Introduction

Welcome

Minetest uses Lua scripts to provide modding support. This online book aims to teach you how to create your own mods, starting from the basics.

What you will need

- A Code Editor. Talked about in the Lua Scripts chapter.
- A copy of Minetest in the 0.4 series. (eg: 0.4.13)
- The ability to work independently, without pestering other developers to write your code for you.
- Motivation to keep trying when things go wrong.

So, go on then.

Start reading. Use the navigation bar on the left (or on the top on mobiles) to open a chapter.

- GitHub.
- Download for offline use.
- Forum Topic.

About this Book

Noticed a mistake, or want to give feedback? Tell us about it using one of these methods:

- GitHub Issue.
- Post in the Forum Topic.
- Send me a PM on the Forum.
- Submit a report below.

You can contribute to this project on GitHub.
Read the contribution README.

Written by rubenwardy.

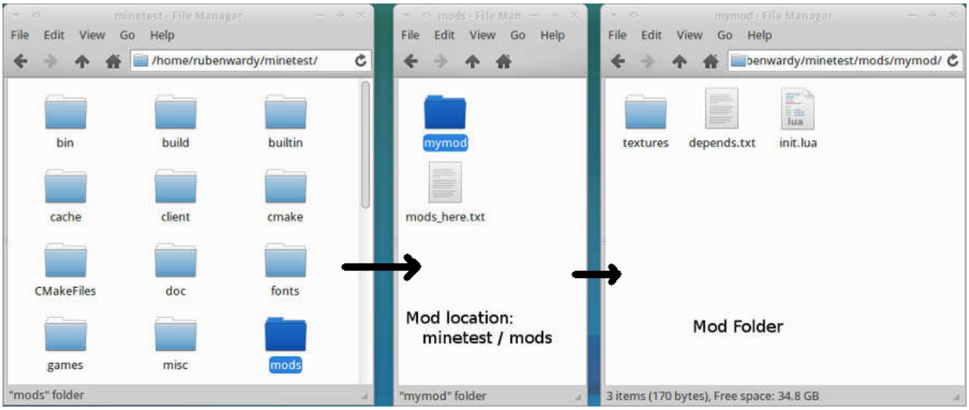
Folder Structure

Introduction

In this chapter we will learn the basic structure of a mod's folder. This is an essential skill when creating mods.

- Mod Folders
- Dependencies
- Mod Packs

Mod Folders



Each mod has its own folder where all its Lua code, textures, models, and sounds are placed. These folders need to be placed in a mod location such as minetest/mods. Mods can be grouped into mod packs which are explained below.

A "mod name" is used to refer to a mod. Each mod should have a unique mod name, which you can choose - a good mod name should describe what the mod does. Mod names can be made up of letters, numbers, or underscores. The folder a mod is in needs to be called the same as the mod name.

Mod Folder Structure

```
Mod name (eg: "mymod")
-    init.lua - the main scripting code file, which is r
-    (optional) depends.txt - a list of mod names that r
-    (optional) textures/ - place images here, commonly
-    (optional) sounds/ - place sounds in here
-    (optional) models/ - place 3d models in here
...and any other lua files to be included by init.lua
```

Only the init.lua file is required in a mod for it to run on game load; however, the other items are needed by some mods to perform their functionality.

Dependencies

The depends text file allows you to specify which mods this mod requires to run and what needs to be loaded before this mod.

depends.txt

```
modone
modtwo
modthree?
```

As you can see, each modname is on its own line.

Mod names with a question mark following them are optional dependencies. If an optional dependency is installed, it is loaded before the mod. However, if the dependency is not installed, the mod still loads. This is in contrast to normal dependencies which will cause the current mod not to work if the dependency is not installed.

Mod Packs

Modpacks allow multiple mods to be packaged together and be moved together. They are useful if you want to supply multiple mods to a player but don’t want to make them download each one individually.

Mod Pack Folder Structure

```
modpackfolder/
-    modone/
-    modtwo/
-    modthree/
-    modfour/
-    modpack.txt – signals that this is a mod pack, cont
```

Example Time

Are you confused? Don’t worry, here is an example putting all of this together.

Mod Folder

```
mymod/
-    textures/
-    -    mymod_node.png
-    init.lua
-    depends.txt
```

depends.txt

```
default
```

init.lua

```
print("This file will be run at load time!")

minetest.register_node("mymod:node", {
    description = "This is a node",
    tiles = {
        "mymod_node.png",
        "mymod_node.png",
        "mymod_node.png",
        "mymod_node.png",
        "mymod_node.png",
        "mymod_node.png"
    },
    groups = {cracky = 1}
})
```

Our mod has a name of “mymod”. It has two text files: init.lua and depends.txt.
The script prints a message and then registers a node – which will be explained in the next chapter.
The depends text file adds a dependency to the default mod which is in minetest_game.
There is also a texture in textures/ for the node.

Complex Chat Commands

Introduction

This chapter will show you how to make complex chat commands, such as `/msg <name> <message>`, `/team join <teamname>` or `/team leave <teamname>`.

- Why ChatCmdBuilder?
- Routes.
- Subcommand functions.
- Installing ChatCmdBuilder.
- Admin complex command.

Why ChatCmdBuilder?

Traditionally mods implemented these complex commands using Lua patterns.

```lua
local name = string.match(param, "^join ([%a%d_-]+)")
```

I however find Lua patterns annoying to write and unreadable. Because of this, I created a library to do this for you.

```lua
ChatCmdBuilder.new("sethp", function(cmd)
    cmd:sub(":target :hp:int", function(name, target, hp
        local player = minetest.get_player_by_name(targe
        if player then
            player:set_hp(hp)
            return true, "Killed " .. target
        else
            return false, "Unable to find " .. target
        end
    end)
end, {
    description = "Set hp of player",
    privs = {
        kick = true
```

```
            -- ^ probably better to register a custom priv
        }
    })
```

`ChatCmdBuilder.new(name, setup_func, def)` creates a new chat command called `name`. It then calls the function passed to it (`setup_func`), which then creates sub commands. Each `cmd:sub(route, func)` is a sub command.

A sub command is a particular response to an input param. When a player runs the chat command, the first sub command that matches their input will be run, and no others. If no subcommands match then the user will be told of the invalid syntax. For example, in the above code snippet if a player types something of the form `/sethp username 12` then the function passed to cmd:sub will be called. If they type `/sethp 12 bleh` then a wrong input message will appear.

`:name :hp:int` is a route. It describes the format of the param passed to /teleport.

Routes

A route is made up of terminals and variables. Terminals must always be there. For example, `join` in `/team join :username :teamname`. The spaces also count as terminals.

Variables can change value depending on what the user types. For example, `:username` and `:teamname`.

Variables are defined as `:name:type`. The `name` is used in the help documention. The `type` is used to match the input. If the type is not given, then the type is `word`.

Valid types are:

- `word` - default. Any string without spaces.
- `int` - Any integer/whole number, no decimals.
- `number` - Any number, including ints and decimals.
- `pos` - 1,2,3 or 1.1,2,3.4567 or (1,2,3) or 1.2, 2 ,3.2
- `text` - Any string. There can only ever be one text variable, no variables or terminals can come afterwards.

In `:name :hp:int`, there are two variables there:

- `name` - type of `word` as no type is specified. Accepts any string without spaces.
- `hp` - type of `int`

Subcommand functions

The first argument is the caller's name. The variables are then passed to the function in order.

```lua
cmd:sub(":target :hp:int", function(name, target, hp)
    -- subcommand function
end)
```

Installing ChatCmdBuilder

There are two ways to install:

1. Install ChatCmdBuilder as a mod and depend on it.
2. Include the init.lua file in ChatCmdBuilder as chatcmdbuilder.lua in your mod, and dofile it.

Admin complex command

Here is an example that creates a chat command that allows us to do this:

- `/admin kill <username>` - kill user
- `/admin move <username> to <pos>` - teleport user
- `/admin log <username>` - show report log
- `/admin log <username> <message>` - log to report log

```
local admin_log
local function load()
    admin_log = {}
end
local function save()
    -- todo
end
load()

ChatCmdBuilder.new("admin", function(cmd)
    cmd:sub("kill :name", function(name, target)
        local player = minetest.get_player_by_name(targe
        if player then
            player:set_hp(0)
            return true, "Killed " .. target
        else
            return false, "Unable to find " .. target
        end
    end)

    cmd:sub("move :name to :pos:pos", function(name, tar
        local player = minetest.get_player_by_name(targe
        if player then
            player:setpos(pos)
            return true, "Moved " .. target .. " to " .
        else
            return false, "Unable to find " .. target
        end
    end)

    cmd:sub("log :username", function(name, target)
        local log = admin_log[target]
        if log then
            return true, table.concat(log, "\n")
        else
            return false, "No entries for " .. target
        end
    end)
```

```lua
    cmd:sub("log :username :message", function(name, tar
        local log = admin_log[target] or {}
        table.insert(log, message)
        admin_log[target] = log
        save()
        return true, "Logged"
    end)
end, {
    description = "Admin tools",
    privs = {
        kick = true,
        ban = true
    }
})
```

Player Physics

Introduction

Player physics can be modified using physics overrides. Physics overrides can set the walking speed, jump speed and gravity constants. Physics overrides are set on a player by player basis, and are multipliers - a value of 2 for gravity would make gravity twice as strong.

- Basic Interface
- Your Turn

Basic Interface

Here is an example which adds an antigravity command, which puts the caller in low G:

```
minetest.register_chatcommand("antigravity", {
    func = function(name, param)
        local player = minetest.get_player_by_name(name
        player:set_physics_override({
            gravity = 0.1 -- set gravity to 10% of its 
                          -- (0.1 * 9.81)
        })
    end
})
```

Possible Overrides

player:set_physics_override() is given a table of overrides.
According to lua_api.txt, these can be:

- speed: multiplier to default walking speed value (default: 1)
- jump: multiplier to default jump value (default: 1)
- gravity: multiplier to default gravity value (default: 1)
- sneak: whether player can sneak (default: true)
- sneak_glitch: whether player can use the sneak glitch (default: true)

The sneak glitch allows the player to climb an ‘elevator’ made out of a certain placement of blocks by sneaking (pressing shift) and pressing space to ascend. It was originally a bug in Minetest, but was kept as it is used on many servers to get to higher levels. They added the option above so you can disable it.

Multiple mods

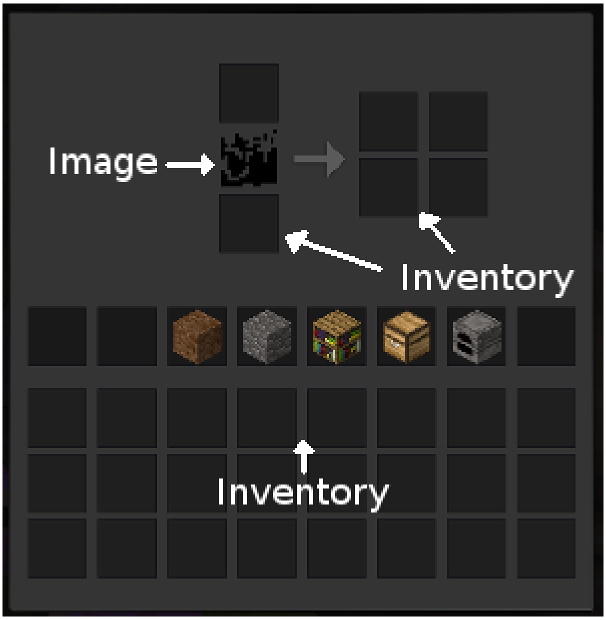
Please be warned that mods that override the same physics values of a player tend to be incompatible with each other. When setting an override, it overwrites any overrides that have been set before, by your or anyone else’s mod.

Your Turn

- **sonic**: Set the speed multiplayer to a high value (at least 6) when a player joins the game.
- **super bounce**: Increase the jump value so that the player can jump up 20 meters (1 meter is 1 block).
- **space**: Make the gravity decrease as the player gets higher and higher up.

Formspecs

Introduction



Screenshot of furnace formspec, labelled.

In this chapter we will learn how to create a formspec and display it to the user. A formspec is the specification code for a form. In Minetest, forms are windows like the Inventory which allow you to move your mouse and enter information. You should consider using Heads Up Display (HUD) elements if you do not need to get user input - notifications, for example - as unexpected windows tend to disrupt game play.

- Formspec syntax
- Displaying Forms
- Callbacks
- Contexts
- Node Meta Formspecs

Formspec Syntax

Formspecs have a rather weird syntax. They consist of a series of tags which are in the following form:

```
element_type[param1;param2;...]
```

Firstly the element type is declared, and then the attributes are given in square brackets.

(An element is an item such as a text box or button, or it is meta data such as size or background).

Here are two elements, of types foo and bar.

```
foo[param1]bar[param1]
```

Size[w, h]

Nearly all forms have a size tag. They are used to declare the size of the window required. **Forms don't use pixels as co-ordinates, they use a grid**, based on inventories. A size of (1, 1) means the form is big enough to host a 1x1 inventory. The reason this is used is because it is independent on screen resolution - The form should work just as well on large screens as small screens. You can use decimals in sizes and co-ordinates.

```
size[5,2]
```

Co-ordinates and sizes only use one attribute. The x and y values are separated by a comma, as you can see above.

Field[x, y; w, h; name; label; default]

This is a textbox element. Most other elements have a similar style of attributes. The "name" attribute is used in callbacks to get the submitted information. The others are pretty self-explaintary.

```
field[1,1;3,1;firstname;Firstname;]
```

It is perfectly valid to not define an attribute, like above.

Other Elements

You should look in lua_api.txt for a list of all possible elements, just search for "Formspec". It is near line 1019, at time of writing.

Displaying Formspecs

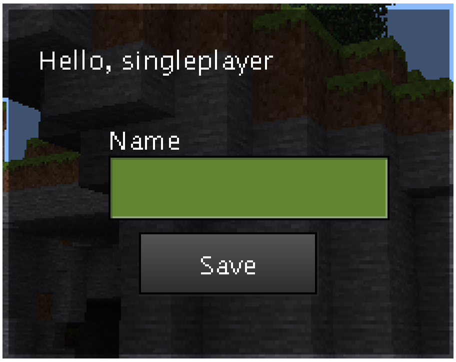
Here is a generalized way to show a formspec

```
minetest.show_formspec(playername, formname, formspec)
```

Formnames should be itemnames, however that is not enforced. There is no need to override a formspec here, formspecs are not registered like nodes and items are, instead the formspec code is sent to the player’s client for them to see, along with the formname. Formnames are used in callbacks to identify which form has been submitted, and see if the callback is relevant.

Example

```
-- Show form when the /forms
minetest.register_chatcommar
    func = function(name, pa
        minetest.show_formsp
                "size[4,3]"
                "label[0,0;H
                "field[1,1.5
                "button_exit
    end
})
```



The formspec generated by the example's code

The above example shows a formspec to a player when they use the /formspec command.

Note: the .. is used to join two strings together. The following two lines are equivalent:

```
"foobar"
"foo" .. "bar"
```

Callbacks

Let’s expand on the above example.

```
-- Show form when the /formspec command is used.
minetest.register_chatcommand("formspec", {
    func = function(name, param)
        minetest.show_formspec(name, "mymod:form",
                "size[4,3]" ..
                "label[0,0;Hello, " .. name .. "]" ..
                "field[1,1.5;3,1;name;Name;]" ..
                "button_exit[1,2;2,1;exit;Save]")
    end
})

-- Register callback
minetest.register_on_player_receive_fields(function(play
    if formname ~= "mymod:form" then
        -- Formname is not mymod:form,
        -- exit callback.
        return false
    end

    -- Send message to player.
    minetest.chat_send_player(player:get_player_name(),

    -- Return true to stop other minetest.register_on_p
    -- from receiving this submission.
    return true
end)
```

The function given in minetest.register_on_player_receive_fields is called everytime a user submits a form. Most callbacks will check the formname given to the function, and exit if it is not the right form. However, some callbacks may need to work on multiple forms, or all forms - it depends on what you want to do.

Fields

The fields parameter to the function is a table, index by string, of the values submitted by the user. You can access values in the table by doing fields.name, where ‘name’ is the name of the element.

As well as having the values of each element, you can also get which button was clicked. In this case, the button called ‘exit’ was clicked, so fields.exit will be true.

Some elements can submit the form without the user having to click a button, such as a check box. You can detect for these cases by looking for a clicked button.

```
-- An example of what fields could contain,
--   using the above code
{
    name = "Foo Bar",
    exit = true
}
```

Contexts

In quite a lot of cases you want your minetest.show_formspec to give information to the callback which you don’t want to have to send to the client. Information such as what a chat command was called with, or what the dialog is about.

Let’s say you are making a form to handle land protection information.

```lua
--
-- Step 1) set context when player requests the formspec
--

-- land_formspec_context[playername] gives the player's
local land_formspec_context = {}

minetest.register_chatcommand("land", {
    func = function(name, param)
        if param == "" then
            minetest.chat_send_player(name, "Incorrect p
            return
        end

        -- Save information
        land_formspec_context[name] = {id = param}

        minetest.show_formspec(name, "mylandowner:edit",
                "size[4,4]" ..
                "field[1,1;3,1;plot;Plot Name;]" ..
                "field[1,2;3,1;owner;Owner;]" ..
                "button_exit[1,3;2,1;exit;Save]")
    end
})



--
-- Step 2) retrieve context when player submits the form
--
minetest.register_on_player_receive_fields(function(play
    if formname ~= "mylandowner:edit" then
        return false
    end

    -- Load information
    local context = land_formspec_context[player:get_pla
    if context then
        minetest.chat_send_player(player:get_player_name
                fields.plot .. " and owned by " .. field

        -- Delete context if it is no longer going to be
        land_formspec_context[player:get_player_name()]

        return true
    else
        -- Fail gracefully if the context does not exist
        minetest.chat_send_player(player:get_player_name
    end
end)
```

Node Meta Formspecs

minetest.show_formspec is not the only way to show a formspec, you can also add formspecs to a node's meta data. This is used on nodes such as chests to allow for faster opening times - you don't need to wait for the server to send the player the chest formspec.

```
minetest.register_node("mymod:rightclick", {
    description = "Rightclick me!",
    tiles = {"mymod_rightclick.png"},
    groups = {cracky = 1},
    after_place_node = function(pos, placer)
        -- This function is run    when the chest node i
        -- The following code sets the formspec for ches
        -- Meta is a way of storing data onto a node.

        local meta = minetest.get_meta(pos)
        meta:set_string("formspec",
                "size[5,5]"..
                "label[1,1;This is shown on right click]
                "field[1,2;2,1;x;x;]")
    end,
    on_receive_fields = function(pos, formname, fields,
        if(fields.quit) then return end
        print(fields.x)
    end
})
```

Formspecs set this way do not trigger the same callback. In order to receive form input for meta formspecs, you must include an `on_receive_fields` entry when registering the node.

This style of callback can trigger the callback when you press enter in a field, which is impossible with `minetest.show_formspec`, however, this kind of form can only be shown by right-clicking on a node. It cannot be triggered programmatically.

HUD

Experimental Feature

The HUD feature will probably be rewritten in an upcoming Minetest release. Be aware that you may need to update your mods if the API is changed.

Introduction

Heads Up Display (HUD) elements allow you to show text, images, and other graphical elements.

HUD doesn’t accept user input. For that, you should use a Formspec.

- Basic Interface
- Positioning
- Text Elements
- Image Elements
- Other Elements

Basic Interface

HUD elements are created using a player object. You can get the player object from a username like this:

```lua
local player = minetest.get_player_by_name("username")
```

Once you have the player object, you can create an element:

```lua
local idx = player:hud_add({
         hud_elem_type = "text",
         position = {x = 1, y = 0},
         offset = {x=-100, y = 20},
         scale = {x = 100, y = 100},
         text = "My Text"
})
```

This attributes in the above table and what they do vary depending on the `hud_elem_type`.
A number is returned by the hud_add function which is needed to identify the HUD element at a later time, if you wanted to change or delete it.

You can change an attribute after creating a HUD element, such as what the text says:

```
player:hud_change(idx, "text", "New Text")
```

You can also delete the element:

```
player:hud_remove(idx)
```

Positioning

Screens come in different sizes, and HUD elements need to work well on all sizes. You locate an element using a combination of a position and an offset.

The position is a co-ordinate between (0, 0) and (1, 1) which determines where, relative to the screen width and height, the element goes. For example, an element with a position of (0.5, 0.5) will be in the center of the screen.

The offset applies a pixel offset to the position.
An element with a position of (0, 0) and an offset of (10, 10) will end up at the screen co-ordinates (0 * width + 10, 0 * height + 10).

Please note that offset scales to DPI and a user defined factor.

Text Elements

A text element is the simplest form of a HUD element.
Here is our earlier example, but with comments to explain each part:

```
local idx = player:hud_add({
    hud_elem_type = "text",      -- This is a text elemen
    position = {x = 1, y = 0},
    offset = {x=-100, y = 20},
    scale = {x = 100, y = 100}, -- Maximum size of text,
    text = "My Text"            -- The actual text show
})
```

Colors

You can apply colors to the text, using the `number` attribute. Colors are in Hexadecimal form.

```
local idx = player:hud_add({
    hud_elem_type = "text",
    position = {x = 1, y = 0},
    offset = {x=-100, y = 20},
    scale = {x = 100, y = 100},
    text = "My Text",
    number = 0xFF0000 -- Red
})
```

Image Elements

Displays an image on the HUD.

The X co-ordinate of the `scale` attribute is the scale of the image, with 1 being the original texture size. Negative values represent that percentage of the screen it should take; e.g. x=-100 means 100% (width).

Use `text` to specify the name of the texture.

Other Elements

Have a look at lua_api.txt for a complete list of HUD elements.

ItemStacks

Introduction

In this chapter you will learn how to use ItemStacks.

- Creating ItemStacks
- Name and Count
- Adding and Taking Items
- Wear
- Lua Tables
- Metadata
- More Methods

Creating ItemStacks

An item stack is a… stack of items. It's basically just an item type with a count of items in the stack.

You can create a stack like so:

```lua
local items = ItemStack("default:dirt")
local items = ItemStack("default:stone 99")
```

You could alternatively create a blank ItemStack and fill it using methods:

```lua
local items = ItemStack()
if items:set_name("default:dirt") then
    items:set_count(99)
else
    print("An error occured!")
end
```

And you can copy stacks like this:

```lua
local items2 = ItemStack(items)
```

Name and Count

```lua
local items = ItemStack("default:stone")
print(items:get_name()) -- default:stone
print(items:get_count()) -- 1

items:set_count(99)
print(items:get_name()) -- default:stone
print(items:get_count()) -- 99

if items:set_name("default:dirt") then
    print(items:get_name()) -- default:dirt
    print(items:get_count()) -- 99
else
    error("This shouldn't happen")
end
```

Adding and Taking Items

Adding

Use `add_item` to add items to an ItemStack. An ItemStack of the items that could not be added is returned.

```lua
local items = ItemStack("default:stone 50")
local to_add = ItemStack("default:stone 100")
local leftovers = items:add_item(to_add)

print("Could not add" .. leftovers:get_count() .. " of t
-- ^ will be 51
```

Taking

The following code takes **up to** 100. If there aren't enough items in the stack, it will take as much as it can.

```lua
local taken = items:take_item(100)
-- taken is the ItemStack taken from the main ItemStack
```

```lua
print("Took " .. taken:get_count() .. " items")
```

Wear

ItemStacks also have wear on them. Wear is a number out of 65535, the higher it is, the more wear.

You use `add_wear()`, `get_wear()` and `set_wear(wear)`.

```lua
local items = ItemStack("default:dirt")
local max_uses = 10

-- This is done automatically when you use a tool that d
-- It increases the wear of an item by one use.
items:add_wear(65535 / (max_uses - 1))
```

When digging a node, the amount of wear a tool gets may depends on the node being dug. So max_uses varies depending on what is being dug.

Lua Tables

```lua
local data = items:to_table()
local items2 = ItemStack(data)
```

Metadata

ItemStacks can also have a single field of metadata attached to them.

```lua
local meta = items:get_metadata()
print(dump(meta))
meta = meta .. " ha!"
items:set_metadata(meta)
-- if ran multiple times, would give " ha! ha! ha!"
```

More Methods

Inventories

Introduction

In this chapter you will learn how to use manipulate inventories, whether that is a player inventory, a node inventory, or a detached inventory. This chapter assumes that you already know how to create and manipulate ItemStacks.

- Basic Concepts.
- Types of Inventories.
    - Player Inventories.
    - Node Inventories.
    - Detached Inventories.
- InvRef and Lists.
    - Type of inventory.
    - List sizes.
    - List is empty.
    - Lua Tables.
    - Lua Tables for Lists.
- InvRef, Items and Stacks.
    - Adding to a list.
    - Checking for room.
    - Taking items.
    - Contains.
    - Manipulating Stacks.

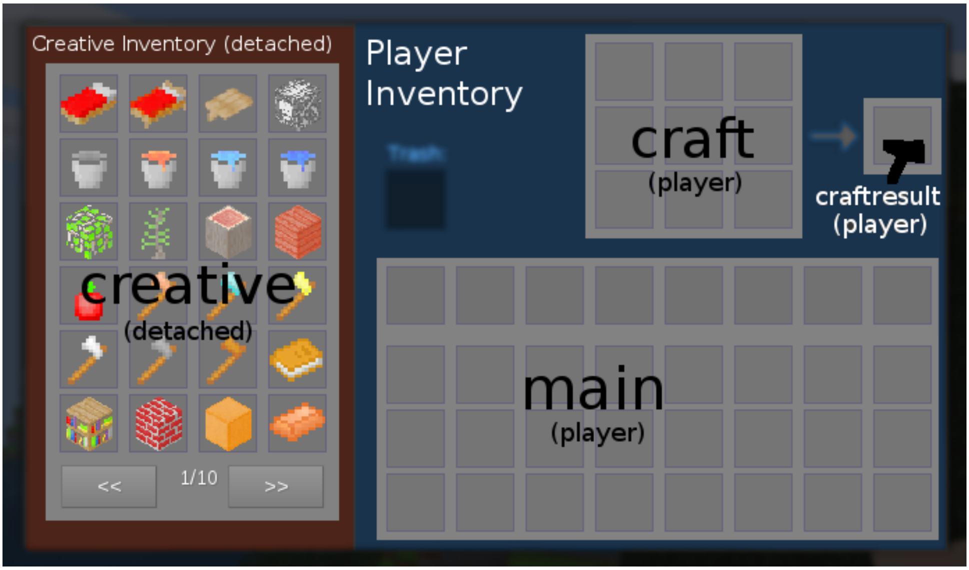
Basic Concepts

Components of an inventory:

- An **Inventory** is a collection of **Inventory List**s (also called a **list** when in the context of inventories).
- An **Inventory List** is an array of **slot**s. (By array, I mean a table indexed by numbers).
- A **slot** is a place a stack can be - there may be a stack there or there may not.
- An **InvRef** is an object that represents an inventory, and has functions to manipulate it.

There are three ways you can get inventories:

- **Player Inventories** - an inventory attached to a player.
- **Node Inventories** - an inventory attached to a node.
- **Detached Inventories** - an inventory which is not attached to a node or player.



This image shows the two inventories visible when you press i. The gray boxes are inventory lists.
The creative inventory, left (in red) is detached and it made up of a single list.
The player inventory, right (in blue) is a player inventory and is made up of three lists.
Note that the trash can is a formspec element, and is not part of the inventory.

Types of Inventories

There are three types of Inventories.

Player Inventories.

This is what you see when you press i. A player inventory usually has two grids, one for the main inventory, one for crafting.

```
local inv = minetest.get_inventory({type="player", name=
```

Node Inventories.

An inventory related to a position, such as a chest. The node must be loaded, as it’s stored in Node Metadata.

```
local inv = minetest.get_inventory({type="node", pos={x=
```

Detached Inventories

A detached inventory is independent of players and nodes. One example of a detached inventory is the creative inventory is detached, as all players see the same inventory. You may also use this if you want multiple chests to share the same inventory.

This is how you get a detached inventory:

```
local inv = minetest.get_inventory({type="detached", nar
```

And this is how you can create one:

```
minetest.create_detached_inventory("inventory_name", cal
```

Creates a detached inventory. If it already exists, it is cleared. You can supply a table of callbacks.

InvRef and Lists

Type of Inventory

You can check where the inventory is from by doing:

```
local location = inv:get_location()
```

It will return a table like the one passed to `minetest.get_inventory()`.

If the location is unknown, `{type="undefined"}` is returned.

List sizes

Inventory lists have a size, for example `main` has size of 32 slots by default. They also have a width, which is used to divide them into a grid.

```lua
if inv:set_size("main", 32) then
    inv:set_width("main", 8)
    print("size:  " .. inv.get_size("main"))
    print("width: " .. inv:get_width("main"))
else
    print("Error!")
end
```

List is empty

```lua
if inv:is_empty("main") then
    print("The list is empty!")
end
```

Lua Tables

You can convert an inventory to a Lua table using:

```lua
local lists = inv:get_lists()
```

It will be in this form:

```lua
{
    list_one = {
        ItemStack,
        ItemStack,
        ItemStack,
        ItemStack,
        -- inv:get_size("list_one") elements
    },
    list_two = {
        ItemStack,
        ItemStack,
        ItemStack,
        ItemStack,
        -- inv:get_size("list_two") elements
    }
}
```

You can then set an inventory like this:

```
inv:set_lists(lists)
```

Please note that the sizes of lists will not change.

Lua Tables for Lists

You can do the same as above, but for individual lists

```
local list = inv:get_list("list_one")
```

It will be in this form:

```
{
    ItemStack,
    ItemStack,
    ItemStack,
    ItemStack,
    -- inv:get_size("list_one") elements
}
```

You can then set the list like this:

```
inv:set_list("list_one", list)
```

Please note that the sizes of lists will not change.

InvRef, Items and Items

Adding to a list

```
local stack = ItemStack("default:stone 99")
local leftover = inv:add_item("main", stack)
if leftover:get_count() > 0 then
    print("Inventory is full! " .. leftover:get_count()
end
```

`"main"` is the name of the list you're adding to.

Checking for room

```lua
if not inv:room_for_item("main", stack) then
    print("Not enough room!")
end
```

Taking items

```lua
local taken = inv:remove_item("main", stack)
print("Took " .. taken:get_count())
```

Contains

This works if the item count is split up over multiple stacks, for example looking for "default:stone 200" will work if there are stacks of 99 + 95 + 6.

```lua
if not inv:contains_item(listname, stack) then
    print("Item not in inventory!")
end
```

Manipulating Stacks

Finally, you can manipulate individual stacks like so:

```lua
local stack = inv:get_stack(listname, 0)
inv:set_stack(listname, 0, stack)
```

Releasing a Mod

Introduction

In this chapter we will find out how to publish a mod so that other users can use it.

- License Choices
- Packaging
- Uploading
- Forum Topic

Before you release your mod, there are some things to think about:

- Is there another mod that does the same thing? If so, how does yours differ or improve on it?
- Is your mod useful?

License Choices

You need to specify a license for your mod. **Public domain is not a valid licence**, as the definition varies in different countries.

First thing you need to note is that your code and your art need different things from the license they use. Creative Commons licenses shouldn’t be used with source code, but rather with artistic works such as images, text and meshes.

You are allowed any license, however mods which disallow derivatives are banned from the forum. (Other developers must be able to take your mod, modify it, and release it again.)

LGPL and CC-BY-SA

This is a common license combination in the Minetest community, as it is what Minetest and minetest_game use. You license your code under LGPL 2.1 and your art under CC-BY-SA.

- Anyone can modify, redistribute and sell modified or unmodified versions.
- If someone modifies your mod, they must give their version the same license.
- Your copyright notice must be kept.

Add this copyright notice to your README.txt, or as a new file called LICENSE.txt

```
License for Code
----------------

Copyright (C) 2010-2013 Your Name <emailaddress>

This program is free software; you can redistribute it a
it under the terms of the GNU Lesser General Public Lice
the Free Software Foundation; either version 2.1 of the
(at your option) any later version.

This program is distributed in the hope that it will be
but WITHOUT ANY WARRANTY; without even the implied warra
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  Se
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser Genera
with this program; if not, write to the Free Software Fo
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 U

License for Textures, Models and Sounds
---------------------------------------

CC-BY-SA 3.0 UNPORTED. Created by Your Name
```

WTFPL or CC0

These licenses allows anyone to do what they want with your mod. Modify, redistribute, sell, leave out attribution. They can be used for both code and art.

Packaging

There are some files that we recommend you include in your mod when you release it.

README.txt

You should provide a readme file. This should state:

- What the mod does.
- What the license is.
- Current version of mod.
- How to install the mod.
- What dependencies there are / what the user needs to install.
- Where to report problems/bugs or get help.

See appendix for an example and a generator

description.txt

Write a sentence or two explaining what your mod does. Be concise without being too vague. This is displayed in the mod store.

For example:

```
GOOD: Adds soups, cakes, bakes and juices. The food mod
BAD:  The food mod for Minetest.
```

screenshot.png

Screenshots should be 3:2 (3 pixels of width for every 2 pixels of height) and a minimum size of 300 x 200px. This is displayed in the mod store.

Uploading

In order for a potential user to download your mod, you need to upload it to somewhere which is publicly accessible.
I will outline several methods you can use, but really you should use the one that works best for you, as long as it mets these requirements:
(and any other requirements which may be added by forum moderators)

- **Stable** - the hosting website should not just shutdown randomly.
- **Direct link** - you should be able to click a link on the forum and download the file, without having to view another page.
- **Virus Free** - pretty obvious.

Github, or another VCS

It is recommended that you use a Version Control System for the following reasons:

- Allows other developers to submit changes (easily).
- Allows the code to be previewed before downloading.
- Allows users to submit bug reports.

However, such systems may be hard to understand when you first start out.

The majority of Minetest developers use GitHub as a website to host their code, however that doesn't matter that much.

- Using Git - Basic concepts. Using the command line.
- GitHub for Windows - Use a graphical interface on Windows to upload your code.

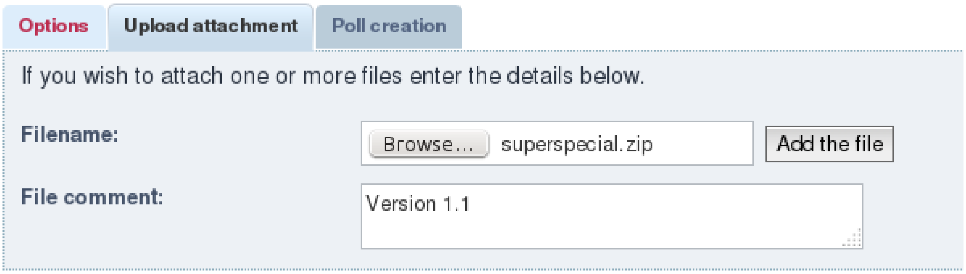
Forum Attachments

You could use forum attachments instead. This is done when creating a mod's topic - covered below.

First, you need to zip the files into a single file. This varies from operating system to operating system.

On Windows, go to the mod's folder. Select all the files. Right click, Send To > Compressed (zipped) folder. Rename the resulting zip file to the name of your modfolder.

On the create a topic page, see below, go to the "Upload Attachment" tab at the bottom. Click browse and select the zipped file. I suggest that you enter the version of your mod in the comment field.



If you wish to attach one or more files enter the details below.

Filename: superspecial.zip

File comment: Version 1.1

Upload Attachment tab.

Forum Topic

You can now create a forum topic. You should create it in the “WIP Mods” (Work In Progress) forum.
When you consider your mod no longer a work in progress, you can request that it be moved to “Mod Releases.”

Content

The requirements of a forum topic are mostly the same as what is recommended for a README.txt

- What the mod does.
- What the license is.
- Current version of mod.
- How to install the mod.
- What dependencies there are.
- Where to report problems/bugs or get help.
- Link to download, or an attachment.

You should also include screenshots of your mod in action, if relevant.

Here is an example. The Minetest forum uses bbcode for formating.

```
Adds magic, rainbows and other special things.

See download attached.

[b]Version:[/b] 1.1
[b]License:[/b] LGPL 2.1 or later

Dependencies: default mod (found in minetest_game)

Report bugs or request help on the forum topic.

[h]Installation[/h]

Unzip the archive, rename the folder to to modfoldername
place it in minetest/mods/minetest/

(  GNU/Linux: If you use a system-wide installation plac
    it in ~/.minetest/mods/minetest/.  )

(  If you only want this to be used in a single world, p
    the folder in worldmods/ in your worlddirectory.  )

For further information or help see:
[url]http://wiki.minetest.com/wiki/Installing_Mods[/url]
```

If you modify the above example for your mod topic, remember to change “modfldername” to the name of the folder your mod should be in.

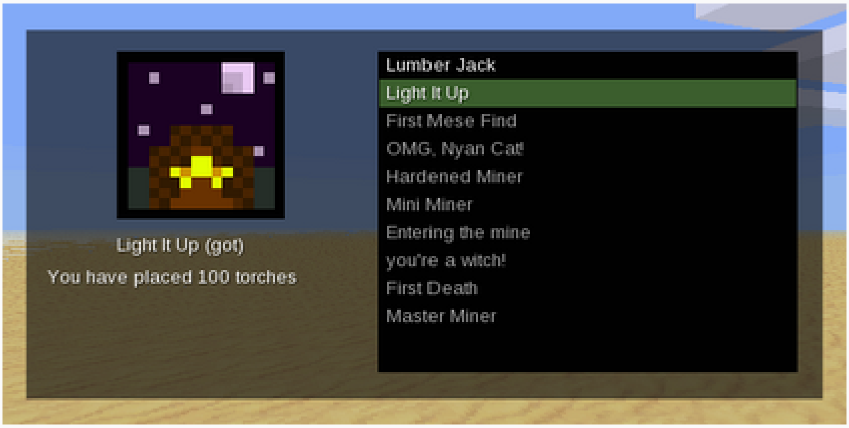
Title

Subject of topic must be in one of these formats:

- [Mod] Mod Title [modname]
- [Mod] Mod Title [version number] [modname]
- eg: [Mod] More Blox [0.1] [moreblox]

Profit

[Mod] Super Special Mod [superspecial]
by rubenwardy » Sun Jan 11, 2015 8:30 pm



Adds magic, rainbows and other special things.

Download is attached.

License: WTFPL
Dependencies: Default (found in minetest_game)

Installation

Unzip the archive, rename the folder to to modfoldername and place it in minetest/mods/minetest/

(Linux: If you have a linux system-wide installation place it in ~/.minetest/mods/minetest/.)

(If you only want this to be used in a single world, place the folder in worldmods/ in your worlddirectory.)

Appendix: Readme and Forum Generator

Title:	My Super Special Mod
Modname:	mysuperspecial
Description:	Adds magic, rainbows and o
Version:	1.1
License:	LGPL 2.1 or later
Dependencies:	default mod (found in minete

Download: http://example.com/downloac
Additional:

Report bugs or
request help on the

```
My Super Special Mod
====================

Adds magic, rainbows and other special things.

Version: 1.1
License: LGPL 2.1 or later
Dependencies: default mod (found in minetest_game)

Report bugs or request help on the forum topic.

Installation
------------

Unzip the archive, rename the folder to mysuperspecial a
place it in minetest/mods/

(  GNU/Linux: If you use a system-wide installation plac
    it in ~/.minetest/mods/.  )

(  If you only want this to be used in a single world, p
    the folder in worldmods/ in your worlddirectory.  )

For further information or help see:
http://wiki.minetest.com/wiki/Installing_Mods
```

```
Adds magic, rainbows and other special things.

[b]Version:[/b] 1.1
[b]License:[/b] LGPL 2.1 or later
[b]Dependencies:[/b] default mod (found in minetest_game
[b]Download:[/b] http://example.com/download.zip

Report bugs or request help on the forum topic.

[h]Installation[/h]
```

```
Unzip the archive, rename the folder to mysuperspecial a
place it in minetest/mods/

(  GNU/Linux: If you use a system-wide installation plac
    it in ~/.minetest/mods/.  )

(  If you only want this to be used in a single world, p
    the folder in worldmods/ in your worlddirectory.  )

For further information or help see:
http://wiki.minetest.com/wiki/Installing_Mods
```

Read More

List of Resources

After you’ve read this book, take a look at the following

Minetest Modding

- Minetest’s Lua API Reference - HTML version | Text version.
- Explore the Developer Wiki.
- Look at existing mods.

Lua Programming

- Programming in Lua (PIL).
- Lua Crash Course.

3D Modelling

- Blender 3D: Noob to pro.
- Using Blender with Minetest.

Lua Scripts

Introduction

In this chapter we will talk about scripting in Lua, the tools required, and go over some techniques which you will probably find useful.

Tools

A text editor with code highlighting is sufficient for writing scripts in Lua. Code highlighting gives different colors to different words and characters depending on what they mean. This allows you to spot mistakes.

```lua
function ctf.post(team,msg)
    if not ctf.team(team) then
        return false
    end
    if not ctf.team(team).log then
        ctf.team(team).log = {}
    end

    table.insert(ctf.team(team).log,1,msg)
    ctf.save()

    return true
end
```

For example, keywords in the above snippet are highlighted such as if, then, end, return. table.insert is a function which comes with Lua by default.

Recommended Editors

Other editors are available, of course.

- Windows: Notepad++, Atom
- Linux: Kate, Gedit, Atom
- OSX: Atom

Integrated Programming Environments

IDEs allow you to debug code like a native application. These are harder to set up than just a text editor.

One such IDE is Eclipse with the Koneki Lua plugin:

- Install Eclipse + Koneki.
- Create a new Lua project from existing source (specify Minetest’s base directory).
- Follow instructions from Koneki wiki how to do “Attach to remote Application” debugging (just a few steps).
- It is suggested to add those lines from wiki at beginning of builtin.lua.
- Start the debugger (set “Break on first line” in debugger configuration to see if it is working).
- Start Minetest.
- Enter the game to startup Lua.

Coding in Lua

This section is a Work in Progress. May be unclear.

Programs are a series of commands that run one after another. We call these commands “statements.”

Program flow is important, it allows you to direct or skip over statements. There are three main types of flow:

- Sequence: Just run one statement after another, no skipping.
- Selected: Skip over statements depending on conditions.
- Iteration: Repeating, looping. Keep running the same statements until a condition is met.

So, what do statements in Lua look like?

```lua
local a = 2      -- Set 'a' to 2
local b = 2      -- Set 'b' to 2
local result = a + b -- Set 'result' to a + b, which is
a = a + 10
print("Sum is "..result)
```

Woah, what happened there? a, b, and result are **variables**. They're like what you get in mathematics, A = w * h. The equals signs are **assignments**, so "result" is set to a + b. Variable names can be longer than one character unlike in mathematics, as seen with the "result" variable. Lua is **case sensitive**. A is a different variable than a.

The word "local" before they are first used means that they have local scope, I'll discuss that shortly.

Variable Types

Type	Description	Example
Integer	Whole number	local A = 4
Float	Decimal	local B = 3.2, local C = 5 / 2
String	A piece of text	local D = "one two three"
Boolean	True or False	local is_true = false, local E = (1 == 1)
Table	Lists	Explained below
Function	Can run. May require inputs and may return a value	local result = func(1, 2, 3)

Not an exhaustive list. Doesn't contain every possible type.

Arithmetic Operators

Symbol	Purpose	Example
A + B	Addition	2 + 2 = 4
A - B	Subtraction	2 - 10 = -8
A * B	Multiplication	2 * 2 = 4
A / B	Division	100 / 50 = 2
A ^ B	Powers	2 ^ 2 = 2^2 = 4
A .. B	Join strings	"foo" .. "bar" = "foobar"

A string in programming terms is a piece of text.

Not an exhaustive list. Doesn’t contain every possible operator.

Selection

The most basic selection is the if statement. It looks like this:

```
local random_number = math.random(1, 100) -- Between 1 a

if random_number > 50 then
    print("Woohoo!")
else
    print("No!")
end
```

That example generates a random number between 1 and 100. It then prints “Woohoo!” if that number is bigger than 50, otherwise it prints “No!”. What else can you get apart from ‘>’?

Logical Operators

Symbol	Purpose	Example
A == B	Equals	1 == 1 (true), 1 == 2 (false)
A ~= B	Doesn’t equal	1 ~= 1 (false), 1 ~= 2 (true)
A > B	Greater than	5 > 2 (true), 1 > 2 (false), 1 > 1 (false)
A < B	Less than	1 < 3 (true), 3 < 1 (false), 1 < 1 (false)
A >= B	Greater than or equals	5 >= 5 (true), 5 >= 3 (true), 5 >= 6 (false)
A <= B	Less than or equals	3 <= 6 (true), 3 <= 3 (true)
A and B	And (both must be correct)	(2 > 1) and (1 == 1) (true), (2 > 3) and (1 == 1) (false)
A or B	either or. One or both must be true.	(2 > 1) or (1 == 2) (true), (2 > 4) or (1 == 3) (false)
not A	not true	not (1 == 2) (true), not (1 == 1) (false)

That doesn’t contain every possible operator, and you can combine operators like this:

```
if not A and B then
    print("Yay!")
end
```

Which prints “Yay!” if A is false and B is true.

Logical and arithmetic operators work exactly the same, they both accept inputs and return a value which can be stored.

```
local A = 5
local is_equal = (A == 5)

if is_equal then
    print("Is equal!")
end
```

Programming

Programming is the action of talking a problem, such as sorting a list of items, and then turning it into steps that a computer can understand.

Teaching you the logical process of programming is beyond the scope of this book; however, the following websites are quite useful in developing this:

Codecademy

Codecademy is one of the best resources for learning to ‘code’, it provides an interactive tutorial experience.

Scratch

Scratch is a good resource when starting from absolute basics, learning the problem solving techniques required to program.
Scratch is **designed to teach children** how to program, it isn’t a serious programming language.

Local and Global

Whether a variable is local or global determines where it can be written to or read to. A local variable is only accessible from where it is defined. Here are some examples:

```lua
-- Accessible from within this script file
local one = 1

function myfunc()
    -- Accessible from within this function
    local two = one + one

    if two == one then
        -- Accessible from within this if statement
        local three = one + two
    end
end
```

Whereas global variables can be accessed from anywhere in the script file, and from any other mod.

```lua
my_global_variable = "blah"

function one()
    my_global_variable = "three"
end

print(my_global_variable) -- Output: "blah"
one()
print(my_global_variable) -- Output: "three"
```

Locals should be used as much as possible

Lua is global by default (unlike most other programming languages). Local variables must be identified as such.

```lua
function one()
    foo = "bar"
end

function two()
    print(dump(foo))  -- Output: "bar"
end

one()
two()
```

dump() is a function that can turn any variable into a string so the programmer can see what it is. The foo variable will be printed as “bar”, including the quotes which show it is a string.

This is sloppy coding, and Minetest will in fact warn you about this:

```
[WARNING] Assigment to undeclared global 'foo' inside fu
```

To correct this, use “local”:

```
function one()
    local foo = "bar"
end

function two()
    print(dump(foo))  -- Output: nil
end

one()
two()
```

Nil means **not initalised**. The variable hasn’t been assigned a value yet, doesn’t exist, or has been uninitialised. (ie: set to nil)

The same goes for functions. Functions are variables of a special type. You should make functions as local as much as possible, as other mods could have functions of the same name.

```
local function foo(bar)
    return bar * 2
end
```

If you want your functions to be accessible from other scripts or mods, it is recommended that you add them all into a table with the same name as the mod:

```
mymod = {}

function mymod.foo(bar)
    return "foo" .. bar
end

-- In another mod, or script:
mymod.foo("foobar")
```

Including other Lua Scripts

You can include Lua scripts from your mod or another mod like this:

```
dofile(minetest.get_modpath("modname") .. "/script.lua"
```

“local” variables declared outside of any functions in a script file will be local to that script. You won’t be able to access them from any other scripts.

As for how you divide code up into files, it doesn’t matter that much. The most important thing is that your code is easy to read and edit. You won’t need to use it for smaller projects.

Nodes, Items, and Crafting

Introduction

In this chapter we will learn how to register a new node or craftitem, and create craft recipes.

- Item Strings
- Textures
- Registering a Craftitem
- Foods
- Registering a basic Node
- Crafting
- Groups

Item Strings

Each item, whether that be a node, craftitem, tool, or entity, has an item string.
This is sometimes referred to as registered name or just name. A string in programming terms is a piece of text.

```
modname:itemname
```

The modname is the name of the folder your mod is in. You may call the itemname anything you like; however, it should be relevant to what the item is and it can’t already be registered.

Overriding

Overriding allows you to:

- Redefine an existing item.
- Use an item string with a different modname.

To override, you prefix the item string with a colon, `:`. Declaring an item as `:default:dirt` will override the default:dirt in the default mod.

Textures

Textures are usually 16 by 16 pixels. They can be any resolution, but it is recommended that they are in the order of 2 (eg, 16, 32, 64, 128, etc), as other resolutions may not be supported correctly on older devices.

Textures should be placed in textures/. Their name should match `modname_itemname.png`.
JPEGs are supported, but they do not support transparency and are generally bad quality at low resolutions.

Registering a Craftitem

Craftitems are the simplest items in Minetest. Craftitems cannot be placed in the world. They are used in recipes to create other items, or they can be used by the player, such as food.

```
minetest.register_craftitem("mymod:diamond_fragments", 
    description = "Alien Diamond Fragments",
    inventory_image = "mymod_diamond_fragments.png"
})
```

Item definitions like the one seen above are usually made up of a unique item string and a definition table. The definition table contains attributes which affect the behaviour of the item.

Foods

Foods are items that cure health. To create a food item you need to define the on_use property like this:

```
minetest.register_craftitem("mymod:mudpie", {
    description = "Alien Mud Pie",
    inventory_image = "myfood_mudpie.png",
    on_use = minetest.item_eat(20)
})
```

The number supplied to the minetest.item_eat function is the number of hit points that are healed by this food. Two hit points make one heart and because there are 10 hearts there are 20 hitpoints. Hitpoints don’t have to be integers (whole numbers), they can be decimals.

Sometimes you may want a food to be replaced with another item when being eaten, for example smaller pieces of cake or bones after eating meat. To do this, use:

```
minetest.item_eat(hp, replace_with_item)
```

Where replace_with_item is an item string.

Foods, extended

How about if you want to do more than just eat the item, such as send a message to the player?

```
minetest.register_craftitem("mymod:mudpie", {
    description = "Alien Mud Pie",
    inventory_image = "myfood_mudpie.png",
    on_use = function(itemstack, user, pointed_thing)
        hp_change = 20
        replace_with_item = nil

        minetest.chat_send_player(user:get_player_name(

        -- Support for hunger mods using minetest.regis
        for _ , callback in pairs(minetest.registered_o
            local result = callback(hp_change, replace_
            if result then
                return result
            end
        end

        if itemstack:take_item() ~= nil then
            user:set_hp(user:get_hp() + hp_change)
        end

        return itemstack
    end
})
```

If you are creating a hunger mod, or if you are affecting foods outside of your mod, you should consider using minetest.register_on_item_eat

Registering a basic node

In Minetest, a node is an item that you can place. Most nodes are 1m x 1m x 1m cubes; however, the shape doesn't have to be a cube - as we will explore later.

Let's get onto it. A node's definition table is very similar to a craftitem's definition table; however, you need to set the textures for the faces of the cube.

```
minetest.register_node("mymod:diamond", {
    description = "Alien Diamond",
    tiles = {"mymod_diamond.png"},
    is_ground_content = true,
    groups = {cracky=3, stone=1}
})
```

Let's ignore `groups` for now, and take a look at the tiles. The `tiles` property is a table of texture names the node will use. When there is only one texture, this texture is used on every side.

What if you would like a different texture for each side? Well, you give a table of 6 texture names, in this order:
up (+Y), down (-Y), right (+X), left (-X), back (+Z), front (-Z). (+Y, -Y, +X, -X, +Z, -Z)

Remember: +Y is upwards in Minetest, along with most video games. A plus direction means that it is facing positive co-ordinates, a negative direction means that it is facing negative co-ordinates.

```
minetest.register_node("mymod:diamond", {
    description = "Alien Diamond",
    tiles = {
        "mymod_diamond_up.png",
        "mymod_diamond_down.png",
        "mymod_diamond_right.png",
        "mymod_diamond_left.png",
        "mymod_diamond_back.png",
        "mymod_diamond_front.png"
    },
    is_ground_content = true,
    groups = {cracky = 3},
    drop = "mymod:diamond_fragments"
    -- ^  Rather than dropping diamond, drop mymod:diam
})
```

The is_ground_content attribute allows caves to be generated over the stone.

Crafting

There are several different types of crafting, identified by the `type` property.

- shaped - Ingredients must be in the correct position.
- shapeless - It doesn't matter where the ingredients are, just that there is the right amount.
- cooking - Recipes for the furnace to use.
  - fuel - Defines items which can be burned in furnaces.
- tool_repair - Used to allow the repairing of tools.

Craft recipes do not use Item Strings to uniquely identify themselves.

Shaped

Shaped recipes are the normal recipes - the ingredients have to be in the right place. For example, when you are making a pickaxe the ingredients have to be in the right place for it to work.

```
minetest.register_craft({
    output = "mymod:diamond_chair 99",
    recipe = {
        {"mymod:diamond_fragments", "", ""},
        {"mymod:diamond_fragments", "mymod:diamond_fragr
        {"mymod:diamond_fragments", "mymod:diamond_fragr
    }
})
```

This is pretty self-explanatory. You don't need to define the type, as shaped crafts are default. The 99 after the itemname in output makes the craft create 99 chairs rather than one.

If you notice, there is a blank column at the far end. This means that the craft must always be exactly that. In most cases, such as the door recipe, you don't care if the ingredients are always in an exact place, you just want them correct relative to each other. In order to do this, delete any empty rows and columns. In the above case, there is an empty last column, which, when removed, allows the recipe to be crafted if it was all moved one place to the right.

```
minetest.register_craft({
    output = "mymod:diamond_chair",
    recipe = {
        {"mymod:diamond_fragments", ""},
        {"mymod:diamond_fragments", "mymod:diamond_fragm
        {"mymod:diamond_fragments", "mymod:diamond_fragm
    }
})
```

Shapeless

Shapeless recipes are a type of recipe which is used when it doesn’t matter where the ingredients are placed, just that they’re there. For example, when you craft a bronze ingot, the steel and the copper do not need to be in any specific place for it to work.

```
minetest.register_craft({
    type = "shapeless",
    output = "mymod:diamond",
    recipe = {"mymod:diamond_fragments", "mymod:diamond_
})
```

When you are crafting the diamond, the three diamond fragments can be anywhere in the grid.
Note: You can still use options like the number after the result, as mentioned earlier.

Cooking

Recipes with the type “cooking” are not made in the crafting grid, but are cooked in furnaces, or other cooking tools that might be found in mods. For example, you use a cooking recipe to turn ores into bars.

```
minetest.register_craft({
    type = "cooking",
    output = "mymod:diamond_fragments",
    recipe = "default:coalblock",
    cooktime = 10,
})
```

As you can see from this example, the only real difference in the code is that the recipe is just a single item, compared to being in a table (between braces). They also have an optional “cooktime” parameter which defines how long the item takes to cook. If this is not set it defaults to 3.

The recipe above works when the coal block is in the input slot, with some form of a fuel below it. It creates diamond fragments after 10 seconds!

Fuel

This type is an accompaniment to the cooking type, as it defines what can be burned in furnaces and other cooking tools from mods.

```
minetest.register_craft({
    type = "fuel",
    recipe = "mymod:diamond",
    burntime = 300,
})
```

They don’t have an output like other recipes, but they have a burn time which defines how long they will last as fuel in seconds. So, the diamond is good as fuel for 300 seconds!

Groups

Items can be members of many groups and groups can have many members. Groups are usually identified using `group:group_name` There are several reasons you use groups.

Groups can be used in crafting recipes to allow interchangeability of ingredients. For example, you may use group:wood to allow any wood item to be used in the recipe.

Dig types

Let’s look at our above `mymod:diamond` definition. You’ll notice this line:

```
groups = {cracky = 3}
```

Cracky is a digtype. Dig types specify what type of the material the node is physically, and what tools are best to destroy it.

Group	Description

Group	Description
crumbly	dirt, sand
cracky	tough but crackable stuff like stone.
snappy	something that can be cut using fine tools; e.g. leaves, smallplants, wire, sheets of metal
choppy	something that can be cut using force; e.g. trees, wooden planks
fleshy	Living things like animals and the player. This could imply some blood effects when hitting.
explody	Especially prone to explosions
oddly_breakable_by_hand	Torches, etc, quick to dig

Creating Textures

Introduction

In this chapter we will learn how to create and optimise textures for Minetest. We will use techniques relevant for pixel art.

- Resources
- Editors
- Common Mistakes

Resources

- 16×16 Pixel Art Tutorial

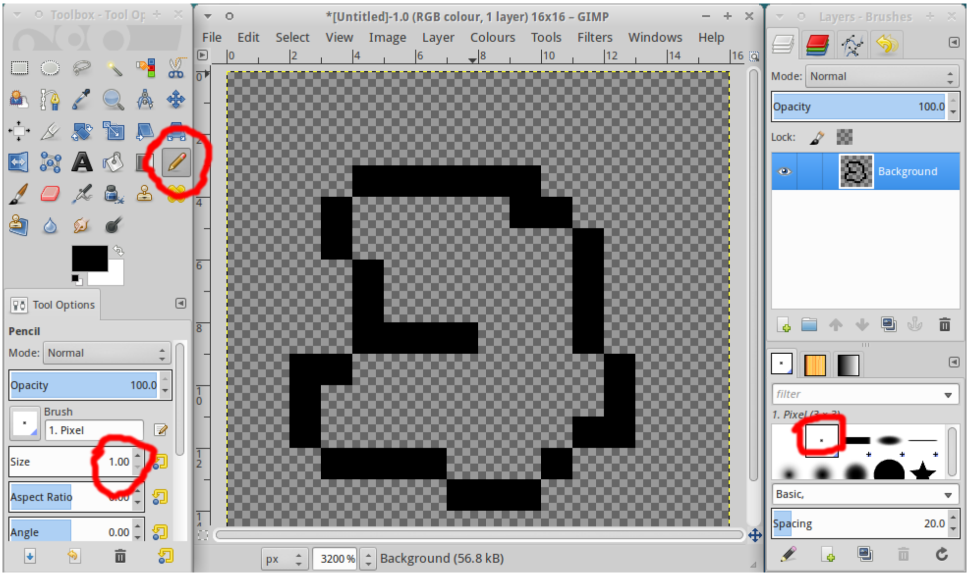
About MS Paint

You need to be aware that MS Paint does not support transparency. This won't matter if you're making textures for the side of nodes, but generally you need transparency for craft items, etc.

Editing in GIMP

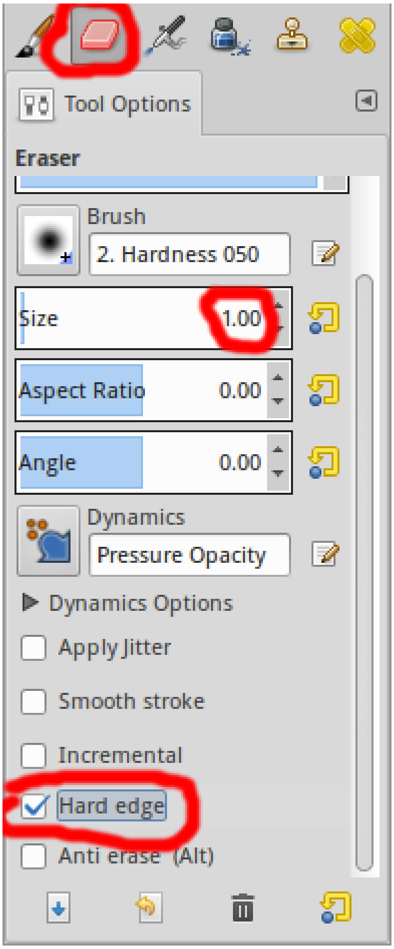
GIMP is commonly used in the Minetest community. It has quite a high learning curve as lots of its features are hidden away.

Use the pencil tool to edit individual pixels



Pencil in GIMP

Set the rubber to hard edge



Tool Options

Eraser

Brush: 2. Hardness 050

Size 1.00

Aspect Ratio 0.00

Angle 0.00

Dynamics: Pressure Opacity

Dynamics Options

☐ Apply Jitter

☐ Smooth stroke

☐ Incremental

☑ Hard edge

☐ Anti erase (Alt)

Rubber in GIMP

Common Mistakes

Blurred textures through usage of incorrect tools

For the most part, you want to manipulate pixels on an individual basis. The tool for this in most editors is the pencil tool.

Node Drawtypes

This chapter is incomplete

Some drawtypes have not been explained yet, and placeholder images are being used.

Introduction

In this chapter we explain all the different types of node drawtypes there are.

First of all, what is a drawtype? A drawtype defines how the node is to be drawn. A torch looks different to water, water looks different to stone.

The string you use to determine the drawtype in the node definition is the same as the title of the sections, except in lower case.

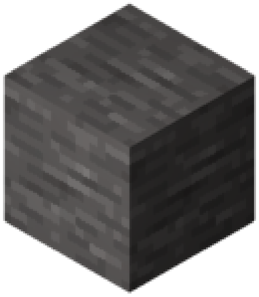
- Normal
- Airlike
- Liquid
    - FlowingLiquid
- Glasslike
- Glasslike_Framed
    - Glasslike_Framed_Optional
- Allfaces
    - Allfaces_Optional
- Torchlike
- Nodebox

This article is not complete yet. These drawtypes are missing:

- Signlike
- Plantlike
- Firelike
- Fencelike
- Raillike
- Mesh

Normal

This is, well, the normal drawtypes. Nodes that use this will be cubes with textures for each side, simple-as.
Here is the example from the Nodes, Items and Crafting chapter. Notice how you don't need to declare the drawtype.


Normal Drawtype

```
minetest.register_node("mymod:diamond
    description = "Alien Diamond",
    tiles = {
        "mymod_diamond_up.png",
        "mymod_diamond_down.png",
        "mymod_diamond_right.png",
        "mymod_diamond_left.png",
        "mymod_diamond_back.png",
        "mymod_diamond_front.png"
    },
    is_ground_content = true,
    groups = {cracky = 3},
    drop = "mymod:diamond_fragments"
})
```

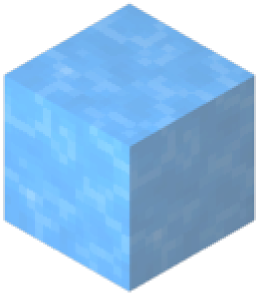
Airlike

These nodes are see through and thus have no textures.

```
minetest.register_node("myair:air", {
    description = "MyAir (you hacker you!)",
    drawtype = "airlike",

    paramtype = "light",
    -- ^ Allows light to propagate through the node with
    --   light value falling by 1 per node.

    sunlight_propagates = true, -- Sunlight shines throu
    walkable     = false, -- Would make the player colli
    pointable    = false, -- You can't select the node
    diggable     = false, -- You can't dig the node
    buildable_to = true,  -- Nodes can be replace this n
                          -- (you can place a node and r
                          -- that used to be there)

    air_equivalent = true,
    drop = "",
    groups = {not_in_creative_inventory=1}
})
```

Liquid

These nodes are complete liquid nodes, the liquid flows outwards from position using the flowing liquid drawtype. For each liquid node you should also have a flowing liquid node.


Liquid Drawtype

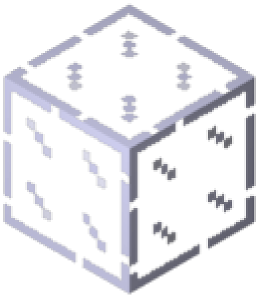
```
-- Some properties have been removed
minetest.register_node("default:water
    drawtype = "liquid",
    paramtype = "light",

    inventory_image = minetest.invent
    -- ^ this is required to stop the

    tiles = {
        {
            name = "default_water_sou
            animation = {
                type     = "vertical_
                aspect_w = 16,
                aspect_h = 16,
                length   = 2.0
            }
        }
    },

    special_tiles = {
        -- New-style water source mat
        {
            name      = "default_wate
            animation = {type = "vert
            backface_culling = false,
        }
    },

    --
    -- Behavior
    --
    walkable     = false, -- The play
    pointable    = false, -- The play
    diggable      = false, -- The play
    buildable_to = true,   -- Nodes ca

    alpha = 160,

    --
    -- Liquid Properties
    --
    drowning = 1,
    liquidtype = "source",

    liquid_alternative_flowing = "def
    -- ^ when the liquid is flowing

    liquid_alternative_source = "defa
    -- ^ when the liquid is a source

    liquid_viscosity = WATER_VISC,
    -- ^ how fast

    liquid_range = 8,
    -- ^ how far

    post_effect_color = {a=64, r=100,
    -- ^ color of screen when the pla
})
```

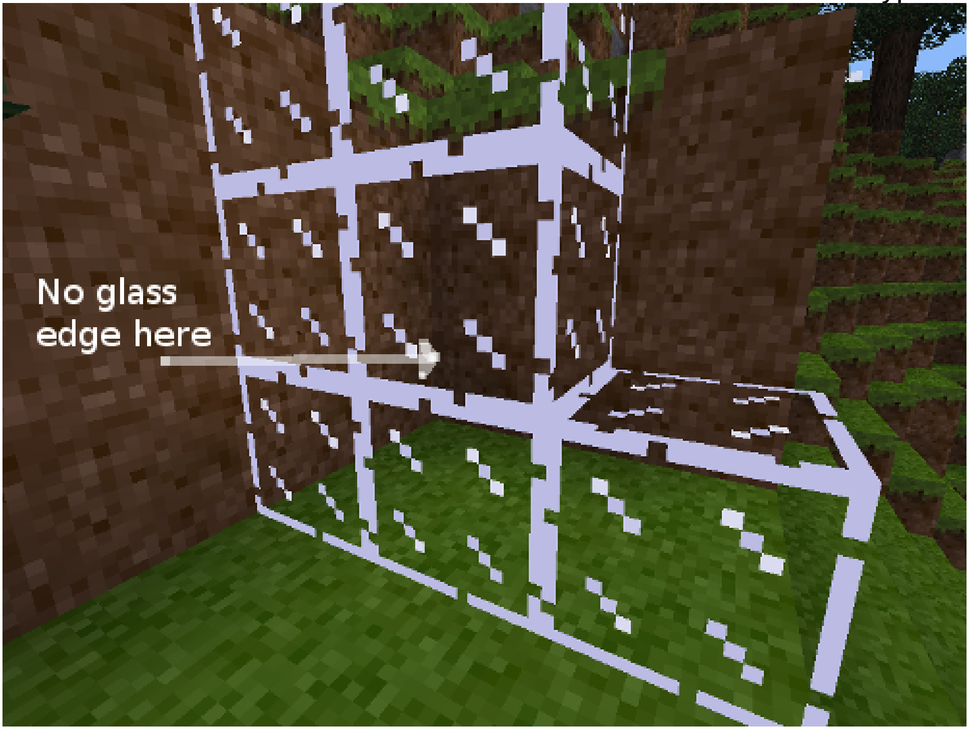
FlowingLiquid

See default:water_flowing in the default mod in minetest_game, it is mostly the same as the above example.

Glasslike

When you place multiple glasslike nodes together, you'll notice that the internal edges are hidden, like this:



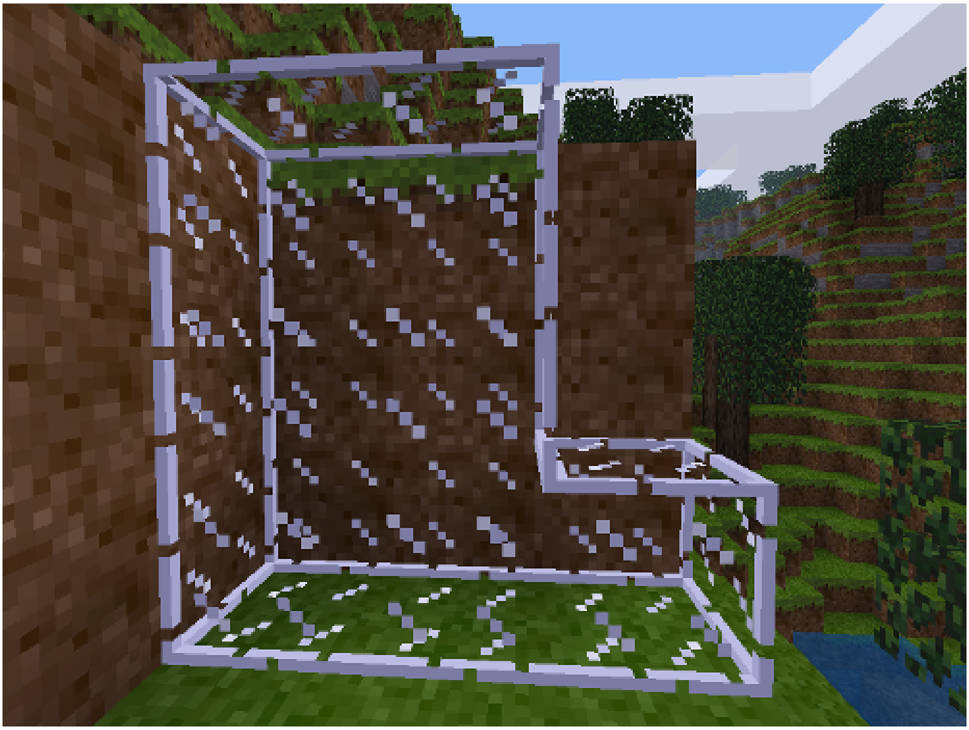


Glasslike's Edges

```
minetest.register_node("default:obsidian_glass", {
    description = "Obsidian Glass",
    drawtype = "glasslike",
    tiles = {"default_obsidian_glass.png"},
    paramtype = "light",
    is_ground_content = false,
    sunlight_propagates = true,
    sounds = default.node_sound_glass_defaults(),
    groups = {cracky=3,oddly_breakable_by_hand=3},
})
```

Glasslike_Framed

This makes the node’s edge go around the whole thing with a 3D effect, rather than individual nodes, like the following:


Glasslike_Framed's Edges

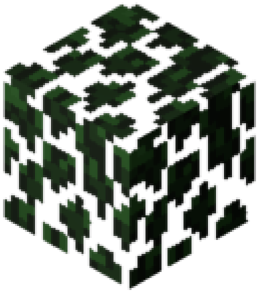
```
minetest.register_node("default:glass", {
    description = "Glass",
    drawtype = "glasslike_framed",

    tiles = {"default_glass.png", "default_glass_detail
    inventory_image = minetest.inventorycube("default_gl

    paramtype = "light",
    sunlight_propagates = true, -- Sunlight can shine th
    is_ground_content = false, -- Stops caves from being

    groups = {cracky = 3, oddly_breakable_by_hand = 3},
    sounds = default.node_sound_glass_defaults()
})
```

Glasslike_Framed_Optional

“optional” drawtypes need less rendering time if deactivated on the client’s side.

Allfaces

Allfaces nodes are partially transparent nodes - they have holes on the faces - which show every single face of the cube, even if sides are up against another node (which would normally be hidden). Leaves in vanilla minetest_game use this drawtype.


Allfaces drawtype

```
minetest.register_node("default:leave
    description = "Leaves",
    drawtype = "allfaces_optional",
    tiles = {"default_leaves.png"}
})
```

Allfaces_Optional

Allows clients to disable it using `new_style_leaves = 0`, requiring less rendering time.

TorchLike

TorchLike nodes are 2D nodes which allow you to have different textures depending on whether they are placed against a wall, on the floor, or on the ceiling.

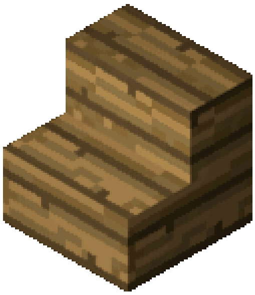
TorchLike nodes are not restricted to torches, you could use them for switches or other items which need to have different textures depending on where they are placed.

```
minetest.register_node("foobar:torch", {
    description = "Foobar Torch",
    drawtype = "torchlike",
    tiles = {
        {"foobar_torch_floor.png"},
        {"foobar_torch_ceiling.png"},
        {"foobar_torch_wall.png"}
    },
    inventory_image = "foobar_torch_floor.png",
    wield_image = "default_torch_floor.png",
    light_source = LIGHT_MAX-1,
    -- Determines how the torch is selected, ie: the wi
    -- each value is { x1, y1, z1, x2, y2, z2 }
    -- (x1, y1, y1) is the bottom front left corner
    -- (x2, y2, y2) is the opposite - top back right.
    -- Similar to the nodebox format.
    selection_box = {
        type = "wallmounted",
        wall_top = {-0.1, 0.5-0.6, -0.1, 0.1, 0.5, 0.1},
        wall_bottom = {-0.1, -0.5, -0.1, 0.1, -0.5+0.6,
        wall_side = {-0.5, -0.3, -0.1, -0.5+0.3, 0.3, 0
    }
})
```

Nodebox

Nodeboxes allow you to create a node which is not cubic, but is instead made out of as many cuboids as you like.

```
minetest.register_node("stairs:stair_
    drawtype = "nodebox",
    paramtype = "light",
    node_box = {
        type = "fixed",
        fixed = {
            {-0.5, -0.5, -0.5, 0.5, 0
            {-0.5, 0, 0, 0.5, 0.5, 0.
        },
    }
})
```


Nodebox drawtype

The most important part is the nodebox table:

```
{-0.5, -0.5, -0.5,       0.5,    0,   0.5},
{-0.5,    0,    0,       0.5,  0.5,   0.5}
```

Each row is a cubiod which are joined to make a single node. The first three numbers are the co-ordinates, from -0.5 to 0.5 inclusive, of the bottom front left most corner, the last three numbers are the opposite corner. They are in the form X, Y, Z, where Y is up.

You can use the NodeBoxEditor to create node boxes by dragging the edges, it is more visual than doing it by hand.

Wallmounted Nodebox

Sometimes you want different nodeboxes for when it is placed on the floor, wall, or ceiling like with torches.

```
minetest.register_node("default:sign_wall", {
    drawtype = "nodebox",
    node_box = {
        type = "wallmounted",

        -- Ceiling
        wall_top    = {
            {-0.4375, 0.4375, -0.3125, 0.4375, 0.5, 0.31
        },

        -- Floor
        wall_bottom = {
            {-0.4375, -0.5, -0.3125, 0.4375, -0.4375, 0
        },

        -- Wall
        wall_side   = {
            {-0.5, -0.3125, -0.4375, -0.4375, 0.3125, 0
        }
    },
})
```

Node Metadata

Introduction

In this chapter you will learn how to manipulate a node’s metadata.

- What is Node Metadata?
- Getting a Metadata Object
- Reading Metadata
- Setting Metadata
- Lua Tables
- Infotext
- Your Turn

What is Node Metadata?

Metadata is data about data. So Node Metadata is **data about a node**.

You may use metadata to store:

- an node’s inventory (such as in a chest).
- progress on crafting (such as in a furnace).
- who owns the node (such as in a locked chest).

The node’s type, light levels, and orientation are not stored in the metadata, but rather are part of the data itself.

Metadata is stored in a key value relationship.

Key	Value
foo	bar
owner	player1
counter	5

Getting a Metadata Object

Once you have a position of a node, you can do this:

```lua
local meta = minetest.get_meta(pos)
-- where pos = { x = 1, y = 5, z = 7 }
```

Reading Metadata

```lua
local value = meta:get_string("key")

if value then
    print(value)
else
    -- value == nil
    -- metadata of key "key" does not exist
    print(value)
end
```

Here are all the get functions you can use, as of writing:

- get_string
- get_int
- get_float
- get_inventory

In order to do booleans, you should use `get_string` and `minetest.is_yes`:

```lua
local value = minetest.is_yes(meta:get_string("key"))

if value then
    print("is yes")
else
    print("is no")
end
```

Setting Metadata

Setting meta data works pretty much exactly the same way.

```lua
local value = "one"
meta:set_string("key", value)
```

```
meta:set_string("foo", "bar")
```

Here are all the set functions you can use, as of writing:

- set_string
- set_int
- set_float

Lua Tables

You can convert to and from lua tables using `to_table` and `from_table`:

```
local tmp = meta:to_table()
tmp.foo = "bar"
meta:from_table(tmp)
```

Infotext

The Minetest Engine reads the field `infotext` in order to make text appear on mouse-over. This is used by furnaces to show progress and signs to show their text.

```
meta:set_string("infotext", "Here is some text that wil
```

Your Turn

- Make a block which disappears after it has been punched 5 times. (use on_punch in the node def and minetest.set_node)

Active Block Modifiers

Introduction

In this chapter we will learn how to create an **A**ctive **B**lock **M**odifier (**ABM**). An active block modifier allows you to run code on certain nodes at certain intervals. Please be warned, ABMs which are too frequent or act on too many nodes cause massive amounts of lag. Use them lightly.

- Special Growing Grass
- Your Turn

Special Growing Grass

We are now going to make a mod (yay!). It will add a type of grass called alien grass - it grows near water on grassy blocks.

```
minetest.register_node("aliens:grass", {
    description = "Alien Grass",
    light_source = 3, -- The node radiates light. Values
    tiles = {"aliens_grass.png"},
    groups = {choppy=1},
    on_use = minetest.item_eat(20)
})

minetest.register_abm({
    nodenames = {"default:dirt_with_grass"},
    neighbors = {"default:water_source", "default:water_
    interval = 10.0, -- Run every 10 seconds
    chance = 50, -- Select every 1 in 50 nodes
    action = function(pos, node, active_object_count, ac
        minetest.set_node({x = pos.x, y = pos.y + 1, z =
    end
})
```

Every ten seconds the ABM is run. Each node which has the correct nodename and the correct neighbors then has a 1 in 5 chance of being run. If a node is run on, an alien grass block is placed above it. Please be warned, that will delete any blocks above grass blocks - you should check there is space by doing minetest.get_node.

That’s really all there is to ABMs. Specifying a neighbor is optional, so is chance.

Your Turn

- **Midas touch**: Make water turn to gold blocks with a 1 in 100 chance, every 5 seconds.
- **Decay**: Make wood turn into dirt when water is a neighbor.
- **Burnin’**: Make every air node catch on fire. (Tip: “air” and “fire:basic_flame”). Warning: expect the game to crash.

Privileges

Introduction

Privileges allow server owners to grant or revoke the right to do certain actions.

- When should a priv be used?
- Checking for privileges
- Getting and Setting

When should a priv be used?

A privilege should give a player **the right to do something**. They are **not for indicating class or status**.

The main admin of a server (the name set by the `name` setting) has all privileges given to them.

Good:

- interact
- shout
- noclip
- fly
- kick
- ban
- vote
- worldedit
- area_admin - admin functions of one mod is ok

Bad:

- moderator
- admin
- elf
- dwarf

Declaring a privilege

```
minetest.register_privilege("vote", {
    description = "Can vote on issues",
    give_to_singleplayer = true
})
```

If `give_to_singleplayer` is true, then you can remove it as that's the default value when not specified:

```
minetest.register_privilege("vote", {
    description = "Can vote on issues"
})
```

Checking for privileges

There is a quicker way of checking that a player has all the required privileges:

```
local has, missing = minetest.check_player_privs(player_
    interact = true,
    vote = true })
```

`has` is true if the player has all the privileges needed.

If `has` is false, then `missing` will contain a dictionary of missing privileges[checking needed].

```
if minetest.check_player_privs(name, {interact=true, vot
    print("Player has all privs!")
else
    print("Player is missing some privs!")
end

local has, missing = minetest.check_player_privs(name,
    interact = true,
    vote = true })
if has then
    print("Player has all privs!")
else
```

```lua
    print("Player is missing privs: " .. dump(missing))
end
```

Getting and Setting

You can get a table containing a player’s privileges using `minetest.get_player_privs`:

```lua
local privs = minetest.get_player_privs(name)
print(dump(privs))
```

This works whether or not a player is logged in.
Running that example may give the following:

```lua
{
    fly = true,
    interact = true,
    shout = true
}
```

To set a player’s privs, you use `minetest.set_player_privs`:

```lua
minetest.set_player_privs(name, {
    interact = true,
    shout = true })
```

To grant a player some privs, you would use a mixture of those two:

```lua
local privs = minetest.get_player_privs(name)
privs.vote = true
minetest.set_player_privs(name, privs)
```

Adding privileges to basic_privs

Workaround / PR pending

This is a workaround for a missing feature. I have submitted a pull request / patch to make it so you don't need to edit builtin to add a priv to basic_privs.

To allow people with basic_privs to grant and revoke your priv, you'll need to edit builtin/game/chatcommands.lua:

In both grant and revoke, change the following if statement:

```
if priv ~= "interact" and priv ~= "shout" and
        not core.check_player_privs(name, {privs=true}) then
    return false, "Your privileges are insufficient."
end
```

For example, to add vote:

```
if priv ~= "interact" and priv ~= "shout" and priv ~= "
        not core.check_player_privs(name, {privs=true}) then
    return false, "Your privileges are insufficient."
end
```

Chat and Commands

Introduction

In this chapter we will learn how to interact with player chat, including sending messages, intercepting messages and registering chat commands.

- Send a message to all players.
- Send a message to a certain player.
- Chat commands.
- Complex subcommands.
- Intercepting messages.

Send a message to all players

It's as simple as calling the chat_send_all function, as so:

```
minetest.chat_send_all("This is a chat message to all pl
```

Here is an example of how it would appear ingame (there are other messages around it).

```
<player1> Look at this entrance
This is a chat message to all players
<player2> What about it?
```

Send a message to a certain player

It's as simple as calling the chat_send_player function, as so:

```
minetest.chat_send_player("player1", "This is a chat mes
```

Only player1 can see this message, and it's displayed the same as above.

Older mods

Occasionally you'll see mods with code like this:

```
minetest.chat_send_player("player1", "This is a server r
minetest.chat_send_player("player1", "This is a server r
```

The boolean at is no longer used, and has no affect [commit].

Chat commands

In order to register a chat command, such as /foo, use register_chatcommand:

```
minetest.register_chatcommand("foo", {
    privs = {
        interact = true
    },
    func = function(name, param)
        return true, "You said " .. param .. "!"
    end
})
```

Calling /foo bar will result in `You said bar!` in the chat console.

Let's do a break down:

```
privs = {
    interact = true
},
```

This makes it so that only players with the `interact` privilege can run the command. Other players will see an error message informing them which privilege they're missing.

```
return true, "You said " .. param .. "!"
```

This returns two values, firstly a boolean which says that the command succeeded and secondly the chat message to send to the player.

The reason that a player name rather than a player object is passed is because **the player might not actually be online, but may be running commands from IRC**. So don't assume that `minetest.get_player_by_name` will work in a chat command call back, or any other function that requires an ingame player. `minetest.show_formspec` will also not work for IRC players, so you should provide a text only version. For example, the email mod allows both `/inbox` to show the formspec, and `/inbox text` to send to chat.

Complex subcommands

It is often required to make complex chat commands, such as:

- /msg
- /team join
- /team leave
- /team list

Traditionally mods implemented this using Lua patterns. However, a much easier way is to use a mod library that I wrote to do this for you. See Complex Chat Commands.

Intercepting messages

You can use register_on_chat_message, like so:

```
minetest.register_on_chat_message(function(name, message
    print(name .. " said " .. message)
    return false
end)
```

By returning false, we're allowing the chat message to be sent by the default handler. You can actually miss out the line `return false`, and it would still work the same.

WARNING: CHAT COMMANDS ARE ALSO INTERCEPTED. If you only want to catch player messages, you need to do this:

```
minetest.register_on_chat_message(function(name, message
    if message:sub(1, 1) == "/" then
        print(name .. " ran chat command")
```

```
            return false
        end

        print(name .. " said " .. message)
        return false
    end)
```