Fractal Block World 1.01.13
Creation Manual

by Dan Hathaway

October 29, 2023

Contents

Chapter 1

Introduction

1.1 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".

Ideally all but a select few of the script .lua files within a package will have a prefix of the package name. So for example in the blank package, most of the script files start with "blank". So when you copy the blank package to myworld, for cleanliness you could replace each instance of the word "blank" in the lua files with "myworld". Certain files are required to be named certain things because they are entry points. For example, "Game/top.lua".

1.2 Dependencies

Your package can depend on other packages, as specified by the dependencies.txt file in the myworld directory.

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

```
wf base
```

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. Moreover, it we look at all well-founded package dependencies of the current package, then the dependencies of these packages with each other should form a directed acyclic graph.

Suppose the current package depends on the packages X and Y, but X and Y depend on each other. Then in dependencies.txt we would have

```
nonwf X
nonwf Y
```

The X/dependencies.txt file would be

```
nonwf Y
```

and the Y/dependencies.txt file would be

```
nonwf X
```

However, we consider this practice of packages depending on each over to be bad and it should be avoided.

To make like simple, only depend on the base package.

1.3 Subdirectories

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

- BasicEnts
- EnvRects
- Game
- Meshes
- MovingEnts
- 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

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.

1.4 Errors (causing the program to exit)

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

1.4.1 System Errors

System errors are generally due to bugs in the program. Usually 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.

1.4.2 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 chunk generation Lua module returns a texture name that does not exist, this is a hard user error.

1.4.3 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.

1.5 Lua-to-C API’s

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

- Chunk Generation API
- Initialization API
- Game API

1.5.1 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).

1.5.2 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.

$\boxed{\text{Functions part of the Initialization API start with ia\_}}$

1.5.3 Game API

The Game API can be used by

- Basic Entites (in BasicEnts/)
- Environment Rects (in EnvRects)
- Game Lua Scripts (modules) (in Game/)
- Moving Entities (in MovingEnts/)
- Windows (in Windows/).

$\boxed{\text{Functions part of the Game API start with ga\_}}$

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.jpg
```

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 ".tga". If not, the file type must be ".jpg". 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
```

The first string is the NAME of the sound. The second string is the FILENAME of the sound file. The only file type supported for sounds for this program is "Ogg Vorbis".

Chapter 3

Block Lua Scripts Part 1

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, a 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 complete the discussion in Chapter 4.

These “main” functions of Block Lua Scripts also have 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 7.

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       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/cave_world_top.lua
```

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 ∼) 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.

Additionally, the starting configuration file can have a line like as follows:

```
emergency_waypoint_path 778_777_c5f
```

This will set the "emergency waypoint" to be in the chunk with the specified path. The player is always able to teleport to his emergency waypoint from any other waypoint (without having to manually activate the emergency waypoint). It is also possible to add other built-in waypoints that are activated from the beginning of a new game. This can be done by calling the functions "ga_add_waypoint_sloppy" and "ga_add_waypoint_sloppy_in_only" from the file "Game/top.lua" in the "new_game" function.

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 "grass". Then, when the block needs to get subdivided into a chunk, the "main" function in the script

WorldNodes/Nodes/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". Here is what the file "grass.lua" might look like:

```
-- Block type: "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("r_green")
    for x = 0,15 do
        for y = 0,15 do
            z = 15
            set_pos(x,y,z,"grass2")
        end
    end
end
```

When a grass type block is expanded into a chunk, it is composed of solid r_green blocks, except for the top layer which consists of 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/big_tree.lua

exists, then "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, "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 "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,"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 "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 "r_concrete.lua":

```
function get_is_solid() return true end
function get_tex() return "block_concrete" end
function main() set_default_block("r_concrete") end
```

When a type "r_concrete" block is subdivided, it turns into 16 by 16 by 16 smaller "r_concrete" blocks. For organization purposes, I would recommend prefixing these types of blocks with "r_" (for "recursive").

It is also convenient to have a file called "s.lua" ("s" for "solid", and it is easy to type). The file "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("s") end
```

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

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

```
function get_is_solid() return false end
function get_tex() return "" end
function main() set_default_block("e") end
```

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

3.3 The 3 Necessary Functions

Consider the file "WorldNodes/Nodes/grass.lua" presented in the section about the directory "WorldNodes/Nodes". This lua script is executed whenever a "grass" type block needs to be subdivided to become a chunk. There are 3 functions defined in "grass", 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 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 is "p.main". Again here is the main function in our example "grass":

```
function p.main()
    set_default_block("r_green")
    for x = 0,15 do
        for y = 0,15 do
            z = 15
            set_pos(x,y,z,"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 "grass2" type blocks. The rest of the blocks in the chunk are of type 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 is not wise to try to maintain state in a Lua module using a global 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.

3.4 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 "air" type, except one block which is of type "iron":

```
function p.main()
    set_default_block("air")
    set_pos(7,7,7,"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("air1")
    set_pos(7,7,7,"iron")
    set_default_block("air2")
end
```

You might think that the second call to "set_default_block" will replace the iron block with an 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 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("air")
    set_pos(7,7,7,"iron")
end
```

the "set_pos" function is used to set the block at position (7,7,7) to be of type "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("air")
    set_pos(7,7,7,"iron")
    set_pos(7,7,7,"grass")
end
```

the position (7,7,7) is initially set to have type "air", then it is set to be of type "iron", and finally it is set to have type "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("air")
    set_pos(7,7,7,"iron")
    block_type = get_pos(7,7,7)
end
```

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

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("air")
    if (randf() < 0.8) then
        set_pos(7,7,7,"steel")
    else
        set_pos(7,7,7,"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("air")
    x = randi(0,15)
    y = randi(0,15)
    z = randi(0,15)
    set_pos(x,y,z,"iron")
end
```

3.7 Getting Chunk Generation Input

```
int get_input_path_length();
PATH get_input_path();
BTS get_input_path_bts();
string get_input_adj_bt(int dx, int dy, int dz);
```

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.

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. 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("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
```

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("air")
    below_type = get_input_adj_bt(0,0,-1)
    if below_type == "grass" then
        --Can actually have a dandelion.
        create_rect(7,7,0, 7,7,7, "green")
        create_rect(6,6,6, 8,8,8, "yellow")
    end
end
```

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("air")
    if (randf() < 0.8) then
        set_pos(7,7,7,"steel")
    else
        set_pos(7,7,7,"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

```
void 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("air")
    for i = 1,10 do
        x = randi(0,15)
        y = randi(0,15)
        -- Making a "tree".
        set_pos(x,y,0,"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("dirt")
    for i = 1,10 do
        x = randi(0,15)
        y = randi(0,15)
        -- Making a "tree root".
        set_pos(x,y,15,"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

```
void seed_xy();
void seed_xz();
void 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("dirt")
    -- 10 shafts:
    for i = 1,10 do
        -- Creating an air shaft.
        x = randi(0,15)
        y = randi(0,15)
        for z = 0,15 do
            set_pos(x,y,z,"air")
        end
    end
end
```

3.8.4 _chop type seed functions

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

```
void 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 "mars_like_planet" has already been created. Here is what the main function of "mars_like_planet_treasure" might look like:

```
function p.main()
    set_default_block("dirt")

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

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

```
function p.main()
    set_default_block("air")
    add_ent(7,7,7,"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:

```
base_save
base_ring_green
base_ring_red
```

```
base_ring_pink_source
base_ring_pink_dest
base_ring_blue
base_respawn_point
base_waypoint_out_only
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”:

```
base_txt
base_waypoint
base_waypoint_in_only
```

Here is an example:

```
function p.main()
    set_default_block("air")

    msg = "You better have enough rockets. "
        .. "Seriously. "
    add_ent_s(7,7,7,"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 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 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,"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 add_ment_f

```
void add_ment_f(float x, float y, float z, string type);
```

This function will add a moving entity (MEnt) centered at position (x,y,z) of the current chunk. This is identical to add_ment, except the function add_ment will place the ment as the center of a block.

3.12.3 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.

3.12.4 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.

3.12.5 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

In the chapter Chunk Generation 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);

//-----------------------------------------------
//                    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(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z, string type);
void create_sprinkles(
    int min_x, int min_y, int min_z,
    int max_x, int max_y, int max_z,
    float prob, string 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 Chunk Generation Input
//-----------------------------------------------

//Getting the input.
int get_input_path_length();
PATH get_input_path();
BTS get_input_path_bts();
string get_input_adj_bt(int dx, int dy, int dz);

//-----------------------------------------------
//  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);

//-----------------------------------------------
//              Blue Ring Related
//-----------------------------------------------

//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);

//-----------------------------------------------
//              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 add_ment_f(float x, float y, float 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();

//----------------------------------------------
//                  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 "Chunk Generation Lua Scripts Part 1".

4.2 More Block Functions

4.2.1 create_rect

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

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

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.

4.2.2 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
```

4.3 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.

4.3.1 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".

4.3.2 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.

4.3.3 Example

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

```
function p.main()
    set_default_block("e") --Empty block.

    --The vertices and the edges of the maze
    --will be solid (of type "s").
    --Everything else will be empty (of type "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,"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,"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,"s")
            end
        end
    end
    end
end
```

4.3.4 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,"r_black")
        end
    end
    end
```

4.3.5 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,"r_green")
set_pos(15,15,7,"r_green")

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

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

4.4 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.

4.4.1 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.

4.4.2 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.

4.4.3 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("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
        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,"e")
        end
    end
    end
    end
end
```

4.4.4 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.
    filled_nodes = {}

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

        data = caves_close_to_node2(x,y,z)
        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,"e")
        end

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

            if (dist < 1.5) and
                filled_nodes[which_node] == nil
            then
                filled_nodes[which_node] = true
                add_ent(x,y,z, "gold_10")
            end
        end

    end
    end
    end
```

4.5 Debugging

4.5.1 print

`void print(string str);`

This will print the given string str to standard output. At the beginning the following will be prepended:

"Proc world gen: CHUNK_FILENAME.lua: "

4.5.2 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."

4.5.3 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.

Chapter 5

Block Lua Scripts Part 3

5.1 More Block Lua Module Functions

We have already seen in Section 3.3 3 functions that appear in all Block Lua Scripts: get_is_solid, get_tex, and 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.

Here is a list of more functions which can be put in Block Lua Scripts:

```
//-----------------------------------------------
//        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();

//-----------------------------------------------
//      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();

//-----------------------------------------------
//    The main function (generating the chunk)
//-----------------------------------------------

void main(); //Mandatory.
```

5.2 Other Module Functions in Block Lua Scripts

There are other Module functions in Block Lua Scripts. These are functions used by the "Game" system. These functions are described in the chapter Block Lua Scripts Part 4.

For example, perhaps there might be a function in a block's Chunk Generation Lua script that is called when the player wants to "use" the block. We decided to keep all of these module functions in the same Block Lua Script, instead of having multiple Lua scripts describing a single block type.

5.3 Lua-To-C API's

The "main" function can call functions in the Chunk Generation Lua-To-C API. However the remaining functions described in this chapter (get_is_solid_physically, get_is_solid_visibly, get_is_solid_visibly_glass, get_tex_XXX, get_inv_tex_XXX) do NOT have access to any Lua-To-C API.

5.4 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.

5.5 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("invisible_wall")
end
```

Note the following:

$\boxed{\text{The get\_tex function should return a non-empty string iff the block is visibly solid.}}$

Here is a 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("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("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.

5.6 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 three possibilities: visibly solid, visibly empty, or visibly glass. 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. Specifically, each pixel in the texture should be either 100% transparent or 100% opaque. 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 an example of a glass 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("glass")
end
```

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

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_pos, etc.

For example, here is a 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("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 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("e") --Empty.
    create_rect("ceiling_pain", 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? We allow you to configure that. That is, you should have the file

WorldNodes/side_tex_conflict.lua

which should look like the following:

```
--Returns true iff the inv side takes precedence.
function p.resolve(
    bt1, --solid type
    tex1,
    bt2, --empty type
    tex2,
    side)

    --Could make this more complex.
    --This function always returns true, so the
    --inv texture of the empty bt2 block always takes
    --precedence over the texture of the solid bt1 block.
    --Feel free to change this.
    return true
end
```

Chapter 6

Block Lua Scripts Part 4

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.

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

```
//----------------------------------------------
//                 Proximity Events
//----------------------------------------------

void on_close(int level, BlockPos bp);

void on_adj_block_changed(
    int level, BlockPos bp, int side,
    string adj_old_bt_str,
    string adj_new_bt_str);
```

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:

```
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...
```

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.

Chapter 7

STD 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.

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()”.

7.1 More Block Functions

7.1.1 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(7,7,7, 7,7,7, "stone"); Calling std.create_center(2,"stone") is equivalent to calling create_rect(7,7,7, 8,8,8, "stone"); Calling std.create_center(3,"stone") is equivalent to calling create_rect(6,6,6, 8,8,8, "stone"); Etc.

7.1.2 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(7,7,0, 8,8,15, "stone").

Here is a way you can create a hollow tube:

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

7.1.3 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")
```

7.1.4 std.create_edges

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

This function will create the 12 edges of the chunk.

7.1.5 std.create_shell

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

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

7.1.6 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.

Chapter 8

In Game Tools

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

8.1 The Path Command

If you open the console (press ∼) and run the command

```
path
```

it will print to the console the name of the script for whatever chunk you are in.

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. It will also print the chunk path as a list of triples (x,y,z).

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").

8.2 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.

Chapter 9

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.

9.1 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.

9.2 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.

9.3 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.

9.3.1 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).

9.3.2 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.

9.3.3 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.

9.4 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*.

9.4.1 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 $(8.0, 8.0, 8.0)$.

9.4.2 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.

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

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

9.5.1 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.
```

9.5.2 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 10

Environment Rect Lua Scripts

An Environment Rect can be added in the main function of a Chunk Generation 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 Chunk Generation 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).

10.1 Environment Rect Lua Script Module Functions

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

```
void on_touch();
```

10.1.1 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()
    ga_damage_player(true, 10000)
end
```

Here ga_damage_player is a function (which is part of the Game API) 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_set_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.

10.2 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 11

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 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).

11.1 Basic Entity Lua Script Module Functions

Here are all the module functions of Basic Entity Lua Scripts.

```
//-----------------------------------------------
//        Called During (Type) Initialization
//-----------------------------------------------
string get_mesh();
string get_mesh2();
bool get_pulsates();
float get_scale();
float get_touch_dist();

//-----------------------------------------------
//            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_telekinesis(int level, BlockPos bp);
```

11.2 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.

11.2.1 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.

11.2.2 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).

11.2.3 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.

11.2.4 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.

11.2.5 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.

11.3 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 9.5.1 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.

11.3.1 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.

11.3.2 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
```

11.3.3 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. 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
```

11.3.4 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
```

11.3.5 p.on_telekinesis

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

When the player uses their "telekinesis ability", all entities that are visible to the player and not too far away will have their p.on_telekinesis function called.

This p.on_telekinesis function may be removed from future versions of the game, in place of telekinesis being something completely soft coded by the package (instead of something that is part of the engine).

11.4 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

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

Chapter 12

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).

12.1 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.

12.2 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.

12.3 Moving Entity Lua Script Module Functions

Here are the (module) functions that can exist in Moving Entity Lua Scripts. These functions are called by the game engine.

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

//-----------------------------------------------
//            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_die(
    int inst_id);
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_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);
bool get_can_use(
    int inst_id);
string get_use_msg(
    int inst_id);
void on_use(
    int inst_id);
```

12.3.1 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.

12.3.2 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, the on_add_to_live_world 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”.

12.3.3 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.

12.3.4 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).

12.3.5 on_die

```
void on_die(int inst_id);
```

If the moving entity has a variable called "health", then when this variable is first ≤ 0 during a discrete update, then the on_die function will be called.

This is an example of what the function might look like:

```
function p.on_die(inst_id)
    --Dropping some gold.
    local level = ga_ment_get_i(inst_id, "__level")
    local lp = ga_ment_get_lp(inst_id) --The position (level position).
    local bp = std.lp_to_bp(lp)
    local exist_length = 5*60 --Will exist for 5 minutes.
    ga_bent_add(level, bp, "gold_10", exist_length)
end
```

The on_die function may be removed in future versions of the game.

12.3.6 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).

12.3.7 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
```

12.3.8 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
```

12.3.9 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
```

12.3.10 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.

12.3.11 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
```

12.3.12 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, then that is equivalent to the function existing and always returning the 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 = 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
```

12.3.13 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
```

12.4 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).

12.4.1 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.

12.4.2 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.

12.4.3 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).

12.5 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). Some of the variables are static.

```
static bool __disable_saving = false

READ_ONLY bool __from_world_gen = false (rl = one million minutes)

static float __ttl = -1.0
static float __ttl_grounded = 60*60 (one hour)
READ_ONLY float __game_end_time = -1.0 (rl = one million minutes)
float __respawn_length = 60*60 (one hour) (rl = one million minutes)

static int __extra_min_levels = 0
static int __extra_max_levels = 0

READ_ONLY int __start_level = -1 (rl = one minute)
READ_ONLY int __min_level = -1 (rl = one minute)
READ_ONLY int __max_level = -1 (rl = one minute)
READ_ONLY int __level = -1 (rl = one minute)
READ_ONLY int __chunk_id = -1 (rl = one minute)

READ_ONLY Vector __offset = Vector(7.5, 7.5, 7.5) (rl = one million minutes)
READ_ONLY Vector __offset_old = Vector(7.5, 7.5, 7.5) (rl = one million minutes)

Vector __vel = Vector(0.0, 0.0, 0.0) (rl = one million minutes)

static string __mesh = ""

int __team_id_source = 0 (rl = one minute)
int __team_id_target = 0 (rl = one minute)

bool __solid_wrt_player = false (rl = one minute)
bool __collides = true (rl = one minute)

float __radius = 1.0 (rl = one minute)
bool __radius_lvlinv = false (rl = one minute)

string __tex_override = "" (rl = one minute)

bool __homing = false (rl = one minute)
READ_ONLY int __homing_target = -1 (rl = one minute)

float __turn_speed = 1.0 (rl = one minute)
bool __turn_towards_player = false (rl = one minute)
bool __turning_disabled = false (rl = one hour)
bool __mesh_fixed_frame = false (rl = one minute)
Vector __mesh_fixed_frame_v1 = Vector(1.0, 0.0, 0.0) (rl = one minute)
Vector __mesh_fixed_frame_v2 = Vector(0.0, 1.0, 0.0) (rl = one minute)
Vector __mesh_fixed_frame_v3 = Vector(0.0, 0.0, 1.0) (rl = one minute)
```

```
READ_ONLY bool __towards_viewer_valid = false (rl = one minute)
READ_ONLY Vector __towards_viewer_vec = Vector(0.0, 0.0, 0.0) (rl = one minute)
READ_ONLY Vector __towards_viewer_dir = Vector(0.0, 0.0, 1.0) (rl = one minute)
READ_ONLY float __dist_to_viewer = -1.0 (rl = one minute)
READ_ONLY float __dist_to_viewer_old = -1.0 (rl = one minute)

string __death_anim = "" (rl = one minute)
float __death_anim_start = -1.0 (rl = one minute)
float __death_anim_end = -1.0 (rl = one minute)
READ_ONLY int __death_anim_stage = 0 (rl = one minute)
```

12.6 Explanation of all moving entity built-in vars

12.6.1 __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 15.2.4. 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.

12.6.2 __from_world_gen

```
READ_ONLY bool __from_world_gen = false (rl = one million minutes)
```

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. This version of the game does not support non-roaming entities moving from their original chunk. In other words, all non-roaming entities must be grounded. This may change in a later version of the game.

12.6.3 __ttl, __ttl_grounded, __game_end_time

```
static float __ttl = -1.0
static float __ttl_grounded = 60*60 (one hour)
READ_ONLY float __game_end_time = -1.0 (rl = one million minutes)
```

__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. __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 reverted back to its original state. To make life simple, it makes sense to leave __ttl_grounded as one hour: all changes to a moving entity will be reverted in one hour. Note that once all variables of a moving entity have reverted to their default value, then the moving entity itself is automatically reverted by the engine. So one hour is an upper bound for how long it will take a moving entity to be fully reverted.

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 reverted back to its original state.

Note that __game_end_time is in game time, not in a level's time.

12.6.4 __respawn_length

```
float __respawn_length = 60*60 (one hour) (rl = one million minutes)
```

__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, then it will remain gone and will not respawn for __respawn_length many seconds. After removing a non-roaming entity, in the chunk where the entity was originally created we leave a tag which says that the moving entity has been removed. In part of this tag, we say when the the moving entity will respawn. 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).

12.6.5 __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
```

12.6.6 __start_level, __min_level, __max_level

```
READ_ONLY int __start_level = -1 (rl = one minute)
READ_ONLY int __min_level = -1 (rl = one minute)
READ_ONLY int __max_level = -1 (rl = one minute)
```

These are described in the previous subsection.

12.6.7 __level, __chunk_id

```
READ_ONLY int __level = -1 (rl = one minute)
READ_ONLY int __chunk_id = -1 (rl = one minute)
```

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.

12.6.8 __offset, __offset_old

```
READ_ONLY Vector __offset = Vector(7.5, 7.5, 7.5) (rl = one million minutes)
READ_ONLY Vector __offset_old = Vector(7.5, 7.5, 7.5) (rl = one million minutes)
```

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.

12.6.9 __vel

```
Vector __vel = Vector(0.0, 0.0, 0.0) (rl = one million minutes)
```

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.

12.6.10 __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.

12.6.11 __team_id_source, __team_id_target

```
int __team_id_source = 0 (rl = one minute)
int __team_id_target = 0 (rl = one minute)
```

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).

12.6.12 __solid_wrt_player, __collides

```
bool __solid_wrt_player = false (rl = one minute)
bool __collides = true (rl = one minute)
```

The variable __solid_wrt_player is true iff the player cannot move through the moving entity. When this is true, physically the moving entity is modeled as a sphere with radius __radius.

When the variable __collides is true, the moving entity can collide with other moving entities and blocks. This will result in the on_block_hit and on_ment_hit functions of the moving entity being called when there is a collision. Collisions are detected by moving the moving entity while keeping everything else in the world still.

12.6.13 __radius, __radius_lvlinv

```
float __radius = 1.0 (rl = one minute)
bool __radius_lvlinv = false (rl = one minute)
```

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.

12.6.14 __tex_override

```
string __tex_override = "" (rl = one minute)
```

Recall that the mesh name __mesh of the moving entity is associated to a wavefront.obj file and a texture name. These associations can be found in

"Meshes/mesh_names.txt".

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.

12.6.15 __homing, __homing_target

```
bool __homing = false (rl = one minute)
READ_ONLY int __homing_target = -1 (rl = one minute)
```

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.

12.6.16 __turn_speed, __turn_towards_player, __turning_disabled

```
float __turn_speed = 1.0 (rl = one minute)
bool __turn_towards_player = false (rl = one minute)
bool __turning_disabled = false (rl = one hour)
```

When a moving entity is put in the world, it starts with a random orientation. If __turn_towards_player is true, then the entity will turn towards the player with a speed specified by __turn_speed (1.0 is the default). When __turning_disabled is true, turning is disabled.

12.6.17 __mesh_fixed_frame, __mesh_fixed_frame_vX

```
bool __mesh_fixed_frame = false (rl = one minute)
Vector __mesh_fixed_frame_v1 = Vector(1.0, 0.0, 0.0) (rl = one minute)
Vector __mesh_fixed_frame_v2 = Vector(0.0, 1.0, 0.0) (rl = one minute)
Vector __mesh_fixed_frame_v3 = Vector(0.0, 0.0, 1.0) (rl = one minute)
```

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 vectors __mesh_fixed_frame_v1, __mesh_fixed_frame_v2, and __mesh_fixed_frame_v3 (which should be orthogonal and of length one).

12.6.18 __towards_viewerXXX and __dist_to_viewerXXX

```
READ_ONLY bool __towards_viewer_valid = false (rl = one minute)
READ_ONLY Vector __towards_viewer_vec = Vector(0.0, 0.0, 0.0) (rl = one minute)
READ_ONLY Vector __towards_viewer_dir = Vector(0.0, 0.0, 1.0) (rl = one minute)
READ_ONLY float __dist_to_viewer = -1.0 (rl = one minute)
READ_ONLY float __dist_to_viewer_old = -1.0 (rl = one minute)
```

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_viewerXXX 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.

12.6.19 __death_animXXX

```
string __death_anim = "" (rl = one minute)
float __death_anim_start = -1.0 (rl = one minute)
float __death_anim_end = -1.0 (rl = one minute)
READ_ONLY int __death_anim_stage = 0 (rl = one minute)
```

When a moving entity "dies", it can optionally use a "death animation". Right now there is only one death animation: "dark_hole". When this is set, 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.

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 13

Window Lua Scripts

13.1 Introduction

There are three types of windows:

- Main menu windows
- Game windows
- HUD windows

All windows Lua scripts are put in the Windows directory of the package.

13.1.1 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.

13.1.2 Stacks vs Sets

The main menu windows are put into a "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" into or "popped" from this stack.

Game windows are also put into a similar 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).

13.2 Main Menu Windows

Main menu windows can be accessed from the “package’s top menu”. That is, when you are in a game, if you go to MAIN MENU → OPTIONS → PACKAGE TOP MENU you will be able to access the top most main menu window. This top most main menu window script MUST be called

“main_menu.lua”.

Here are the main menu window lua script module functions:

```
//-----------------------------------------------
//      Main Menu Windows Module Functions
//-----------------------------------------------
void on_start(int wid);
void on_end(int wid);
void process_input(int wid);
void render(int wid);
```

13.2.1 p.on_start

```
void on_start(int wid);
```

Consider the lua script “my_window.lua”. When a window of this type is pushed onto the main menu window stack, then this on_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.on_start function of my_window.lua might look like:

```
function p.on_start(wid)
    ga_play_sound_menu("chimes_sound")
end
```

Here the on_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).

13.2.2 p.on_end

```
void on_end(int wid);
```

Consider the lua script “my_window.lua”. When a window of this type is popped from the main menu window stack, then this on_end function will be called.

There is another situation when on_end will be called. If W.lua is the window on top of the main menu stack and another window is pushed on top of it, then the p.on_end function of W.lua will also be called.

Here is what the p.on_end function of my_window.lua might look like:

```
function p.on_end(wid)
    ga_play_sound_menu("sad_sound")
end
```

13.2.3 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
        local return_to_game = true
        ga_main_menu_pop_all(return_to_game)
        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.

13.2.4 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 main menu window stack.

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.

13.2.5 An Example

This is what the 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_main_menu_push("win_get_free_gold")
        return
    end
    if( selection_str == "LIST OF CHEAT CODES" )
        ga_main_menu_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_main_menu_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 17.

13.3 Game Windows

Game Windows are basically identical to Main Menu Windows. The only main difference is that there are two window stacks: one for each type of window.

You would use a game window if you wanted the player to look at their inventory. You would use a main menu window if you wanted the player to be able to change the difficulty of the game.

13.4 HUD Windows

HUD windows are similar to main menu and game windows, except they do not have on_start and on_end functions. They only have process_input and render functions. These functions are only called when there are no windows in either the main menu window stack or the game window stack.

Chapter 14

Game Lua Scripts

14.1 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.

14.2 All top.lua Module Functions

```
//----------------------------------------------
//        Game/top.lua Module Functions
//----------------------------------------------
void top.new_game();
void top.load_game();
void top.update()
void top.update_passive();
void top.update_discrete_pre();
void top.update_discrete_post();
string top.game_input(string str);
void top.killed_player();
void top.respawn_player();
```

14.3 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
```

14.4 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
```

14.5 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.

14.6 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).

14.7 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).

14.8 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
```

14.9 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.

14.10 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.

14.11 top.respawn_player

```
int top.respawn_player();
```

Once the player is dead, to respawn the player must call either the system command "respawn passive" or the system command "respawn force". 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()
    ga_set_i("health", 100)
    ga_set_i("bullets", 0)
    ga_set_i("shells", 0)
    ga_set_i("rockets", 0)
end
```

Chapter 15

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".

15.1 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_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_var_saving(int tid, string var, bool value);

//-----------------------------------------------
//          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_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);
```

15.2 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.

15.2.1 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.

15.2.2 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.

15.2.3 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);
```

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 12.5 for a list of all the built-in moving entity variables, and see Section 12.6 for an explanation of what these variables do.

15.2.4 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.

15.3 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.

15.3.1 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
```

15.3.2 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".

15.4 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.

15.4.1 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 16

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.

16.1 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".

16.2 The Full Game Lua-to-C API

```
//-----------------------------------------------
//             Program Level Functions
//-----------------------------------------------

void ga_command(string command);

void ga_save(bool play_sound);
void ga_load();
void ga_exit();

void ga_print(string line);
void ga_flush();

void ga_console_print(string line);

void ga_dump_lua_env();

//-----------------------------------------------
//         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);

//-----------------------------------------------
//           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);

//-----------------------------------------------
//                   Env Vars
//-----------------------------------------------
//Setting if var exists.
bool ga_exists(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 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);

//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);

//-----------------------------------------------
//                   Sounds
//-----------------------------------------------

void ga_play_sound(string sound);
void ga_play_sound_menu(string sound);
void ga_play_music(string sound);
void ga_stop_music();

//-----------------------------------------------
//                 Game Stuff
//-----------------------------------------------

//System game stuff.
bool ga_genesis();

//Life and death (game stuff).
bool ga_kill_player();

//-----------------------------------------------
//               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_dir_tex(string name, string tex, Vector dir);
```

```
//----------------------------------------------
//           Moving 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_tele_pink();
void ga_tele_pink2(Vector lp);
void ga_tele_blue();
void ga_tele_blue2(Vector lp);
void ga_tele_same_level(Vector lp);

//----------------------------------------------
//               Windows (Part 1)
//----------------------------------------------

//Window related.
void ga_window_push(string win_name);
void ga_window_pop();
void ga_window_pop_all();
void ga_main_menu_push(string win_name);
void ga_main_menu_pop();
void ga_main_menu_pop_all(bool return_to_game);
void ga_hud_window_add(string win_name, int priority);
void ga_hud_window_remove(string win_name);

//----------------------------------------------
//               Viewer queries
//----------------------------------------------

//Viewer queries.
int ga_get_viewer_chunk_id();
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);

//----------------------------------------------
//              Moving entities (type)
//----------------------------------------------

bool ga_ment_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 (inst)
//----------------------------------------------

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_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);

bool ga_ment_b_exists_and_true(int inst_id, string var);

void ga_ment_set_var_rt_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);

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);
void ga_ment_remove(int inst_id);
string ga_ment_get_type(int inst_id);
Vector ga_ment_get_lp(int inst_id);
Vector ga_ment_get_sllp(int inst_id);
void ga_ment_dump(int inst_id);

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();

//----------------------------------------------
//                  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
//----------------------------------------------

//LBPs.
int ga_lbp_seed_pos(int chunk_id, int lbp_hash);

//Blocks.
int ga_block_seed_pos(int level, BlockPos bp);
string ga_block_get(int level, BlockPos bp);
string ga_get_cocoon_block_of_chunk(int level, BlockPos 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);

void ga_block_get_b(int level, BlockPos bp, string var, bool value);
void ga_block_get_i(int level, BlockPos bp, string var, int value);
void ga_block_get_f(int level, BlockPos bp, string var, float value);
void ga_block_get_v(int level, BlockPos bp, string var, Vector value);
void ga_block_get_s(int level, BlockPos bp, string var, string value);

bool   ga_block_set_b(int level, BlockPos bp, string var);
int    ga_block_set_b(int level, BlockPos bp, string var);
float  ga_block_set_b(int level, BlockPos bp, string var);
Vector ga_block_set_b(int level, BlockPos bp, string var);
string ga_block_set_b(int level, BlockPos bp, string var);

//Block types.
bool ga_bt_get_physically_solid(string bt);

//----------------------------------------------
//         Respawn Point and Waypoints
//----------------------------------------------

void ga_set_respawn_point(string path, BlockPos lbp);
void ga_add_waypoint_sloppy(string path, string name_override);
void ga_add_waypoint_sloppy_in_only(string path, string name_override);
```

```
//-----------------------------------------------
//                 Coordinates
//-----------------------------------------------

int ga_chunk_id_to_level(int chunk_id);
BlockPos ga_chunk_id_to_vcp(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);

float ga_level_scale_factor(int source_level, int target_level);
Vector ga_convert_lp(
    int source_level, int target_level, Vector source_lp);

BlockPos ga_vcp_to_bp(int level, BlockPos vcp);

string ga_bp_to_path(int level, 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.

//-----------------------------------------------
//                    Math
//-----------------------------------------------
//Math.
//This is all in "base/Game/std.lua".

//-----------------------------------------------
//             Movement and Physics
//-----------------------------------------------

void ga_move_set_desired_travel(Vector travel);
void ga_move_set_spin(float spin);
bool ga_move_get_on_sure_footing();
void ga_move_set_ledge_guards(bool on);
void ga_move_set_body_spirit();
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);

//-----------------------------------------------
//             Window (Part 2)
//-----------------------------------------------
//These are described in another chapter.
//These are in addition to the previous "windows" functions.

//-----------------------------------------------
//          Deprecate Eventually
//-----------------------------------------------
void ga_add_emp_sphere(
    int chunk_id, Vector offset, float radius,
    float stun_length, int damage);
```

We will now describe all of these functions.

16.3 Game API: Program Level Functions

```
void ga_command(string command);

void ga_save(bool play_sound);
void ga_load();
void ga_exit();

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.

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).

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 funtion ga_console_print adds the given string to the console as a line (no newline character is required).

16.4 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.

16.5 Game API: Time

```
float ga_get_game_time();
float ga_get_level_time(int level);
```

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".

16.6 Game API: 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);
```

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).

16.7 Game API: Env Vars

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 18.3.

```
bool ga_exists(string var);
```

Use this ga_exists functions to determine if the given *global* variable exists. 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.

```
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.

```
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.

Here are the system variables that can be accessed:

```
//Bool system vars:
game.player.alive
game.constant_saving
game.player.move.fly.use_true_up (READ ONLY)
menu.in_main
metagame.cheat.god (READ ONLY)
metagame.cheat.enabled (READ ONLY)

//Int system vars:
game.input.mouse.wheel_value (READ ONLY)
game.package.seed (READ ONLY)
game.player.health_max
stats.in_dps
stats.last_in_dps

//Float system vars:
game.time.total
game.time.play (READ ONLY)
game.time.elapsed (READ ONLY)
game.player.move.fly.radius (READ ONLY)
game.player.move.ground.radius (READ ONLY)
game.player.move.ground.bot_to_eye (READ ONLY)
game.player.move.ground.eye_to_top (READ ONLY)
```

```
time.current (READ ONLY)
time.elapsed (READ ONLY)

//Vector system vars:
game.player.camera.look (READ ONLY)
game.player.camera.up (READ ONLY)
game.player.camera.left (READ ONLY)
game.player.move.last_pos_diff (READ ONLY)
game.player.camera.offset (READ ONLY)
menu.text_color (READ ONLY)

//String system vars:
game.player.name (READ ONLY)
game.package.config_desc (READ ONLY)
game.player.move.mode (READ ONLY)
```

16.8 Game API: Sounds

```
void ga_play_sound(string sound);
void ga_play_sound_menu(string sound);
void ga_play_music(string sound);
void ga_stop_music();
```

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.

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".

16.9 Game API: Game Stuff

16.9.1 ga_genesis

```
bool ga_genesis();
```

This determines if the game is in "genesis mode". This only affects the Xar package (so you can ignore this). This was here to make a version of the program where there were no monsters or weapons.

16.9.2 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().

16.10 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_dir_tex(string name, string tex, Vector dir);
```

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".

16.11 Game API: Moving 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_tele_pink();
void ga_tele_pink2(Vector lp);
void ga_tele_blue();
void ga_tele_blue2(Vector lp);
void ga_tele_same_level(Vector lp);
```

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.

The function ga_tele_pink is just like ga_shrink or ga_grow. It will teleport the user towards the root of the chunk tree as if they touched a Pink Ring Device. The function ga_tele_pink2 is similar except it first teleports the player to a location in the same level first before simulating touching a Pink Ring Device. If the first teleportation cannot happen, then that is fine and the player will simulate touching a Pink Ring Device from their current location.

The function ga_tele_blue and ga_tele_blue2 are like ga_tele_pink and ga_tele_pink2, except for Blue Rings instead of Pink Rings.

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.

16.12 Game API: Windows (Part 1)

```
void ga_window_push(string win_name);
void ga_window_pop();
void ga_window_pop_all();
void ga_main_menu_push(string win_name);
void ga_main_menu_pop();
void ga_main_menu_pop_all(bool return_to_game);
void ga_hud_window_add(string win_name, int priority);
void ga_hud_window_remove(string win_name);
```

There are three types of windows: game windows, main menu windows, and hud windows. A window cannot be in more than one category. The game windows are put into a stack, as are the main menu windows. 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_main_menu_push, ga_main_menu_pop, and ga_main_menu_pop_all are similar to their game windows counterparts, except for the main menu 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 the highest priority first.

16.13 Game API: Viewer Queries

```
int ga_get_viewer_chunk_id();
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 center of the world is the eyeball of the player (which we call the viewer).

ga_get_viewer_chunk_id gets the chunk id of the viewer. 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_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" 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. 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).

16.14 Game API: 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.

```
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).

16.15 Game API: Moving Entities (type)

```
bool ga_ment_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.

16.16 Game API: Moving Entities (inst)

16.16.1 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.

16.16.2 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.

16.16.3 Testing is a variable 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.

16.16.4 Changing the revert length of a variable

```
void ga_ment_set_var_rt_only(int inst_id, string var, float rl);
```

Use the ga_ment_set_var_rt_only function for changing the revert time of a variable for (this instance of) a moving entity. This does NOT change the value of the variable.

16.16.5 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. The revert time is already specified (it is associated to the variable).

16.16.6 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 persistant. 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).

16.16.7 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.

16.16.8 Removing a moving entity

```
void ga_ment_remove(int inst_id);
```

The ga_ment_remove function will remove the given moving entity (instance). If it is a roaming entity, it will be gone for good. If it is a non-roaming entity (it is generated from procedural world generation), then it will respawn after __respawn_length number of seconds.

16.16.9 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.

16.16.10 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").

16.16.11 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.

16.16.12 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.

16.16.13 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).

16.16.14 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.

16.16.15 Dumping all moving entities

```
void ga_ment_all_dump();
```

This will dump information about ALL moving entities that exist in the active chunk tree.

16.17 Game API: Particles

A particle is a point like entity used for rendering only. They are not saved when the game is saved.

16.17.1 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.

16.17.2 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.

16.17.3 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.

16.17.4 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.

16.18 Game API: Blocks

16.18.1 Local block position functions

```
int ga_lbp_seed_pos(int chunk_id, int lbp_hash);
```

This function gets a seed (for pseudo random number generation) that is determined by the given block position. The block is given by the chunk (the chunk_id) and the local block position hash code of the position within the chunk.

16.18.2 Miscellaneous block functions

```
int ga_block_seed_pos(int level, BlockPos bp);
string ga_block_get(int level, BlockPos bp);
string ga_get_cocoon_block_of_chunk(int level, BlockPos vcp);
```

The function ga_block_seed_pos is like ga_lbp_seed_pos, except it takes the level and the block position to determine the seed.

The function ga_block_get returns the block type (string) of the block with the given position.

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).

16.18.3 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 15.4. So calling one of these "block_change" functions will push a block onto a block stack.

16.18.4 Block variables

```
void ga_block_get_b(int level, BlockPos bp, string var, bool value);
void ga_block_get_i(int level, BlockPos bp, string var, int value);
void ga_block_get_f(int level, BlockPos bp, string var, float value);
void ga_block_get_v(int level, BlockPos bp, string var, Vector value);
void ga_block_get_s(int level, BlockPos bp, string var, string value);

bool   ga_block_set_b(int level, BlockPos bp, string var);
int    ga_block_set_b(int level, BlockPos bp, string var);
```

```
float  ga_block_set_b(int level, BlockPos bp, string var);
Vector ga_block_set_b(int level, BlockPos bp, string var);
string ga_block_set_b(int level, BlockPos bp, string var);
```

Use these functions to get and set block variables. The concept of the "revert time" of a block variable is discussed in Section 15.4.

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_diamond" 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 player_health = ga_get_i("var.health")
    player_health = player_health + 5
    ga_set_i("var.health", player_health)
    num_sodas = num_sodas - 1
    ga_block_set_i(level, bp, "num_sodas", num_sodas)
end
```

16.18.5 Block types

```
bool ga_bt_get_physically_solid(string bt);
```

The function ga_bt_get_physically_solid returns whether or not the block (of the given type) is physically solid.

16.19 Game API: Respawn Point and Waypoints

16.19.1 Respawn point

```
void ga_set_respawn_point(string path, BlockPos lbp);
```

This function sets 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.

16.19.2 Waypoints

```
void ga_add_waypoint_sloppy(string path, string name_override);
void ga_add_waypoint_sloppy_in_only(string path, string name_override);
```

These functions are called "sloppy" because we do not specify the position of the waypoint within the chunk. The function ga_add_waypoint_sloppy adds the given chunk to the list of available waypoints. Note that the chunk must actually contain a waypoint for this to work.

The function ga_add_waypoint_sloppy_in_only is similar but it applies to in-only waypoints. The reason why there are two of these functions is because a chunk could contain a normal waypoint and an in-only waypoint.

16.20 Game API: Coordinates

16.20.1 From chunk id

```
int ga_chunk_id_to_level(int chunk_id);
BlockPos ga_chunk_id_to_vcp(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_path returns the chunk path of the given chunk.

16.20.2 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.

16.20.3 Converting from one level to another

```
float ga_level_scale_factor(int source_level, int target_level);
Vector 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.

16.20.4 The block position of a chunk

```
BlockPos ga_vcp_to_bp(int level, BlockPos vcp);
```

Every chunk itself is a block. Suppose C is a chunk on level L. Suppose C has viewer centric position vcp (on level L). Now C is also a block on level $L-1$. To get the block position of C in level $L-1$, make the call ga_vcp_to_bp(L, vcp).

```
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, but the chunk occupying the same space as the block need not be in the active chunk tree.

16.20.5 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_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
```

16.21 Game API: Math

There are no math functions in the Game Lua-to-C API. However take a look at base/Game/std.lua for some math related functions.

16.22 Game API: Movement and Physics

16.22.1 Moving

```
void ga_move_set_desired_travel(Vector travel);
void ga_move_set_spin(float spin);
```

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_spin 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.

16.22.2 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.

16.22.3 Setting the body type

```
void ga_move_set_body_spirit();
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.

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.

16.22.4 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.

The functions ga_player_model_q2md2_set_cmd and ga_player_model_q2md2_set_state set the command and state respectively of the player model. The state can be one of "", "run", "crouch" and "crouch_run". The cmd can be one of "", "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.

16.23 Visibility

```
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.

16.24 Game API: Windows (Part 2)

These functions are described in Chapter 17.

16.25 Game API: Deprecate Eventually

These functions will probably be removed from the engine at some point.

```
void ga_add_emp_sphere(
    int chunk_id, Vector offset, float radius,
    float stun_length, int damage);
```

The function ga_add_emp_sphere adds an EMP blast sphere to the world. Note: these are not saved when the game is saved.Note: once there is an EMP blast, for every affected moving entity the function xar_ment_emp.emp_stun will be called.

Chapter 17

The Game Lua-to-C API: Windows

In Chapter 16 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.

17.1 The API

```
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);
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_quad(
    int wid, float min_x, float min_y, float max_x, float max_y, string tex);
void ga_win_quad_two(
    int wid, flaot 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_txt(
    int wid, float min_x, float min_y, string txt);
void ga_win_txt_alpha_bg(
    int wid, float min_x, float max_x, float alpha, string txt);
void ga_win_txt_center(
    int wid, float min_y, string txt);
```

```
void 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);

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);
int ga_win_widget_small_list_process_input(int wid);
string ga_win_widget_small_list_get_entry(int wid, int index);

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);

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_end(int wid);

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_mtos(int wid, float x, float y);
Vector ga_win_stom(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);

bool ga_win_key_pressed(int wid, string key);
bool ga_win_mouse_pressed(int wid, bool left);
bool ga_win_mouse_released(int wid, bool left);
bool ga_win_mouse_wheel_up(int wid);
bool ga_win_mouse_wheel_down(int wid);
bool ga_win_key_pressed_or_spammed(
    int wid, string key, float init_wait, float subsequent_wait);
```

17.2 The Window ID (WID)

All of these API functions take the window ID (WID) of the current window.

17.3 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);
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_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
```

17.4 Screen Elements

```
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);
```

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.

```
void ga_win_txt(
    int wid, float min_x, float min_y, string txt);
void ga_win_txt_alpha_bg(
    int wid, float min_x, float max_x, float alpha, string txt);
void ga_win_txt_center(
    int wid, float min_y, string txt);
void 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.

17.5 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.

17.6 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);
int ga_win_widget_small_list_process_input(int wid);
string ga_win_widget_small_list_get_entry(int wid, int index);
```

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_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.

17.7 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);
```

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 inputed (the user typed something and then pressed enter).

17.8 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_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 function ga_win_widget_mutable_text_box_get_text gets the text string for the mutable text box.

The function ga_win_widget_mutable_text_box_end destroys the mutable text box widget of the given window. That might naturally be called in the on_end function of the associated window script.

17.9 Cursor and Map Coordinates

```
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_mtos(int wid, float x, float y);
Vector ga_win_stom(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 scree and (1.0, 1.0) is the upper right hand corner.

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_mtos to convert from map coordinates to screen coordinates. Use ga_win_stom to convert from screen coordinates to map coordinates.

17.10 Keyboard and Mouse Input

```
bool ga_win_key_pressed(int wid, string key);
bool ga_win_mouse_pressed(int wid, bool left);
bool ga_win_mouse_released(int wid, bool left);
bool ga_win_mouse_wheel_up(int wid);
bool ga_win_mouse_wheel_down(int wid);
bool ga_win_key_pressed_or_spammed(
    int wid, string key, float init_wait, float subsequent_wait);
```

The functions ga_win_key_pressed return whether a given key has been pressed during this update phase. Here are the valid key strings:

```
"A" through "Z"
"0" through "9"
"F1" through "F12"
"ESC"
"ENTER"
"SPACE"
"LEFT"
"RIGHT"
"/"
```

The functions ga_win_mouse_pressed and ga_win_mouse_released return whether a mouse button (left or right) was pressed or released.

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.

Chapter 18

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.

18.1 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".

18.2 dependencies.txt

This is described in Section 1.2.

18.3 globals.txt

You can read how to set and get (global) environment variables in Section 16.7. 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.

18.4 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
```