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.