Worldgen

General Notes

Biome Filters:

Biome filters work similarly to *recipe filters* and can be used to create complex and exact filters to fine-tune where your features may and may not spawn in the world. They are used for the `biomes` field of a feature and may look something like this:

```
WorldgenEvents.add(event => {
  event.addOre(ore => {
    // let's look at all of the 'simple' filters first
    ore.biomes = 'minecraft:plains' □□// only spawn in exactly this biome
    ore.biomes = /^minecraft:.*/□□□// spawn in all biomes that match the given pattern
    ore.biomes = '#minecraft:is_forest' □// spawn in all biomes tagged as 'minecraft:is_forest'

    // filters can be arbitrarily combined using AND, OR and NOT logic
    ore.biomes = {}□□□□□□□□// empty AND filter, always true
    ore.biomes = []□□□□□□□□// empty OR filter, always true
    ore.biomes = { not: 'minecraft:ocean' }□// spawn in all biomes that are NOT
'minecraft:ocean'

    // since AND filters are expressed as maps and expect string keys,
    // all of the 'primitive' filters can also be expressed as such
    ore.biomes = {□□□□□// see above for an explanation of these filters
      id: 'minecraft:plains',
      id: /^minecraft:.*/,□□□// regex (also technically an ID filter)
      tag: 'minecraft:is_forest',
    }
    // note all of the above syntax may be mixed and matched individually
  })
})
```

Rule Tests and Targets:

In 1.18, Minecraft WorldGen has changed to a "target-based" replacement system, meaning you can specify specific blocks to be replaced with specific other blocks within the same feature configuration. (For example, this is used to replace Stone with the normal ore and Deepslate with the Deepslate ore variant).

Each target gets a "rule test" as input (something that checks if a given block state should be replaced or not) and produces a specific output block state. While scripting, both of these concepts are expressed as the same class: `BlockStatePredicate`.

Syntax-wise, `BlockStatePredicate` is pretty similar to biome filters as they too can be combined using `AND` or `OR` filters (which is why we will not be repeating that step here), and can be used to match one of three different things fundamentally:

1. **Blocks:** these are simply parsed as strings, so for example `'minecraft:stone'` to match Stone
2. **Block States:** these are parsed as the block ID followed by an array of properties. You would use something like `'minecraft:furnace[lit=true]'` to match only Furnace blocks that are lit. You can use `F3` to figure out a block's properties, as well as possible values through using the debug stick.
3. **Block Tags**: these are parsed in the "familiar" tag syntax, so you could use `'#minecraft:base_stone_overworld'` to match all types of stone that can be found generating in the ground in the Overworld.

> Note that these are **block** tags, not **item** tags. They may (and probably will) be different! (F3 is your friend!)

> You can also use regular expressions with block filters, so `/^mekanism:.+_ore$/` would match any block from Mekanism whose ID ends with `_ore`. Keep in mind this will *not* match block state properties!

> When a `RuleTest` is required instead of a `BlockStatePredicate`, you can also supply that rule test directly in the form of a JavaScript object (it will then be parsed the same as vanilla would parse JSON or NBT objects). This can be useful if you want rule tests that have a random chance to match.

More examples of how targets work can be found in the example script down below.

Height Providers:

Another system that may appear a bit confusing at first is the system of "height providers", which are used to determine at what Y level a given ore should spawn and with what frequency. Used in tandem with this feature are the so-called "vertical anchors", which may be used to get the height of something relative to a specific anchor point (for example the top or bottom of the world).

In KubeJS, this system has been simplified a bit to make it easier to use for script developers. There are two common types of ore placement:

1. **Uniform**: has the same chance to spawn anywhere in between the two anchors
2. **Triangle**: is more likely to spawn in the center of the two anchors than it is to spawn further outwards

To use these two, you can use the methods `uniformHeight` and `traingleHeight` in `AddOreProperties`, respectively. Vertical anchors have also been simplified, as you can use the `aboveBottom` / `belowTop` helper methods in `AddOreProperties`.

Once again, see the example script for more information!

Example script

```
WorldgenEvents.add(event => {
  // use the anchors helper from the event
  const { anchors } = event

  event.addOre(ore => {
    ore.id = 'kubejs:glowstone_test_lmao' // (optional, but recommended) custom id for the feature
    ore.biomes = {
      not: 'minecraft:savanna' // biome filter, see above (technically also optional)
    }

    // examples on how to use targets
    ore.addTarget('#minecraft:stone_ore_replaceables', 'minecraft:glowstone') // replace anything in the vanilla stone_ore_replaceables tag with Glowstone
    ore.addTarget('minecraft:deepslate', 'minecraft:nether_wart_block')       // replace Deepslate with Nether Wart Blocks
    ore.addTarget([
      'minecraft:gravel',     // replace gravel...
      /minecraft:(.*)_dirt/   // or any kind of dirt (including coarse, rooted, etc.)...
    ], 'minecraft:tnt')       // with TNT (trust me, it'll be funny)

    ore.count([15, 50])             // generate between 15 and 50 veins (chosen at random), you could use a single number here for a fixed amount of blocks
      .squared()                    // randomly spreads the ores out across the chunk, instead of generating them in a column
```

```
      .triangleHeight(□□□□      // generate the ore with a triangular distribution, this means it
will be more likely to be placed closer to the center of the anchors
        anchors.aboveBottom(32),    // the lower bound should be 32 blocks above the bottom of
the world, so in this case, Y = -32 since minY = -64
        anchors.absolute(96)□      // the upper bound, meanwhile is set to be just exactly at Y
= 96
      )□□□□□□□□□              // in total, the ore can be found between Y levels -32 and 96, and
be most likely at Y = (96 + -32) / 2 = 32

    // more, optional parameters (default values are shown here)
    ore.size = 9                            // max. vein size
    ore.noSurface = 0.5                     // chance to discard if the ore would be exposed
to air
    ore.worldgenLayer = 'underground_ores'  // what generation step the ores should be
generated in (see below)
    ore.chance = 0□□□□□□□□            // if != 0 and count is unset, the ore has a 1/n chance to
generate per chunk
  })

  // oh yeah, and did I mention lakes exist, too?
  // (for now at least, Vanilla is likely to remove them in the future)
  event.addLake(lake => {
    lake.id = 'kubejs:funny_lake' // BlockStatePredicate
    lake.chance = 4
    lake.fluid = 'minecraft:lava'
    lake.barrier = 'minecraft:diamond_block'
  })
})

WorldgenEvents.remove(event => {
  // print all features for a given biome filter
  event.printFeatures('', 'minecraft:plains')

  event.removeOres(props => {
    // much like ADDING ores, this supports filtering by worldgen layer...
    props.worldgenLayer = 'underground_ores'
    // ...and by biome
    props.biomes = [
      { category: 'icy' },
      { category: 'savanna' },
```

```
      { category: 'mesa' }
    ]

    // BlockStatePredicate to remove iron and copper ores from the given biomes
    // Note tags may NOT be used here, unfortunately...
    props.blocks = ['minecraft:iron_ore', 'minecraft:copper_ore']
  })

  // remove features by their id (first argument is a generation step)
  event.removeFeatureById('underground_ores', ['minecraft:ore_coal_upper',
'minecraft:ore_coal_lower'])
})
```

Generation Steps

1. `raw_generation`
2. `lakes`
3. `local_modifications`
4. `underground_structures`
5. `surface_structures`
6. `strongholds`
7. `underground_ores`
8. `underground_decoration`
9. `fluid_springs`
10. `vegetal_decoration`
11. `top_layer_modification`

Revision #7
Created 2023-04-10 11:25:31 UTC by Lexxie
Updated 2024-08-11 22:24:35 UTC by Lexxie