Technical

Documentation of technical functionality in RadGame

- Multiplayer
  - Joining and Leaving a Server
  - Basic Concepts - Authority and RPC
- player Scene
  - Movement
- main scene
- Spells
  - Creating New Spells
  - Spell Dictionary
  - BaseSpell Class
  - Spell Container
  - Auras
  - Spell Queue
- Interactables
- Combat
  - Entering the Combat Script
  - Combat Events
  - Checks
  - Value Application
  - Stat Modificiation
  - Log Messages
- UI
  - Actionbars

Multiplayer

Hosting, Joining, Synchronizing

Joining and Leaving a Server

Currently, the game can only be joined on a dedicated server. This server needs to be built with a separate build template, which currently only runs on linux. A dedicated server uses the "dedicated_server" feature, which allows for queries to determine whether a function or section of a function should be run.

Spawning and Despawning Players

When the server is started, the presence of the dedicated_server feature is queried to connect the spawn_player and remove_player functions, such that only the server can spawn and despawn players.

The spawn_player function instatiates a new player scene, calls the pre_ready function, adds the node to the scene tree, and finally calls the post_ready function on the new player.

The pre_ready function sets the name of the node to the peer_id of the joined player, and calls the initialization of the BaseUnit class, from which player scenes inherit. This initialization initializes stats and the available spells in the spell container.

When the scene is added to the scene tree, _enter_tree is called. This function sets the multiplayer authority of the player_input node within the player scene. This has to be done at this stage, as otherwise the game crashes, or the authority is not properly set.

After the scene is fully loaded into the scene tree, the post_ready function is called. This function adds a camera for the player scene via rpc from the server to the peer controlling this player scene. The camera orientation is used in the player_input script to generate the direction vector that is then synced to the server and to peers. As the resulting direction is synced, the camera orientation does not have to be synced, and therefore the camera scene only has to exist locally. Finally, the processing of the player_input node is enabled by an rpc from the server to the controlling peer.

Note: Movement is processed by the server and all peers to reduce lag. However, the position and orientation of all players is synced from the server to enforce consistency, and any checks that involve position and orientation are done by the server.

When a player disconnects, the corresponding player scene in the scene tree is freed.

Basic Concepts - Authority and RPC

By default, everything in RadGame has the Server as authority. The only (so far) Exception is the player_input script that is part of each player character - the client that's controlling the character has the multiplayer authority over his input script.

Movement

The player movement is handled in player.gd, which should have the server as authority. Here's how:

```
func _physics_process(delta) -> void:
    # movement
    if not is_dead:
        handle_movement(delta)
```

```
func handle_movement(delta: float) -> void:
    # jumping
    if input.jumping and is_on_floor():
        velocity.y = jump_velocity
    if not is_on_floor():
        velocity.y -= gravity * delta
    input.jumping = false
    var direction = (transform.basis * Vector3(input.direction.x, 0,
input.direction.y)).normalized()
    if direction:
        velocity.x = direction.x * speed
        velocity.z = direction.z * speed
    else:
        velocity.x = move_toward(velocity.x, 0, speed)
        velocity.z = move_toward(velocity.z, 0, speed)
    move_and_slide()
```

In handle_movement, the direction gets read from the player_input.gd script, over which the player has authority. the player_input node in the player scene is actually a multiplayer synchronizer, and

syncs the "direction" variable to the server, so movement can then happen on the server. here's how the input script does it:

```
func _process(_delta):
	movement_direction()
```

```
func movement_direction():
	# unrotated direction from input
	var direction_ur = Input.get_vector("move_left","move_right","move_forward","move_back")
	# set strafing
	var strafing_left = false
	var strafing_right = false
	if direction_ur.x < 0:
		strafing_left = true
	if direction_ur.x > 0:
		strafing_right = true
	rpc_id(1,"set_strafing",strafing_left,strafing_right)
	# set backpedaling
	var backwards = false
	if direction_ur.y > 0:
		backwards = true
	rpc_id(1,"set_backpedaling",backwards)
	# rotate input according to camera orientation
	direction = Vector2(cos(-$"../camera_rotation".rotation.y)*direction_ur.x -\
		sin(-$"../camera_rotation".rotation.y)*direction_ur.y, \
		sin(-$"../camera_rotation".rotation.y)*direction_ur.x +\
		cos(-$"../camera_rotation".rotation.y)*direction_ur.y)
	if Input.is_action_just_pressed("jump"):
		jump.rpc()
```

So the direction comes straight from the input and gets synced via the synchronizer, while states such as strafing and backpedaling use RPCs to sync animations.

Spells

while there is targeting and interaction functionality in player_input.gd, we don't care about these right now. let's find where spells happen. In actionbar_slot.gd there is this:

```
func _on_pressed() -> void:
	# the rpc call to fire the spell is sent from player_input, as the server does not load
```

```
	# the actionbar ui elements of players, and using player_input is somewhat intuitive
	References.player_reference.get_node("player_input").\
		request_enter_spell_container(slot_spell_id)
```

and back in the player_input.gd :

```
func request_enter_spell_container(spell_id: String):
	rpc_id(1,"enter_spell_container",spell_id)



@rpc("authority","call_local")
func enter_spell_container(spell_id: String):
	$"../spell_container".spell_entrypoint(spell_id)
```

so the first call is a local function that allows the input script, with player authority, to send a request to the server to enter a spellcontainer. the second one calls this in spell_container.gd:

```
func spell_entrypoint(spell_id: String) -> int:
	# determine whether spell is present in container
	var node_name = "spell_"+spell_id
	var spell_node = get_node_or_null(node_name)
	if spell_node == null:
		print("spell %s not known"%spell_id)
		return 1  # spell not known
	result = spell_node.trigger()
	return result
```

the second to last line, `result = spell_node.trigger()` actually triggers the spell. Spells when triggered should send combat events via the combat script.

player Scene

Functionality of the player scene

player Scene

Movement

Player movement is controlled by the handle_movement function in the player script, which is called in _physics_process. It uses the direction variable that is synced from the player_input node.

player_input generated the direction vector based on the direction the player intends to move relative to the camera, i.e. the movement keys are used to determine the 2D horizontal movement vector, which is then rotated according to the orientation of the camera.

Jumping is detected in player_input, and when a jump occurs, the jumping variable is set to true vie rpc, to ensure that the call is properly detected and handled.

As the 2D horizontal movement direction and jumping are synced variables, the server and all peers can use these two variables to process the movement of a player, while only requiring the camera scene to exist locally. While all peers process the movement of all players, the server syncs the position and orientation of all players with all peers, and thus overrides potential local deviations on peers. Peers process movement only to reduce lag.

Movement speed is a separate variable, with which the resulting velocity gained from the direction vector is multiplied. This movement speed variable is considered a stat that can be changed by talents, buff, debuffs, etc., but in the absence of such modifications movement speed is equal for all players.

main scene

the main scene

Spells

How spells are structured and triggered

Creating New Spells

New spells are intended to be simple to add, if they do not do esoteric things. In particular spells that deal damage, heal, apply a buff, debuff or absorb shield, or combine these elements, should be fairly simple to implement. To add a new spell, a few steps are required.

Creating the Database Entry

The spell database data/db_spells.json needs to be appended with an entry for the new spell. This is done by creating a new entry with a new unique ID, and specifying the key and value pairs. Which keys are required depends on the type of spell that is added. More information can be found here.

The spell also needs to be added to the spell_list of the unit or interactable that should use the spell, in either the data/db_stats_player.json file, the data/db_stats_npc.json file, or the data/db_stats_interactable.json file.

Creating the Spell Script

Every spell scene needs an attached spell script that handles the triggering of the spell. The script should inherit from the BaseSpell class, which contains some core functionality. The _ready function of the spell script should set the ID as a string, and call initialize_base_spell with the spell ID as the argument. This creates the dictionaries that contain the spell data, and the cooldown timer. The script should further contain a trigger function, which determines the source and target of the spell, and goes through a number of checks that determine whether the spell can actually be used when it is triggered. The standard checks, such as whether the spell is already on cooldown, whether the target is within range, etc., are contained in the BaseSpell class.

After the checks, the start_cast function of the BaseSpell class can be called, which sets the is_casting state to true and triggers the gcd if applicable. This requires the cast_success function to be passed as an argument, as it links the timeout of the cast timer to this function. The cast_success function then needs to be added to the spell script to determine what happens when the spell is successfully cast. This usually consists of applying the resource cost, sending the combat event, and calling finish_cast. finish_cast is a function in the BaseSpell class, which plays an animation, disconnects the cast timer timeout signal from the cast_success function, starts the spell cooldown if it exists, and sets the is_casting state to false.

Instant cast spells can forego the call to start_cast, as the cast finishes immediately. Instead, the trigger function can send a combat event, trigger the gcd, and call finish_cast. finish_cast requires the cast_success callable as an argument, for which the BaseSpell class has a dummy function that

does nothing, and is intended for use in this case.

Note: Spells that increase the maximum health of a target by applying a buff should in principle also send a second combat event to heal the target for that amount. This is done to avoid reducing the health percentage every time such a buff is applied, and to make sure that a tank increasing his maximum health is also healed for that amount, as doing otherwise might feel clunky. If the heal component is not added, it should be carefully reasoned why not. The heal spell requires a separate spell dictionary to be created, which requires the spelltype key to be set to "heal", and requires the name and can_crit keys. If no value is prescribed (which can be constructed from the buff data), more keys are required to function as a healing spell. Spell ID 8 (Player Test Buff) is an example of a buff that increases maximum HP.

Spell Dictionary

Spells are the combat skills that players, NPCs and interactables use to affect players and NPCs. They can be divided into damage, heal, dot, hot, buff, debuff, and aura spells. All spell data are contained in the data/db_spells.json file.

The spell data are stored in two separate dictionaries. The first is spell_base, which contains the base values as found in the data/db_spells.json file. This dictionary does not change. The second in spell_current, which is based on spell_base, but can be modified by talents or other effects. For example, if a spell gains a 10% increase to critical hit chance from a talent, this is applied to spell_current. Only effects that are applied to individual spells are handled in this way. Changes to player stats are handled in the stats_current dictionary in the player script.

Basic data/db_spells.json keys
All spells have some common keys that are required in the data dictionary, or are common optional keys for all types of spell. These are:

Required
name: The name of the spell, as it appears in the combat log.
targetgroup: The group (player, npc, hostile, etc.) that can be targeted by this spell.

Optional
icon: The icon used for a spell
role: The player role (i.e., tank, heal, melee, ranged) that uses this variant of the spell. Needed for player spells that are available to more than one role.
resource_cost: The cost of the main resource to trigger the spell. Can be negative to gain resource.
max_range: The maximum range of the spell.
cooldown: The cooldown period of the spell in seconds.
on_gcd: Whether the spell is affected by and triggers the global cooldown.

Damage Spell Keys
Damage spells require some further keys to function. These are:
spelltype: Must be set to "damage".
effecttype: Must be set to "physical" or "magic", and will use the corresponding modifiers.
value_base: The stat (e.g., max health, current health, primary, etc.) that is used as a base to calculate the effective damage.
value_modifier: The multiplicative modifier that is applied to the base value to calculate the

effective damage.
avoidable: Whether or not the spell can be avoided. Set to 1 for avoidable, or 0 for not avoidable.
can_crit: Whether or not the spell can critically hit. Set to 1 for critical hits, and 0 for no critical hits.
crit_chance_modifier: Additive constant to the critical hit chance of the spell. Applied to the base critical hit chance of the source.
crit_magnitude_modifier: Multiplicative modifier to the effective damage when a hit is critical.

Healing Spell Keys
Healing spells require some further keys to function. These are:
spelltype: Must be set to "heal".
effecttype: Must be set to "physical" or "magic", and will use the corresponding modifiers.
value_base: The (e.g., max health, current health, primary, etc.) that is used as a base to calculate the effective healing.
value_modifier: The multiplicative modifier that is applied to the base value to calculate the effective healing.
can_crit: Whether or not the spell can critically hit. Set to 1 for critical hits, and 0 for no critical hits.
crit_chance_modifier: Additive constant to the critical hit chance of the spell. Applied to the base critical hit chance of the source.
crit_magnitude_modifier: Multiplicative modifier to the effective healing when a hit is critical.

DoT Spell Keys
DoT effects use the same keys as damage spells for the damage events that they trigger. However, they require a few additional keys:
ticks: The number of ticks that a DoT effect has.
tickrate: The time between two ticks, in seconds.
unique: Whether a DoT effect can only be present on a unit once in total (value 1), or once per source (value 0).

HoT Spell Keys
HoT effects use the same keys as healing spells for the healing events that they trigger. However, they require a few additional keys:
ticks: The number of ticks that a HoT effect has.
tickrate: The time between two ticks, in seconds.
unique: Whether a HoT effect can only be present on a unit once in total (value 1), or once per source (value 0).

Buff Spell Keys
Buffs do not cause damage or healing events, and therefore require a few different keys to function:
auratype: Must be set to "buff".
modifies: An array of stats that is to be modified. Can contain the same stat multiple times.
modify_type: An array of the same length as "modifies", containing either "mult" or "add" for each element, which determines whether a stat is affected additively, or multiplicatively.
modify_value: An array of the same length as "modifies", containing the additive or multiplicative

modifier applied to each stat.
duration: The total duration of the buff effect.
unique: Whether a buff effect can only be present on a unit once in total (value 1), or one per source (value 0).

Debuff Spell Keys
Debuffs do not cause damage or healing events, and therefore require a few different keys to function:
auratype: Must be set to "debuff".
modifies: An array of stats that is to be modified. Can contain the same stat multiple times.
modify_type: An array of the same length as "modifies", containing either "mult" or "add" for each element, which determines whether a stat is affected additively, or multiplicatively.
modify_value: An array of the same length as "modifies", containing the additive or multiplicative modifier applied to each stat.
duration: The total duration of the debuff effect.
unique: Whether a debuff effect can only be present on a unit once in total (value 1), or one per source (value 0).

Absorb Spell Keys
Absorb effects are essentially a combination of healing and buff effects. The keys they require are:
auratype: Must be set to "absorb".
value_base: The value that is used as a base to calculate the total absorb amount.
value_modifier: The multiplicative modifier that is applied to the base value to calculate the total absorb amount.
can_crit: Whether or not the spell can critically hit. Set to 1 for critical hits, and 0 for no critical hits.
crit_chance_modifier: Additive constant to the critical hit chance of the spell. Applied to the base critical hit chance of the source.
crit_magnitude_modifier: Multiplicative modifier to the total absorb amount when a hit is critical.
duration: The total duration of the absorb effect.
unique: Whether an absorb effect can only be present on a unit once in total (value 1), or one per source (value 0).

BaseSpell Class

Spells inherit from the BaseSpell class, which contains some functionality that is shared between all spells. This class contains the dictionaries that store spell data, the cooldown timer, and the flag that determines whether a spell uses mouseover targeting.

initialize_base_spell takes the ID of a spell as an argument, and reads the corresponding data from the data/db_spells.json file. It then creates the spell_base and spell_current dictionaries, and connects the signal for the global cooldown, if the spell is on the global cooldown, and adds the cooldown timer node.

Note: spell_base contains the unmodified data read from the data/db_spells.json file. It does not change. spell_current contains the spell data, and is modified with changes by things such as talents.

get_spell_target determines which target the spell is used on. The target can either be the mouseover target, or the selected target of the source. If both are null, an individual spell may set the target to the source in its own script. The current intent is to enable or disable mouseover targeting for each spell individually in a future update.

is_illegal_target checks whether the target of the spell is a legal target, by checking whether it is in the correct group.

is_on_cd checks whether the spell is currently on cooldown, by checking the cooldown timer.

insufficient_resource checks whether the source of the spell has sufficient resources to trigger the spell.

is_not_in_range checks whether the target of the spell is within the maximum range of the spell.

is_not_in_line_of_sight checks whether the target is within line of sight of the source. This is currently not functional.

trigger_cd starts the cooldown timer of the spell, if it is not currently on cooldown with a higher remaining duration than the new cooldown would be. For example, a spell that has 25 seconds of cooldown left would not have a global cooldown applied to it, but a spell with no running cooldown timer would have the global cooldown applied to it.

trigger_gcd causes the spell container to emit the gcd signal, which starts the gcd of all spells that use it, with the duration being specificed in the spell_container.

update_resource applies the resource cost (or gain) caused by the spell to the source of the spell.

start_cast sets the is_casting state to true, links the cast timer to the cast_success function of the spell being cast, starts the cast timer, and causes the gcd signal to be emitted.

cast_success is a dummy function for use with instant spells, and is overridden by spells that have a cast time. This function is then called when the cast timer times out. For spells with cast time, the overriding function in the individual spell script applies resource costs, sends the combat event, and calls finish_cast.

finish_cast plays the spell success animation, disconnects the cast timer from the cast_success function, triggers the spell cd, sets the is_casting state to false, and checks for queued spells that are automatically cast next.

Spell Container

The spell_container scene consists of a node of type Node, and is used as a parent node for all individual spell scenes that a unit or interactable has available. The spell_container script contains a few functions.

spell_entrypoint is the function that receives the request that a spell is to be triggered. It then checks whether the spell is known to the unit or interactable, and triggers the spell if it is known.

send_gcd emits a signal that all child nodes listen to, if they are affected by the gcd. This signal causes these child nodes to start the gcd timer.

set_queue disconnects the cd_timer timeout signal to the cast_queued function if it is connected. It then sets the current queued spell ID, and connects the timeout signal of that spells cd_timer to the cast_queued function. This causes the spell queue to be handled when the cd of the queued spell expires, i.e. when it can be cast.

cast_queued attempts to cast queued spells when the current cast finishes, or when the cooldown of the queued spell expires.

This script may be expanded in the future to handle changes made to spells, and handle role swaps.

Auras

Auras are temporary or permanent effects on a unit. They can be DoTs, HoTs, buffs, debuffs or absorbs. Every unit has an aura_container scene, which has a Node type root node, and contains Node type nodes named dot_container, hot_container, buff_container, debuff_container, and absorb_container. The dot and hot containers are populated by aura_dot scenes, with hots being negative dots, the buff and debuff containers are populated by aura_buff scenes, with debuffs being negative buffs, and the absorb container is populated by aura_absorb scenes.

aura_dot
The aura_dot scene stores the spell data, source, and target of a dot or hot. In the initialize function, these are stored within the scope of the full script, and the tick timer is added as a child node. In the _ready function, the timer is started. The tick function is connected to the timeout signal of the tick timer, and triggers the combat event by calling Combat.combat_event_entrypoint, and counts the number of ticks that have occurred. If the maximum number of ticks is reached, the aura is removed by calling the remove_aura function, which stops the tick timer and calls Combat.combat_event_aura_entrypoint with remove=true to remove the scene. If the aura is reapplied while it is still active, the reinitialize function is called, which updates the spell data, stops the tick timer, resets the number of occurred ticks to 0, updates the total number of ticks and the tickrate, and restarts the tick timer.

aura_buff
The aura_buff scene stores the spell data, source, and target of a buff or debuff. In the initialize function, these are stored within the scope of the full script, and the expiration timer is added as a child node. In the _ready function, Combat.buff_application is called to apply the buff or debuff to the current stats of the target, and the expiration timer is started. When the expiration timer expires, remove_aura is called, which stops the expiration timer, removes the buff from the target's current stats, and finally remove the buff scene. If a buff is reapplied while it is already active, the reinitialize function is called, which updates the spell data, stops the expiration timer, reapplies the buff, which overwrites old buff values in case they have changed, updates the duration of the expiration timer, and restarts the expiration timer.

aura_absorb
The aura_absorb scene stores the spell data, source, target, and remaining absorb value of an absorb. In the initialize function, the spell data, source and target are stored within the scope of the full script, the total absorb value is calculated by a call to Combat.value_query, and the expiration timer is added as a child node. In the _ready function, the expiration timer is started, and the array of active absorbs on the affected unit are sorted by increasing duration with a call to sort_absorbs. When the expiration timer expires, remove_absorb is called, which sets the remaining absorb value to 0, to avoid any absorbs while the aura is being removed, removes the absorb from the affected unit's array of active absorbs, and finally remove the absorb scene. If an absorb is reapplied while it is already active, the reinitialize function is called, which updates the spell data, stops the

expiration timer, calls Combat.value_query to calculate the new total absorb amount, updates the duration of the expiration timer, restarts the expiration timer, and sorts the active absorbs on the affected target.

Spell Queue

In order to make spell casting smoother, and not have short gaps between spells when the next is not triggered frame-perfectly, a spell queuing system exists. This system allows spells to be queued towards the end of a currently active cast, or a currently active cooldown, including the gcd.

queue_instant is the queue for instant cast spells that do not trigger a gcd. Multiple of these can be queued, as they do not interfere with other spell casts. When such spells are queued, they are appended to an array, and cast before the spell designated by the queue variable. The order in which instant, non-gcd spells are queued is the order in which they are also cast.

queue holds the ID of a single spell that either has a cast time, or triggers a gcd. Only one of these spells can be queued, because the cast time or gcd would interfere with multiple casts. This spell is triggered after the queue_instant array is emptied of queued spells. The next spell can then be queued towards the end of this spell's cast, or towards the end of the triggered gcd.

The queued spells are cast when the cast_queued function is called. This happens either when the cd of the queued spell expires, or when the preceding spell calls the finish_cast function. If the queue is triggered by an expiring cd while a cast is already ongoing, the queued spells are not triggered because the player is already casting, but the queue is not emptied. The queue will only be handled in the call to finish_cast. If the queue is triggered by finish_cast, the cd_timer timeout signal is disconnected to not erroneously trigger the queue twice, and the queue is handled.

Interactables

Interactables are objects in the game world that can be interacted with, and trigger a specific effect when this is done. Effects can be spells cast on the interactable target, or other specified targets, or changes made to the game world. Technically, virtually anything can be done.

Constructing an Interactable
Interactables require a scene with a CharacterBody3D root node, and an attached script that inherits from BaseInteractable. Mandatory child nodes are a Label3D node named "interact_prompt", an Area3D node named "range" that scans collision layer 2, and which has a CollisionShape3D child node named "range_shape" that uses a SphereShape3D as shape, and finally a MultiplayerSynchronizer node named "mpsynchronizer".

The attached script needs to set the multiplayer authority to 1 (the server) in the _enter_tree function, and needs to call create_prompt_text and initialize_base_interactable from the BaseInteractable class. create_prompt_text needs to be called locally for all players, i.e. for all peers that are NOT multiplayer authority. The initialization of the base interactable needs to be called on the server, i.e. for the multiplayer authority.

The script further requires a trigger function, which uses the root node of the interacting player scene as an argument. The BaseInteractable class contains a trigger function, which will warn the server via a printed message if it is not overriden in the script of a specific interactable.

If a spell is to be triggered by interacting, the spell must be triggered directly, and not via the spell_container entrypoint. This is to allow for the easy reuse of the spell_container scene, but to still be able to pass the interacting player as an argument as the target of the triggered spell.

BaseInteractable
The BaseInteractable class is the class from which interactables inherit. It contains a stats_current variable, which is required to calculate spell effects, contains the maximum range at which interactions can occur, and contains the array of spells the interactable has available.

The BaseInteractable class further contains a number of functions:

initialize_base_interactable takes the unit ID of the interactable to read the stats from the data/db_stats_interactable.json file, sets up the spell container with all required spells, calls the connect_signals function described below, and sets the maximum interaction range.

connect_signals connects the body_entered and body_exited signals of the "range" child node to the add_interactable and remove_interactable functions described below. The shape and size of the

body of the "range" node is determined by its "range_shape" child node, and should be set to a SphereShape3D.

add_interactable adds an interactable to the array of interactables of a player when they enter the body of the interactable.

remove_interactable removes an interactable from the array of interactables of a player when they exit the body of the interactable.

Note: A player can only interact with the nearest interactable within their interactables array. The nearest interactable is searched for every frame by the get_nearest_interactable function in the player script. This function also uses rpc to show or hide interact prompts as necessary for the corresponding player. The interact prompt is only shown above the current nearest interactable, to clearly indicate which interactable is triggered if the interaction hotkey is used.

create_prompt_text creates the text that is shown in the interact_prompt label. The prompt is "Interact [%s]", with %s being the hotkey for interactions, where the " (Physical)" suffix is trimmed, if it is part of the hotkey.

trigger is a function that provides a warning in the server standard output if it is not overridden by a trigger function in the interactable script that inherits from BaseInteractable.

Adding a New Interactable
When adding a new interactable, it should be placed in the "interactable" subdirectory of the "scenes" directory, and the script for it should be placed in the "interactable" subdirectory of the "scripts" directory. A basic template for interactables can be found in the testing subdirectory of these directories, as interact_absorb, interact_damage, and interact_heal.

Every interactable also needs an entry in the data/db_stats_interactables.json file. A new ID must be assigned, and a dictionary specified that contains all required information. This ID must then be passed to the initialize_base_interactable function as an argument during the _ready function of the new interactable. The required keys in the dictionary are
unit_name: Specifies the name of the interactable that appears in the combat log.
spell_list: The list of spells that the interactable can use. Can be left empty, if no spell is required, but the key itself is mandatory.
interact_range: The maximum range at which the interactable is present in a players interactables array.

If the interaction triggers a spell, the dictionary must further contain all stats required for the spell to function, such as a primary value, damage modifiers, heal modifiers, etc.

Note: If a triggered spell should target the interacting player, the interactor can be passed to the spell's trigger function to achieve this. This requires that the spell in question takes a target as an argument in its own trigger function.

Combat

The combat script

Entering the Combat Script

There are a number of functions that are intended as entrypoints into the combat script, which are called by spell scripts. The appropriate function must be called to properly proceed through a combat event.

combat_event_entrypoint
This function serves as the entrypoint for simple combat events, where damage is dealt or healing is done. The function uses the spell, the source, the target and an optional value as arguments, and checks whether the spell is a damage or healing spell to call the next function, which is either combat_event_damage or combat_event_heal.

A value can be prescribed, which will cause the combat event to use that specific value, instead of calculating the value from stats. If no value is prescribed, the default value of -1 is used, which causes the following functions to calculate the actual value.

combat_event_aura_entrypoint
This function serves as the entrypoint for the application and removal of auras, i.e. dots, hots, buffs, debuffs, and absorbs. The function uses the spell, the source, the target, and a boolean as arguments. The boolean is optional, with a default value of false. If the boolean is set to true, the aura is removed instead of applied.

buff_application
This function serves as the entrypoint into the application of a buff or debuff, and leads to the recalculation of stats based on buffs and debuffs. The function uses the spell, the source, the target, and a boolean as arguments. The boolean is optional, with a default value of false. If the boolean is set to true, the buff is removed instead of applied, and the stat calculation is done appropriately.

value_query
This function calculates the damage or healing value of a spell without the following application of this value. It is useful to calculate the magnitude of absorb shields, and possibly for snapshotting values of dots and hots. It uses the value_modifier of the spell, the value_base stat and the active modifier (i.e., damage_modifier or heal_modifier) of the source, and the passive modifier (i.e., defense_modifier or heal_taken_modifier) of the trget to calculate the resulting value of a spell. This is the same function that combat events that deal damage or healing eventually use to calculate the value of a spell.

Combat Events

Combat events are damage, healing and aura events that are reached via the entrypoints into the combat script. They calculate the magnitude of a spell, unless prescribed, check for avoidance and ciritcal hits, apply the hp change or aura to the target, and log the interaction.

combat_event_damage
This function handles damage events. It uses the spell, the source, the target, and the value as arguments. If the value was not prescribed when calling the entrypoint, it is -1 at this point, which causes this function to calculate the actual value as a first step. It then checks whether the spell is avoidable, and if so, if it is avoided. If it is not avoided, it checks whether the spell can hit critically, and changes the value accordingly if the hit is critical. This leads to the finalized value, which is then first applied to the absorb shields on the target, if they are present, and then to the health of the target. The interaction is then logged.

combat_event_heal
This function handles healing events. It uses the spell, the source, the target, and the value as arguments. If the value was not prescribed when calling the entrypoint, it is -1 at this point, which causes this function to calculate the actual value as a first step. It then checks whether the spell can hit critically, and changes teh value accordingly if the hit is critical. This leads to the finalized value, which is appriel to the health of the target. The interaction is then logged.

combat_event_aura
This function handles the instantiating and adding of aura scenes. It uses the spell, the source, and the target as arguments. First, the listed name of the aura is created, which is the name that the child scene will have, and the name that will appear in the aura array of the target. It consists of the name of the spell, and if the aura is not unique (i.e., multiple sources can apply the same aura on one target), the name of the source is appended. This array is sued t track which auras are present on a target. If the aura is already present, it is reinitialized by calling the reinitialize function in the already present aura script. If the aura is not yet present, the appropriate aura scene (dot, buff or absorb) is instantiated, the name is changed to the previously generated listed name, the aura is initialized and added to the appropriate aura container, and the aura name is appended to the aura array of the target. The interaction is then logged.

combat_event_aura_remove
This function handles the removal of auras from a target. It uses the spell, the source, and the target as arguments. First, the listed name of the aura is created, which is the name that the child scene will have, and the name that will appear in the aura array of the target. It consists of the name of the spell, and if the aura is not unique (i.e., multiple sources can apply the same aura on one target), the name of the source is appended. The aura name is then erased from the target's aura array, and the scene is removed from the aura container. The interaction is then logged.

Checks

There are a few checks that the combat script requires to function.

is_critical
This function checks whether a hit is critical. It uses the crit modifier of a spell and the base critical hit chance of the source as arguments. A random number generator is initialized, and a float between 0 and 1 is generated. If the generated float is less than or equal to the sum of the base critical hit chance and the crit modifier, the hit is critical. The result for a critical hit is 1, and the result for a non-ciritcal hit is 0, as the return value is used for a calculation.

is_avoid
This function checks whether a hit is avoided. It uses the avoidance chance of the target as the argument. A random number generator is initialized, and a float between 0 and 1 is generated. If the generated float is less than or equal to the avoidance chance of the target, the hit is avoided.

Value Application

The final part of the combat event sequence, before the logging, is the application of the final spell value to the absorb shields and health of the target.

apply_damage
This function applies damage to the health of the target, and returns the difference between the final spell value and the target's current health before the damage is dealt as overkill. It uses the value to be applied and the target as arguments.

apply_absorb
This function applies damage to the currently active absorb shield on the target, in order of the duration-sorted array of absorbs. It takes the value to be applied, the target, the spell name, the source name and the target name as arguments. The function attempts to deplete all absorb shields in order, until either the remaining damage value reaches zero, or all absorb shields are depleted. For every absorb shield that has damage applied to it, a combat event is logged to display which shield from which source is absorbing how much damage. If a shield is depleted, it is scheduled for removal after the iteration during which the damage is applied. The depleted absorb shields are then removed directly afterwards, as doing so during the iteration leads to unpredictable results. Finally, the remaining damage value is returned. If the remaining value is 0, the combat_event_damage function is exited. Otherwise, it continues with the application of the remaining damage to the target's health.

apply_heal
This function applies healing to the health of the target, and returns the overheal value. It uses the value to be applied and the target as arguments. A min() function is used, as healing cannot heal a target beyond maximum health.

Combat

Stat Modificiation

Stat modification occurs when a buff or debuff is applied to a target, and an array of stats is changes either additively (positive or negative) or multiplicatively (with values greater than or less than 1).

apply_buff
This function applies the stat modifications of a buff to the stats_add and stats_mult dictionaries of the target, and calls the calc_current_from_base_partial function, which recalculates the modified stats with the newly updated modifiers. It uses the spell, the source name, and the target as arguments.

remove_buff
This function removes the stat modifications of a buff or debuff from the stats_add and stats_mult dictionaries of the target, and calls the calc_current_from_base_partial function, which recalculates the modified stats with the newly updated modifiers. It uses the spell, the source name, and the target as arguments.

calc_current_from_base_partial
This function calculates the current stat values from the base value and modifiers. The word partial refers to the fact that not all stats are recalculated. It is intended for use when the character is affected by a buff or debuff, or a talent that increases a singular or only few stats. The function uses the target and an array of modified stats as arguments. The array of modified stats are looped over, and the additive and multiplicative modifiers are first then determined, and then applied to the base stats. For changes in maximum health and resource, the difference to the previous value is stored. If this difference is negative, i.e., the maximum values decrease, the current value of health or resource are set to the minimum of either itself or the new maximum, to avoid overcapping.

calc_current_from_base_full
This function calculates the current stat values from the base value and modifiers. The word full refers to the fact that all stats are recalculated. It is intended for use when the character class or role is changed, as this affects most or all stats. The function uses the target and an array of modified stats as arguments. The array of modified stats are looped over, and the additive and multiplicative modifiers are first then determined, and then applied to the base stats. For changes in maximum health and resource, the difference to the previous value is stored. If this difference is negative, i.e., the maximum values decrease, the current value of health or resource are set to the minimum of either itself or the new maximum, to avoid overcapping.

Log Messages

All combat events are logged, to ensure that combat encounters can be analyzed. No in-game analysis exists currently, but is planned to be implemented at some point. It is not actively being developed at the moment, however.

In the future, it is intended that combat log messages can be saved as a text file, such that the file can be subjected to post-processing. Some post-processing scripts will probably be developed by cheese at some point, but are not currently being actively developed.

UI

Description of UI elements, and their interaction with other parts of the software

UI

Actionbars

Actionbars are grid containers that contain buttons. Spells can be assigned to the buttons, which allows clicking or using the appropriate hotkey to trigger the spell.

Layout
By default, there are two actionbars with 12 slots each. The number of columns is set to 12, making both actionbars horizontal. A menu option to resize, reposition, and to change the number of columns is planned, but not in active development at the moment.

Cooldown Swipes
When a spell is on cooldown, this needs to be reflected in the actionbar, such that the player knows the cooldown status. Therefore, a progressbar is used to display the cooldown as a swipe animation. A number display to state the cooldown in seconds is planned, but not in active development at the moment.

The cooldown swipe does not use the cooldown information of the spell on the server. Instead, the player has a cd_timer_container node, in which a cooldown timer is instantiated for every spell that the player knows. The trigger_cd function in the BaseSpell class, which triggers the spell cooldown on the server, also triggers the local cd timers in the player's cd_timer_container. The actionbar buttons use this local timer to display the current remaining cooldown on every frame.

As the wait_time of a cooldown timer can change in some circumstances, e.g. when using a spell reduces the cooldown of another spell, this is not used to normalize the cooldown swipe value. Doing so would lead the swipe to be reset every time a change to the cooldown is made. Instead, a cd_full_duration variable is set, which is equal to the duration given by the cooldown key in the current_spell dictionary of the spell. This way, the cooldown swipe is always relative to the full cooldown that a player expects from the spell.