga_win_is_any_window_open();

bool ga_win_is_window_on_hud(string win_name);
bool ga_win_is_window_on_stack(string win_name);
bool ga_win_is_window_on_stack_top(string win_name);

LIST ga_win_get_windows_on_main_menu_stack(); //Deprecated.
LIST ga_win_get_windows_on_game_stack(); //Deprecated.
string ga_win_get_main_menu_stack_top();  //Deprecated.
string ga_win_get_game_stack_top(); //Deprecated.

//-----------------------------------------------
//             Screen Coordinate Modes
//-----------------------------------------------
```

```
string ga_win_get_screen_coord_mode(int wid);
void   ga_win_set_screen_coord_mode(int wid, string mode);
Vector ga_win_inner_coord_to_screen_coord(Vector p);
Vector ga_win_screen_coord_to_inner_coord(Vector p);

//-----------------------------------------------
//                  Rendering
//-----------------------------------------------
void ga_win_set_back_params(
    int wid, Vector color, float alpha1, float alpha2);
void   ga_win_set_front_color(int wid, Vector color);
void   ga_win_set_front_color_default(int wid);
Vector ga_win_get_default_front_color();
void   ga_win_set_char_size(int wid, float char_width, float char_height);
void   ga_win_set_background(int wid, Vector color, float alpha);
void   ga_win_set_background_default(int wid);

void ga_win_clear_depth_buffer_maybe();

void ga_win_line(
    int wid, float x1, float y1, float x2, float y2,
    Vector color);
void ga_win_line_rect(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color);

void ga_win_quad(
    int wid, float min_x, float min_y, float max_x, float max_y,
    string tex);
void ga_win_quad_two(
    int wid, float min_x, float min_y, float max_x, float max_y,
    string tex1 string tex2, float frac);
void ga_win_quad_color(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color);
void ga_win_quad_color_alpha(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color, float alpha);

CLASS ga_win_txt(
    int wid, float min_x, float min_y, string txt);
CLASS ga_win_txt_alpha_bg(
    int wid, float min_x, float max_x, float alpha, string txt);
CLASS ga_win_txt_center(
    int wid, float min_y, string txt);
```

```
CLASS ga_win_txt_center_at_bg(
    int wid, float center_x, float min_y, string txt);

void ga_win_txt_box(
    int wid, string txt, bool go_back_msg);

float ga_win_get_center_text_min_x(float w, int num_chars);

//-----------------------------------------------
//                    Widgets
//-----------------------------------------------
void ga_win_widget_go_back_button_start(
    int wid, float x, float w, float h, string msg);
bool ga_win_widget_go_back_button_process_input(int wid);

void   ga_win_widget_button_start(
    int wid, int handle, float x, float y, float w, float h, string msg);
void   ga_win_widget_button_center_x_at(int wid, int handle, float x);
void   ga_win_widget_button_set_color(int wid, int handle, Vector color);
string ga_win_widget_button_get_text(int wid, int handle);
void   ga_win_widget_button_set_text(int wid, int handle, string msg);
int    ga_win_widget_button_process_input(int wid);
void   ga_win_widget_button_end(int wid, int handle);

void ga_win_widget_small_list_start(
    int wid, float min_y, float max_y,
    float char_width, float char_height,
    Vector txt_color, LIST items);
void   ga_win_widget_small_list_set_use_nums(bool value);
int    ga_win_widget_small_list_process_input(int wid);
string ga_win_widget_small_list_get_entry(int wid, int index);
int    ga_win_widget_small_list_get_selected(int wid);
void   ga_win_widget_small_list_set_selected(int wid, int sel_num);
void   ga_win_widget_small_list_set_selected_by_substr(int wid, string substr);
void   ga_win_widget_small_list_mute_sound(int wid, string which_sound);
void   ga_win_widget_small_list_end(int wid);

void ga_win_widget_large_list_start(
    int wid, float min_y, float max_y,
    float char_w, float char_h, Vector text_color, LIST items);
void   ga_win_widget_large_list_enable_scroll_bar(int wid, float x);
void   ga_win_widget_large_list_set_scroll_y_min_max(int wid, float min_y, float max_y);
int    ga_win_widget_large_list_process_input(int wid);
string ga_win_widget_large_list_get_entry(int wid, int index);
int    ga_win_widget_large_list_get_selected(int wid);
void   ga_win_widget_large_list_set_selected(int wid, string str);
```

```
void   ga_win_widget_large_list_set_selected_by_num(int wid, int num);
void   ga_win_widget_large_list_set_selected_by_substr(int wid, string substr);

void ga_win_widget_text_input_start(
    int wid, float min_y, float char_width, float chat_height);
string ga_win_widget_text_input_process_input(int wid);
string ga_win_widget_text_input_get_text(int wid);
void   ga_win_widget_text_input_set_text(int wid, string str);
string ga_win_widget_text_input_set_enable_enter(int wid, bool value);
void   ga_win_widget_text_input_forbid_all_chars(int wid);
void   ga_win_widget_text_input_set_char_allowed(
    int wid, string char_str, bool value);

void ga_win_widget_mutable_text_box_start(
    int wid, float min_x, float max_x, float min_y, float max_y,
    float char_width, float char_height,
    Vector txt_color, string init_str);
string ga_win_widget_mutable_text_box_get_text(int wid);
void   ga_win_widget_mutable_text_box_set_text(int wid, string str);
void   ga_win_widget_mutable_text_box_end(int wid);

//-----------------------------------------------
//                    Cursor
//-----------------------------------------------
bool ga_win_get_cursor_enabled();
void ga_win_enable_cursor(bool value);
void ga_win_show_cursor_icon(bool value);
void ga_win_enable_hud_cursor(bool value);
Vector ga_win_get_cursor_pos(int wid);
Vector ga_win_get_cursor_diff(int wid);
void   ga_win_scroll(int wid, float scroll_x, float scroll_y);
Vector ga_win_m_to_s(int wid, float x, float y);
Vector ga_win_s_to_m(int wid, float x, float y);
void   ga_win_set_scroll_bounds(
    int wid, float min_x, float min_y, float max_x, float max_y);

//-----------------------------------------------
//               Input Without WID
//-----------------------------------------------
bool ga_win_key_down(string key);
bool ga_win_mouse_down(bool left);
bool ga_win_mouse_down2(int button);

//-----------------------------------------------
//               Input With WID
//-----------------------------------------------
```

```
bool ga_win_key_pressed(int wid, string key);
bool ga_win_key_pressed_or_spammed(
    int wid, string key, float init_wait, float subsequent_wait);
bool ga_win_key_released(int wid, string key);
bool ga_win_mouse_pressed(int wid, bool left);
bool ga_win_mouse_pressed2(int wid, int button);
bool ga_win_mouse_released(int wid, bool left);
bool ga_win_mouse_released2(int wid, int button);
bool ga_win_mouse_wheel_up(int wid);
bool ga_win_mouse_wheel_down(int wid);

//-----------------------------------------------
//              Other Input Related
//-----------------------------------------------
void ga_win_release_keys_that_are_down();
```

18.2 The Window ID (WID)

(Almost) all of these API functions take the window ID (WID) of the current window. Every window script `win_script.lua` is associated to exactly one WID. The official "name" of that would would be "`win_script`".

18.3 Window Management

```
string ga_win_wid_to_win_name(int wid);
int ga_win_win_name_to_wid(string win_name);
```

Use these to convert back and forth between the name of a window (the name of the window script without the ".lua") and the WID (window id).

```
LIST ga_win_get_windows_on_hud();
LIST ga_win_get_windows_on_stack();
```

Use these functions to get an array of the windows that are on the hud and the window stack. Each element of the returned array is a table, with a member called "name", which is the name of the window. Here is an example:

```
local array = ga_win_get_windows_on_hud()
for i = 1,#array do
    ga_console_print(array[i].name)
end
```

```
string ga_win_get_stack_top();
```

Use the function above to get the name of the window on top of the window stack.

```
bool ga_win_is_any_window_open();
```

The above function is a helper function that returns true iff any one of the following conditions is true:

- The window stack is non-empty,
- The player is in the main menu (the escape key menu), or
- The console is open.

```
bool ga_win_is_window_on_hud(string win_name);
bool ga_win_is_window_on_stack(string win_name);
bool ga_win_is_window_on_stack_top(string win_name);
```

The above functions are helper functions, which you could write yourself. The first function returns whether or not the given window is on the hud. The second function is similar, but returns whether the window is in the window stack (not neccessarily at the top). The third function returns whether or not the window is on top of the window stack.

18.4 Deprecated Window Management

```
LIST ga_win_get_windows_on_main_menu_stack();
LIST ga_win_get_windows_on_game_stack();
string ga_win_get_main_menu_stack_top();
string ga_win_get_game_stack_top();
```

Do not use these functions. The main menu stack has been merged with the game window stack.

18.5 Screen Coordinate Modes

```
string ga_win_get_screen_coord_mode(int wid);
void   ga_win_set_screen_coord_mode(int wid, string mode);
Vector ga_win_inner_coord_to_screen_coord(Vector p);
Vector ga_win_screen_coord_to_inner_coord(Vector p);
```

There are two screen coordinate modes: “screen” and “inner”. You can get and set this mode with the functions `ga_win_get_screen_coord_mode` and `ga_win_set_screen_coord_mode`, however “inner” mode will be set before and after the engine calls any Lua window rendering function. In “screen” mode, the entire screen goes from (0,0) to (1,1). In “inner” mode, points in the range (0,0) and (1,1) will be mapped into the “inner rectangle”. The inner rectangle has the same height as the screen but its width is capped so that it never has an aspect ratio that is too large. So when you play the game the HUD will not be too distorted on a wide monitor.

You can use the function `ga_win_inner_coord_to_screen_coord` to convert a point in "inner" mode into a vector in "screen" mode. The function `ga_win_screen_coord_to_inner_coord` does the opposite.

18.6 Setting Foreground and Background Params

```
void ga_win_set_back_params(
    int wid, Vector color, float alpha1, float alpha2);
void   ga_win_set_front_color(int wid, Vector color);
void   ga_win_set_front_color_default(int wid);
Vector ga_win_get_default_front_color();
void   ga_win_set_char_size(int wid, float char_width, float char_height);
void   ga_win_set_background(int wid, Vector color, float alpha);
void   ga_win_set_background_default(int wid);
```

The function `ga_win_set_back_params` sets various parameters related to the background. This is used to render behind text, for example. The argument alpha2 should be more opaque than alpha1.

The function `ga_win_set_front_color` sets the "front color", which is used as the color of text for example. The function `ga_win_set_front_color_default` sets the front color to the default value (the value stored in the environment variable "`menu.text_color`"). The function `ga_win_get_default_front_color` simply returns the value of `menu.text_color`.

The function `ga_win_set_char_size` sets the character width and height of text. A width of 1.0 means it is the width of the entire screen, and a height of 1.0 means it is a height of the entire screen.

The function `ga_win_set_background` sets the background color and alpha. The function `ga_win_set_background_default` sets the background to its default color and alpha.

Here is an example of how these functions can be used:

```
function p.render(wid)
    ga_win_set_front_color(wid, std.vec(1.0, 1.0, 1.0))
    ga_win_set_back_params(wid, std.vec(0.0, 0.0, 0.0), 0.1, 0.3)
    ga_win_set_background(wid, std.vec(0.0, 0.0, 0.0), 0.2);
end
```

18.7 Depth Buffer

```
void ga_win_clear_depth_buffer_maybe();
```

Normally when you render elements on a window, the depth buffer is not enabled. However, you can turn it on using `ga_render_set_depth_test_enabled` and if you do that, you might want to clear the depth buffer before doing any

window rendering. The engine does not automatically clear the depth buffer before rendering a window, but the function `ga_win_clear_depth_buffer_maybe` will clear the depth buffer *if you have not yet called that function this frame.*

18.8 Screen Shapes

```
void ga_win_line(int wid, float x1, float y1, float x2, float y2,
    Vector color);
```

Use this for drawing a line on the screen.

```
void ga_win_line_rect(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color);
```

Use this to draw a rectangle made out of 4 lines.

```
void ga_win_quad(
    int wid, float min_x, float min_y, float max_x, float max_y,
    string tex);
void ga_win_quad_two(
    int wid, float min_x, float min_y, float max_x, float max_y,
    string tex1 string tex2, float frac);
void ga_win_quad_color(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color);
void ga_win_quad_color_alpha(
    int wid, float min_x, float min_y, float max_x, float max_y,
    Vector color, float alpha);
```

The function `ga_win_quad` pastes a quad on the screen. 0.0 is the left of the screen and 1.0 is the right. 0.0 is the bottom of the scree and 1.0 is the top. This function takes a texture (string), and the texture will be displayed as a rectangle on the screen.

The function `ga_win_quad_two` is just like `ga_win_quad` except the bottom half of the quad will have texture tex1 and the top half will have texture tex2. The number frac determines how much is tex1 verses tex2. If frac is 0.0, then the quad will be entirely tex2. If frac is 1.0, then the quad will be entirely tex1.

The function `ga_win_quad_color` is just like `ga_win_quad` except instead of drawing a textured quad, it draws a quad that is just one solid color.

The function `ga_win_quad_color_alpha` is similar to `ga_win_quad_color` but it also takes an alpha component.

18.9 Screen Text

```
CLASS ga_win_txt(
```

```
    int wid, float min_x, float min_y, string txt);
CLASS ga_win_txt_alpha_bg(
    int wid, float min_x, float max_x, float alpha, string txt);
CLASS ga_win_txt_center(
    int wid, float min_y, string txt);
CLASS ga_win_txt_center_at_bg(
    int wid, float center_x, float min_y, string txt);
```

The function `ga_win_txt` puts text on the screen. The lower left hand corner of the text is at the position (`min_x`, `min_y`). The character width and height is set by the function `ga_win_set_char_size`.

The function `ga_win_txt_alpha_bg` is just like `ga_win_txt` except that it also places a background directly behind the text being drawn. The alpha is for the text itself. The background color and alpha2 will be used to make a quad behind the text being drawn.

The function `ga_win_txt_center` puts text whose x component is in the center of the screen. The minimum y value of the text is given by `min_y`.

While the function `ga_win_txt_center` puts text whose x component is in the center of the screen, the function `ga_win_txt_center_at_bg` puts text whose x center is given by the `center_x` variable. Also, this function draws a background behind the text in an analogous way that `ga_win_txt_alpha_bg` does.

All of these text drawing functions return a table with the members `min` and `max` which are both vectors (with `x`, `y`, `z` components). Those vectors hold the *screen coordinates* for the smallest rectangle that contains the text drawn.

18.10 Text Box

```
void ga_win_txt_box(
    int wid, string txt, bool go_back_msg);
```

The function will render a "text box" in the center of the screen with the given text. If `ga_back_msg` is true, then at the bottom of the screen there will be a message asking of the player would like to "go back" by pressing either Escape or F.

18.11 Screen Coordinate Calculators

```
float ga_win_get_center_text_min_x(float w, int num_chars);
```

The function `ga_win_txt_box` is silly and might be deprecated at some point. It simply returns

$$0.5 - 0.5 * (\mathrm{w} * \mathrm{num\_chars}).$$

18.12 "Go Back" Button Widget

```
void ga_win_widget_go_back_button_start(
    int wid, float x, float w, float h, string msg);
bool ga_win_widget_go_back_button_process_input(int wid);
```

A "go back" button can be implemented using a normal button. However go back buttons are intended to be easy to use. These buttons are always centered on the screen (horizontally).

Using the `ga_win_widget_go_back_button_start` function, you just need to specify the y coordinate on the screen, the text character width and height, and the text message to appear on the button.

The function `ga_win_widget_go_back_button_process_input` will return true iff the player either presses the escape key or clicks on the button (assuming the user has enabled their cursor). This function should be called in the `__process_input` function of the window which owns the go back button.

18.13 Button Widget

```
void   ga_win_widget_button_start(
    int wid, int handle, float x, float y, float w, float h, string msg);
void   ga_win_widget_button_center_x_at(int wid, int handle, float x);
void   ga_win_widget_button_set_color(int wid, int handle, Vector color);
string ga_win_widget_button_get_text(int wid, int handle);
void   ga_win_widget_button_set_text(int wid, int handle, string msg);
int    ga_win_widget_button_process_input(int wid);
void   ga_win_widget_button_end(int wid, int handle);
```

Buttons have text on them. The user can click on buttons (if they have enabled their cursor). You as the modder refer to a button by its integer "handle". When you create a button, you specify what its handle is.

The `ga_win_widget_button_start` creates a button (with the specified handle, lower left x and y screen coordinates, text character width and height, and text message).

You can call `ga_win_widget_button_center_x_at` to center the button's x coordinate.

You can call `ga_win_widget_button_set_color` to set the color of the button.

You can call the get text and set text functions to get and set the text that is displayed on the button.

The function `ga_win_widget_button_process_input` returns the integer handle of the button that has been clicked (if there is any).

The function `ga_win_widget_button_end` destroys the specified button.

18.14 Small List Widget

```
void ga_win_widget_small_list_start(
    int wid, float min_y, float max_y,
    float char_width, float char_height,
    Vector txt_color, LIST);
void   ga_win_widget_small_list_set_use_nums(bool value);
int    ga_win_widget_small_list_process_input(int wid);
string ga_win_widget_small_list_get_entry(int wid, int index);
int    ga_win_widget_small_list_get_selected(int wid);
void   ga_win_widget_small_list_set_selected(int wid, int sel_num);
void   ga_win_widget_small_list_set_selected_by_substr(int wid, string substr);
void   ga_win_widget_small_list_mute_sound(int wid, string which_sound);
void   ga_win_widget_small_list_end(int wid);
```

A small list widget is a list of options that the player can choose from. These are presented on the screen. All options show up on the screen (none are hidden, and no scrolling is required).

The function `ga_win_widget_small_list_start` function creates a small list widget, and should probably be called in the `on_start` function of a window lua script. Here is an example (in a window script):

```
function p.on_start(wid)
    local min_y = 0.3
    local max_y = 0.7
    local char_w = 0.03
    local char_h = 0.06
    local color = {x=0.0, y=0.5, z=0.5}
    local options = {
        "NEW GAME",
        "LOAD GAME",
        "SAVE GAME",
        "PLAY TETRIS",
        "EXIT"}
    ga_win_widget_small_list_start(
        wid, min_y, max_y, char_w, char_h,
        color, options)
```

The function `ga_win_widget_small_list_set_use_nums` specifies whether the user can press number keys to quickly select an option.

The function `ga_win_widget_small_list_process_input` allows the widget to process input. The function returns a positive integer if and only if an item has been selected from the list. For example, continuing our example from above, if this function returns 2, then the selected option is "LOAD GAME".

Every entry in the list is given a number. The first entry is given number 1, the next is given number 2, etc. The function `ga_win_widget_small_list_get_entry` returns the name of the entry with the given number.

The function `ga_win_widget_small_list_get_selected` returns which item is selected. The indexing starts at zero.

The function `ga_win_widget_small_list_set_selected` sets which item is selected. The indexing starts at zero.

The function `ga_win_widget_small_list_set_selected_by_substr` selects the first item whose text has the given substring.

The function `ga_win_widget_small_list_mute_sound` makes it so a given sound is not played. Right now, the only valid option for the `which_sound` string argument is "select".

The function `ga_win_widget_small_list_end` deletes the small selection list widget.

18.15 Large List Widget

```
void ga_win_widget_large_list_start(
    int wid, float min_y, float max_y,
    float char_w, float char_h, Vector text_color, LIST items);
void   ga_win_widget_large_list_enable_scroll_bar(int wid, float x);
void   ga_win_widget_large_list_set_scroll_y_min_max(
    int wid, float min_y, float max_y);
int    ga_win_widget_large_list_process_input(int wid);
string ga_win_widget_large_list_get_entry(int wid, int index);
int    ga_win_widget_large_list_get_selected(int wid);
void   ga_win_widget_large_list_set_selected(int wid, string str);
void   ga_win_widget_large_list_set_selected_by_num(int wid, int num);
void   ga_win_widget_large_list_set_selected_by_substr(int wid, string substr);
```

The function `ga_win_widget_large_list_start` is similar to the one for small selection lists.

The function `ga_win_widget_large_list_enable_scroll_bar` causes a scroll bar to appear that the user can use with their mouse. That function takes the screen x coordinate of the scroll bar. Once this has been called, the function `ga_win_widget_large_list_set_scroll_y_min_max` can be called to set the min and max y screen value of the bar.

The function `ga_win_widget_large_list_process_input` causes the engine to process user input. The function returns which item has been selected (and returns a negative number if nothing was selected).

The function `ga_win_widget_large_list_get_entry` gets the string text of the specified entry. The indexing starts at zero.

The function `ga_win_widget_large_list_set_selected` selects the specified item. That is, it selects the first item whose text matches the given string.

The function `ga_win_widget_large_list_set_selected_by_num(wid, n)` selects the n-th item (the first item is the 0-th item).

The function `ga_win_widget_large_list_set_selected_by_substr` selects the first entry whose text has the specified substring.

18.16 Text Input Widget

```
void ga_win_widget_text_input_start(
    int wid, float min_y, float char_width, float chat_height);
string ga_win_widget_text_input_process_input(int wid);
string ga_win_widget_text_input_get_text(int wid);
void   ga_win_widget_text_input_set_text(int wid, string str);
string ga_win_widget_text_input_set_enable_enter(int wid, bool value);
void   ga_win_widget_text_input_forbid_all_chars(int wid);
void   ga_win_widget_text_input_set_char_allowed(
    int wid, string char_str, bool value);
```

The text input widget is a simple widget for the user to enter a line of text.

The function `ga_win_widget_text_input_start` creates the text input widget. Note that the `min_y` argument specifies the minimum y value of the text input widget.

The function `ga_win_widget_text_input_process_input` processes all keyboard input to the widget. If this function returns a non-empty string, then that is the string that was inputted (the user typed something and then pressed enter).

Use the get text and set text functions to get and set the current contents of the text input widget (as a string).

The function `ga_win_widget_text_input_set_enable_enter` specifies whether pressing the enter key causes the "widget to flush" and have the text replaced with the empty string. This is enabled by default.

The function `ga_win_widget_text_input_forbid_all_chars` makes it so no characteres can be entered.

After you call `ga_win_widget_text_input_forbid_all_chars`, you can call `ga_win_widget_text_input_set_char_allowed` to allow individual characters.

18.17 Mutable Text Box Widget

```
void   ga_win_widget_mutable_text_box_start(
    int wid, float min_x, float max_x, float min_y, float max_y,
    float char_width, float char_height,
    Vector txt_color, string init_str);
string ga_win_widget_mutable_text_box_get_text(int wid);
void   ga_win_widget_mutable_text_box_set_text(int wid, string str);
void   ga_win_widget_mutable_text_box_end(int wid);
```

A mutable text box is like a normal text box except the user can modify the text.

The function `ga_win_widget_mutable_text_box_start` creates the mutable text box widget (for the given window). An initial string is specified.

The functions `ga_win_widget_mutable_text_box_get_text` gets the text string for the mutable text box, and the function `ga_win_widget_mutable_text_box_set_text` sets the string.

The function `ga_win_widget_mutable_text_box_end` destroys the mutable text box widget of the given window. You can put this in the `on_end` function of the associated window script (but you do not have to).

18.18 Cursor and Map Coordinates

```
bool ga_win_get_cursor_enabled();
void ga_win_enable_cursor(bool value);
void ga_win_show_cursor_icon(bool value);
void ga_win_enable_hud_cursor(bool value);
Vector ga_win_get_cursor_pos(int wid);
Vector ga_win_get_cursor_diff(int wid);
void ga_win_scroll(int wid, float scroll_x, float scroll_y);
Vector ga_win_m_to_s(int wid, float x, float y);
Vector ga_win_s_to_m(int wid, float x, float y);
void ga_win_set_scroll_bounds(
    int wid, float min_x, float min_y, float max_x, float max_y);
```

Recall that (0.0, 0.0) is the lower left hand corner of the screen and (1.0, 1.0) is the upper right hand corner.

The function `ga_win_get_cursor_enabled` returns whether or not the user has enabled the cursor.

The function `ga_win_enable_cursor` sets whether or not the cursor is enabled. This only has effect for the current frame.

The function `ga_win_show_cursor_icon` specifies whether the cursor *icon* should be shown. You could call this function passing the value false if you want to render a custom icon for the cursor. Note that an alternate way to do this is to override the cursor texture.

Even if the user enables their cursor, the cursor normally only shows up in the engine's main menu, for windows on the main menu stack, and for windows the game stack. However, if you call `ga_win_enable_hud_cursor` passing the value true, then the cursor will be shown when the player is playing the game normally (and nothing is on the main menu stack or game stack).

The function `ga_win_get_cursor_pos` gets the position of the cursor. Note that it is up to the user to render the cursor itself (probably by calling `ga_win_quad`).

The function `ga_win_get_cursor_diff` gets the difference in the cursor's position between this update and the previous update.

We want to encourage having windows which the user can scroll through. Although the "screen coordinates" are always between (0.0,0.0) and (1.0,1.0) the virtual coordinates (or "map coordinates") can be in any range. Note that all window rendering API functions use screen coordinates instead of map coordinates. For map coordinates, we provide a minimal set of functions for converting

back and forth between screen coordinates and map coordinates. The function `ga_win_set_scroll_bounds` sets the min and max map coordinates for the screen. For example, calling

```
ga_win_set_scroll_bounds(
    wid, 3.0, 3.0, 5.0, 5.0)
```

will set the lower left screen location (0.0, 0.0) to be the map location (3.0, 3.0), and it will set the upper right screen location (1.0, 1.0) to be the map location (5.0, 5.0).

Use `ga_win_m_to_s` to convert from map coordinates to screen coordinates. Use `ga_win_s_to_m` to convert from screen coordinates to map coordinates.

18.19 Keyboard and Mouse Input without the WID

```
bool ga_win_key_down(string key);
bool ga_win_mouse_down(bool left);
bool ga_win_mouse_down2(int button);
```

Use these functions to determine if the given key or mouse button is currently down. Do not confuse this with a key or mouse button being "pressed". The `ga_win_mouse_down` tells you if the left or right mouse button is down (mouse buttons 1 or 2). The function `ga_win_mouse_down2` tells you whether the specified button is down, where the engine supports mouse buttons 1 through 5 inclusive. However we recommend using `ga_win_mouse_down` instead of the function `ga_win_mouse_down2`, because some users only have mice with two buttons.

Note that no WID is required! This allows you to call these functions at any time. On the other hand, the function `ga_win_key_pressed` can only be called at certain times during the game's cycle.

18.20 Keyboard and Mouse Input with the WID

```
bool ga_win_key_pressed(int wid, string key);
bool ga_win_key_pressed_or_spammed(
    int wid, string key, float init_wait, float subsequent_wait);
bool ga_win_key_released(int wid, string key);

bool ga_win_mouse_pressed(int wid, bool left);
bool ga_win_mouse_pressed2(int wid, int button);
bool ga_win_mouse_released(int wid, bool left);
bool ga_win_mouse_released2(int wid, int button);

bool ga_win_mouse_wheel_up(int wid);
```

```
bool ga_win_mouse_wheel_down(int wid);
```

The function `ga_win_key_pressed` return whether a given key has been pressed during this update phase. The function `ga_win_key_released` is similar: It returns whether the specified key was released. Here are some valid key strings:

```
"A" through "Z"
"0" through "9"
"F1" through "F12"
"ESC"
"ENTER"
"SPACE"
"LEFT"
"RIGHT"
"/"
```

To get a list of all key names, open the console and run the command

```
bind dump_inputs
```

The functions `ga_win_mouse_pressed` and `ga_win_mouse_released` return whether a mouse button (left or right) was pressed or released. The mouse pressed 2 and mouse released 2 functions work for mouse buttons 1 through 5 inclusive (but again we do not recommend using them because some users only have two buttons on their mice).

The function `ga_win_mouse_wheel_up` returns whether on not the mouse wheel was scrolled up (at least once). The function `ga_win_mouse_wheel_down` is the same except for scrolling down.

The function `ga_win_key_pressed_or_spammed` is just a helper function. Assuming the user is holding down a key, the function returns true after `init_wait` many seconds since the key was pressed. Then, it returns true once each `subsequent_wait` many seconds afterwards.

18.21 Other Input Related

```
void ga_win_release_keys_that_are_down();
```

The input bind system keeps track of what keys it thinks are down. When you call this function, the bind for the key release event will be called for each function that the bind system thinks is down.

For example, if the player is pressing the forward key (W) and they open some kind of menu that does NOT pause the game, you might want to call this function which would simulate the event for the W key being released. Note that if the input bind system misses an update, then it will automatically call this function. So, you do not need to worry about calling this function when opening windows that pause the game.

Chapter 19

Other Parts of Packages

At this point we have described all the folders inside a package. However, there are also the following files in the package's folder:

- binds.txt
- dependencies.txt
- globals.txt
- light_params.txt

In this chapter we will describe these files.

19.1 binds.txt

The file binds.txt specifies what happens by default when players press and release keys and mouse buttons. The way the input system works is that "input events" are bound to "actions". The file binds.txt declares actions and the input event that by default binds to that action.

Actions have a primary and a secondary command. Most input events are of type "downup", which means that when the associated key is pressed, then primary command of the associated action is executed. When the key is released, the secondary command of the associated action is executed. Suppose the file binds.txt is as follows:

```
PACKAGE_JUMP  SPACE.downup "" "game_input jump ""
```

The first "" means that the command has the empty string as a "nickname". The nickname of an action can be helpful for when the user wants to rebind actions. Note that the player could, for example, bind E.downup to the PACKAGE_JUMP action. Then when the player presses E the player will jump.

The "game_input jump" is the primary command of the PACKAGE_JUMP action. So when the space bar is pressed, this command is executed. Note that

the game_input jump command results in the top.game_input function being called with "jump" as the string argument. The secondary command of the PACKAGE_JUMP action is the empty string.

To summarize, the syntax of a line in the binds.txt file is the following:

```
PACKAGE_ACTION_NAME INPUT_EVENT NICKNAME PRIMARY_CMD SECONDARY_CMD
```

Let us give another example. Consider the action of moving forward, which is usually bound to the W key. The PACKAGE_ACTION_NAME might be something like "PACKAGE_MOVE_FORWARD". Note: all action names declared in binds.txt must start with "PACKAGE_". Next, the INPUT_EVENT would be "w.downup". The NICKNAME could be anything, so let us set it to be "". We can have the primary command be

```
"game_input \"move forward start\""
```

(including the surrounding quotation marks). So when the W key is pressed, the function top.game_input will be given the string "move forward start".

We can have the secondary command be

```
"game_input \"move forward end\""
```

So when the W key is released, the function top.game_input will be given the string "move forward end".

19.2 dependencies.txt

This is described in Section 1.4.

19.3 globals.txt

You can read how to set and get (global) environment variables in Section 17.8. All global variables that are loaded and saved (each time the game is loaded or saved) must be declared in the file "globals.txt". The type of the variable must be specified. Optionally an initial value can be set.

Here is an example of what the globals.txt file might look like:

```
b invisible false
i health 100
f player_height 1.7
v initial_velocity 0.0 0.0 0.0
s favorite_color "blue"
s last_town
```

Here the last_town variable does not have an initial value, so it will be initialized to the empty string. Similarly a vector without an initial value will be set to (0.0, 0.0, 0.0). A float without an initial value will be set to 0.0. An int without an initial value will be set to 0. A bool without an initial value will be set to false.

19.4 light_params.txt

This file holds "lightweight parameters" associated to the package. These may be read before the package is fully loaded. The following parameters should be defined in this file:

```
preferred_engine_version
version
chunk_width
```

The engine has a version, such as "1.01.09". The format for the engine version is "major.minor.patch". If the preferred_engine_version of the package does not match the actual engine version, there may be a warning.

Every saved game stores the engine version number of the engine during the last time the saved game was played. If either the major or minor changes (if the engine version is different from the one in the save file), then when loading the package a warning message will be displayed saying that the engine version has changed since the list time that package was played. However if only the patch number changes then there will be no such warning.

The package also has its own version which is specified here in the version variable. Every saved game stores this package version number of the package during the last time the saved game was played. The format for the version should should be "major.minor.patch". Again if major or minor change, then a warning message will be displayed. However if only the patch number changes then there will be no such warning.

The chunk_width specifies the width of each chunk. This must be an integer between 2 and 16 inclusive.

Here is what light_params.txt might look like:

```
preferred_engine_version = "1.01.09"
version = "1.01.09"
chunk_width = 16
```