AvatarMod-Docs Documentation

Release

CrowsOfWar

Sep 19, 2017

Contents

This site contains technical documentation for the AV2 project. The documentation is intended for developers of the mod only, as add-ons are not currently supported. It assumes you already have experience with the necessary Java and Forge concepts to develop mods. For end-user information on how to play the mod, please read the wiki at avatarmod2.wikia.com.

CHAPTER 1

Topics

Setup

This document shows you how to set up the AvatarMod development environment so you can start working on it.

Prerequisites

You need Git to work on avatar mod. You should have at least basic experience with Git before trying to develop AvatarMod.

Please have also good experience with Java and Forge before developing AvatarMod.

You must know how to open a terminal in this tutorial. Since you probably already know how to, and this varies by the operating system, this is not explained here.

Setting up

Clone the GitHub repository onto your computer using this command. It will create a folder called AvatarMod. .. code-block:: sh

> git clone https://github.com/CrowsOfWar/AvatarMod.git

The next instructions are specific to your IDE:

- *Eclipse*
- *IntelliJ*

Eclipse

If you are running Eclipse, run this commands:

```
./gradlew setupDecompWorkspace eclipse
```

Point Eclipse to the eclipse folder, and the MDKMinecraft project will show up on the package explorer. Eclipse has been successfully set up!

IntelliJ

If you are running IntelliJ, you don't need to use the command line any more and can use IntelliJ's built-in Gradle integration. Start by importing the Gradle project. If you are in the Welcome screen, choose `Import Project`; if you have a window open, choose `File > New > Project From Existing Resources`.

Navigate to the directory that you cloned the AvatarMod repo into, and then click on `build.gradle`.

If the dialog says "Gradle location not specified", check "Use gradle wrapper task configuration". Then click OK. This may take a while, but the project will open.

When the project opens, in the Gradle projects view, find the tasks `setupDecompWorkspace` and `idea`. Run them both, and you should be good to go.

Run Configuration

The last thing you need to do is configure your Run Configurations.

Sometimes run configurations aren't added, so just create them manually if that is the case. Make two configurations, one for the Client (what you use to play minecraft) and Server (the command line server without graphical capabilities). The client main class is `GradleStart`, and the server main class is `GradleStartServer`.

Finally, even if run configurations were automatically set up, you should add username to the run configurations. In the Program arguments section, add the following: `--username <YourName>`. This will ensure Minecraft will start with your username every time. If you don't do this, Minecraft will assign a different name every time so your AV2 progress will be reset each time you restart Minecraft.

Code standards

Here are some guidelines you must follow to maintain a good codebase.

Code formatting

Follow the Google Java Style Guide with the following exceptions:

- Static imports and wildcard imports are allowed

General tips

- You aren't gonna need it: If you think the system you are designing is overcomplicated, it probably is. Keep a clear idea of what the end goal is, so you won't waste time making an extra-extensible system if it's not going to be used
- Document your code changes here to clearly communicate with other developers. If you want to see what happens when you don't document complicated code, please visit world generation code. Have fun :)

- Remember, you are only going to write the code once (hopefully), but read it many times. So take the extra minute or two to make the code readable, instead of writing as little as possible but creating a confusing mess.

Specific requirements

Keep in mind that the Minecraft codebase doesn't always follow the rules, but we will since we are better than the obfuscated, poorly documented, garbled code that Minecraft consists of.

- Avoid one-line *if* statements (without curly brackets). Formatting can get weird
- Avoid switch statements. If-else chains are nicer to read, not much slower to write, and don't require the strange syntax that switch cases consist of (colons and break statements).
- Use tabs for indentation instead of spaces. If you are an avid space user: honestly, at the end of the day, tabs vs spaces doesn't really change your experience. With a proper IDE, you will still be pressing TAB key to indent and using CTRL+arrows to navigate.
- Acronyms should start uppercase but all subsequent letters are lowercase. eg Id or Json, but not ID or JSON.
- Use @*author* annotations. Git blame can also be used, but it's nice to have something explicitly in code

Documentation standards

- Javadoc is for reference, RTD is for explanations
- Assume the reader has knowledge of Java and the Forge API before reading the article
- Explain what each class is for and why it is used
- For player-related classes, explain whether the player can be offline
- Note server and client side availability
- At the end of each class page, include a cookbook section that has a few common or useful examples of code using that class

Git Conventions

The development of AvatarMod relies on git, which also provides valuable information about the project's tasks and history. For example, commit messages are an important summary of changes made.

To allow maximum readability of the git repository, you should follow these conventions.

Note: Disclaimer: While we encourage you to follow these conventions, we haven't followed these rules perfectly 100% of the time.

Commit Messages

Commit messages provide a summary of a commit, and possibly reasoning behind the changes.

The first lines of commit messages (the "subject") should show a summary of the commit. At a glance, someone should know what the commit basically does.

The subject should be one sentence. It should be worded imperatively. This means instead of "The dog gets exercise", you would write "Walk the dog". Omit ending punctuation (.!?). Some good examples include:

- “Tweak turbulence values of ‘furious’ lightning”
- “Rename ostrich armor ‘Platemail’ to ‘Plate”’
- “Mark BendingStyles#get(String) as nullable”

These messages make it easy to tell what’s going on.

Sometimes, if the reason for the change is not immediately obvious, additional explanation is needed. This should be added in the description part of the commit message. The description should be added after two line breaks of the first line.

A good example of a commit description is:

```
Fix duplicate scanning of same BlockPos in FloodFill

Position was only added to processedBlocks after it was
processed, but not when added to the queue. It was possible
for position to be added to the queue multiple times, causing
duplicate scans of the same position.
```

From reading the commit message, it might not be immediately obvious why there was a duplicate scanning issue, which was cleared up by the commit description.

Branch naming

The purpose of a branch name is to convey the feature, hotfix, or version in question in a concise and standardized manner.

For version branches, the branch name should simply be the version name (e.g., “a5.0”). For other branches, name the branch after the feature or fix in question. Words in branch names should be lowercase and separated with hypens. Examples:

- a5.0 (for the version branch for alpha 5.0)
- config-balancing (branch for adjusting default configuration values)
- fix-water-arc-crash (branch for crash hotfix; branches don’t need to explain things in detail)

Some smaller branches (like hotfixes or features) will realistically only be used by one person. In these branches, it is more acceptable to rebase or otherwise force push as nobody else cares. To denote one of these “personal” branches, use syntax `username/branch-name`. For `username`, type in your username (but it can be abbreviated if necessary); and for `branch-name`, use standard branch naming conventions as described above.

General tips

These tips don’t directly apply to conventions, but they can be useful for using Git more efficiently.

- On personal branches (see above), if the parent branch has progressed, use `git rebase` to avoid merge conflicts. Using `git merge` or `git pull` can cause many merge commits which make the repository history less readable. More information here.
- Merge conflicts can require a lot of knowledege about what’s going on, so feel free to ask another team member if you are confused by a merge conflict.
- While introducing a new system in the code, make sure you document it in the AvatarMod-Docs repository (ie, this website!).
- Pull requests aren’t always necessary, but if you think the change requires some discussion it’s probably a good idea.

Classes overview

This page explains the most important classes and interfaces used in AvatarMod that are necessary for basic understanding of the code.

Note: There is a full list of all documented classes *here <class-list.html>*.

Data classes

There needs to be a place to store information about players' bending, progress, and other information. It also must be compatible with mobs and other entities, so data is tied to a `Bender` (more information below). The `BendingData` interface provides the framework for such data. To maintain offline support, it is detached from the entity so you can access offline players' data.

Data and logic needs to not only apply to players, but also to mobs and other things. For example, abilities are compatible with both players and firebender mobs. For this to happen, there needs to be an abstraction. The `Bender` interface detaches logic from traditional `EntityPlayer` oriented code. Benders can represent mobs, players, or whatever else the implementer wishes. There are different implementations of this interface for players and mobs. (main article)

Since a `Bender` object represents an in world mob/player, usually access to an `Entity` is required to get position, motion, and other data. `Bender` objects must provide an `EntityLivingBase` which represents the entity version of the Bender. A player bender would provide an instance of `EntityPlayer`, while a firebender would provide an instance of `EntityFirender`. To maintain offline support, `Bender#getEntity()` can return null if the Bender is not currently in the world. This usually happens when there is a command being used about an offline player.

To obtain an instance of a Bender for your entity, call the static method `Bender.create(entity).` To obtain an instance of data, call `Bender.create(entity).getData()`, or, more conveniently, `Bender.getData(entity).`

Data is fully synchronized across server and client. Whenever data is changed, a packet is sent to the client containing the new data, so you won't have to worry about manually synchronizing data.

Bending controllers

Bending controllers represent each style of bending; mainly all of the abilities but also miscellaneous information about them (such as name). All bending controllers are derived from the `BendingController` class. The `BendingManager` class also keeps instances and IDs for each bending controller.

Bending controllers are stored in `BendingData`. They are found via ID. Each ID can be retrieved through the appropriate static fields in `BendingManager`, or by calling `controller.getId().`

Abilities

The `BendingAbility` class is the base for each ability. Obviously, the most important abstract method is `execute(AbilityContext)`, where the code for the ability is executed. Instances of `BendingAbility` are kept in static fields of the BendingAbility class itself. Their `BendingController` objects also keep track of them.

Any logic in an ability (such as the `execute` method) is only to be called on the server. Calling on client will result in issues like duplicate entities!

Classes Index

Ability Class

The `Ability` class represents an ability that benders can use. Abilities are singletons; they don't contain any information specific to a particular bender (AbilityData is used for this). `Ability` instances are most important because they define the in-game ability's behavior.

An instance of an ability can be obtained with `Abilities.get(abilityName)`, and abilities are executed by calling `Bender.executeAbility(abilityName)`. Avoid directly using an Ability instance unless you are sure that ability is installed (see Offline section).

Information common to all abilities include:

- A name
- The bending style this ability belongs to
- Functionality (i.e., the `execute` method)
- Misc. other information, such as chi cost

(More aspects of the ability like an entity are included only in some abilities)

Registry

Similar to BendingStyles, the `Abilities` class keeps track of all the abilities. In this class, all methods are static so you never should create an instance of Abilities.

An ability instance can be obtained by calling `Abilities.get(abilityName)`. Keep in mind that the ability can be null (explained below).

When creating new abilities, the new ability needs to be registered. This can be done by calling `Abilities.register(myability)` where `myability` is a new instance of your ability (remember, abilities are singletons).

There are several other methods in this registry, notably `Abilities.all()` which returns a list of all abilities.

Offline

From the Registry section, you may have noticed abilities can sometimes return null, and you need to handle for that. This is to avoid issues in scenarios where ability add-ons are uninstalled. For example, consider this scenario:

- A player installs an add-on which adds ability X
- The player creates data which references ability X
- The player decides to remove the add-on

At this point, data exists about ability X, but ability X is no longer present in the current installation. If the code can't handle an ability that isn't present, there is a good chance of fatal NPEs. The code must be able to handle that an ability isn't present (in this case, ignore data about ability X). If the user re-installs that add-on, the data would still be preserved thanks to the code's tolerance for a missing ability.

(A comparison of this approach to Forge's approach: A similar situation happens when a user uninstalls a mod that contained new items. Instead of gracefully ignoring these missing items from the world, Forge will actually delete all those items. Although it's a little easier to implement in the code, this is much worse for the user.)

To account for this problem, simply avoid using an actual instance of `Ability` unless you are *sure* it is loaded. Otherwise, handle using the ability's name.

tl;dr Opt for using ability's name instead of ability instance whenever possible

Naming

As explained above, abilities will often be referred to by their names. Although ability naming does not have any strict rules in place, they still should follow several conventions.

For consistency, names should be all lowercase, with underscores separating different words.

For add-ons, ability names should be namespaced to avoid naming conflicts. Follow vanilla Minecraft's ResourceLocation convention: the add-on name, followed by a colon, followed by the ability name. Ability names from the core AvatarMod need not follow this rule; they only contain the ability name (without `avatarmod:`).

Examples:

- AvatarMod ability "Light Fire": `light_fire`
- Add-on ability "Plasma Blast": `addon:plasma_blast`

Cookbook

(Note: there is a tutorial for how to create new abilities)

Find out which bending style an ability belongs to

```
String abilityName = /* ... */;

// Warning: ability can be null
Ability ability = Abilities.get(abilityName);
String bendingStyle = ability == null ? null : ability.getBendingStyle();
```

Have the bender use the fire arc ability (note: for AI, BendingAi should be used, which also teaches the bender *how* to use that ability)

```
Bender bender = /* ... */;
String abilityName = /* ... */;
bender.executeAbility(abilityName);
```

For an Ai mob - adds the ability Ai to the task list, which tells the mob how to use that ability

```
@Override
protected void initEntityAI() {
  int priority = /* ... */;
  String abilityName = /* ... */;

  BendingAi ai = Abilities.getAi(abilityName, this, this);
  // or...  ai = Abilities.get(abilityName).getAi(this, this)
  tasks.addTask(priority, ai);

  // ... rest of mob ai ...
}
```

Ability data

Any ability data is stored in the *AbilityData* class. Each AbilityData keeps track of a Bender's progress while learning a specific ability, for example:

- ability level
- experience to next level
- upgrades

There is one instance of AbilityData created per ability, per BendingData.

An instance of AbilityData can be retrieved by calling `BendingData#getAbilityData(BendingAbility)`.

Common Methods

Here are most of the methods you will be using in AbilityData:

- `#getLevel()` - returns the current level of the ability. The levels **start at 0**, so a value of 0 would represent level I, etc.
- `#isMasterPath(AbilityTreePath)` - checks if the ability is at level IV and on that upgrade choice. The upgrade path can either be `AbilityTreePath.FIRST` or `AbilityTreePath.SECOND`.
- `#addXp(float)` - adds the given amount of experience to the ability data. Keep in mind, at higher levels players get diminishing returns for the same amount of experience. This means if you grant, say, 10 XP, it might grant less than 10 XP to make it harder for the player to level up.

Cookbook

Check if an ability is unlocked

```
EntityPlayer player = /* ... */;
UUID abilityId = /* ... */;
AbilityData abilityData = AbilityData.get(player, abilityId);
boolean unlocked = !abilityData.isLocked();
```

Airblade's *pierce armor* is the second level IV upgrade. Checks if the player has the Pierce Armor upgrade - the player is at level IV, and the player chose the second upgrade path.

```
AbilityData abilityData = /* see above for info */;
boolean pierceArmor = abilityData.isMasterPath(AbilityTreePath.SECOND);
```

Add 10% XP. Won't advance to the next level.

```
BendingData data = /* ... */;
Ability ability = /* ... */;
AbilityData abilityData = data.getAbilityData(ability.getId());

// Don't use addXp since the number would be adjusted for XP decay
abilityData.addXpDirect(10);
```

Account UUIDs

Minecraft stores data about players by a `UUID`, which is unique per account. The reason Minecraft doesn't store information by username is because people can change it. If data was stored by username, and a user changed their username, all data about them would be deleted.

The `AccountUUIDs` class provides easy access to players' UUIDs by using the Mojang API. All methods in this class are static. You can always get a player's UUID even when they are offline.

Basic usage

To get a player's UUID from their username, use `AccountUUIDs#getId(username)`. To get a player's username from their UUID, use `AccountUUIDs#getUsername`.

Warning: When you have an `EntityPlayer` object and need to access their account UUID, don't use `player.getUniqueId()` or `player.getPersistentId()`. These are generic entity Ids and don't actually refer to the account Id. Instead, use AccountUUIDs: `AccountUUIDs.getId(player.getName())`.

AccountId

In some cases, AvatarMod may not be able to get an account UUID for the player:

- Problems with the internet
- Invalid/cracked account (username not registered)

When this happens, a "fake" UUID is generated based on the username; this allows the player to keep playing although the UUID is not the offical one from Mojang. This UUID is considered to be temporary. When Minecraft is restarted, AccountUUIDs will try to access the Mojang API again and get the offical UUID.

How does that related to the `AccountId` class? Well, this class is needed to keep track of whether a UUID is temporary or not, and therefore whether the Mojang API needs to be contacted again to try to get an official UUID.

The account UUID can be accessed by calling `AccountId#getUUID()`, and you can check if it is temporary by calling `AccountId#isTemporary()`.

Caching

To improve performance and reduce unneeded API calls, the AccountIds are cached both in a map at runtime, and on disk in a text file.

Cookbook

Get a player entity's account UUID

```
// do NOT use player.getUniqueId() or player.getPersistentId()
// these don't refer to the account's UUID
EntityPlayer player = /* ... */;
AccountId accountId = AccountUUIDs.getId(player.getName());
UUID uuid = accountId.getUUID();
```

Get a username based on account UUID

```
UUID uuid = /* ... */;
String username = AccountUUIDs.getUsername(uuid);
```

Using Avatar Entities

Many abilities create an entity as part of the move; for example, the fireball ability creates a fireball entity that's manipulated by the bender. All entities from abilities derive from the `AvatarEntity` class. Keep in mind AvatarEntity is *only* for ability related entities; mobs extend from other classes like `EntityAnimal`.

AvatarEntities all have an owner, which refers to who created the entity (using bending). Under normal conditions, the owner should always be set (though don't count on this). They also sometimes have a *controller*, which refers to who currently controls the movement of the entity. Sometimes nobody is manipulating the entity's movement, so the controller may not be set.

Nullability

Keep in mind that although the owner or controller of an AvatarEntity might be set, they may not be logged in, so an attempt to get the entity instance can return null.

Vectors

Unlike vanilla Entities, AvatarEntities integrate better with avatarmod vectors. Although vectors can already be used with vanilla entities, there are convenient methods to use Vectors for AvatarEntities:

- Use `position()` and `setPosition(vec)` to manipulate the entity's position
- Use `velocity()` and `setVelocity(vec)` to manipulate the entity's velocity. Velocity is in meters per second.

Warning: Don't use the vanilla method `getPosition()` - this returns the `BlockPos` at the entity's current position.

AvId and Lookup

Also unlike vanilla Entities, you can send references to an AvatarEntity over network. Vanilla entity Ids aren't synchronized between server and client, so they can't be used to synchronize entities. AvatarEntities have their own, synchronized Id which is called the AvId. The AvId is synchronized, so it can be used to send references across server and client.

If an entity needs to reference another entity, it's easier to use SyncedEntity instead.

There are several lookup methods for AvatarEntities. They only return a single entity:

- `lookupEntity(world, id)` - look up based on AvId
- `lookupEntity(world, cls, predicate)` - look up based on a type, and filter entities out
- `lookupControlledEntity(world, cls, controller)` - look up an entity which is controlled by the given entity

Hooks

AvatarEntities have many publicly callable hook methods. One example is the method `onLargeWaterContact()`, which should be called when the entity touches a large source of water. These hooks are available not only for convenience and conciseness, but also to promote interactions between different objects. Imagine a fire arc hits a water bubble. The water bubble calls `onLargeWaterContact()` on the fire arc, causing the fire to be extinguished. Hooks make interactions like this simple to implement without needing lots of special cases.

Hooks return `true` if the AvatarEntity was destroyed by the interaction (the AvatarEntity is responsible for calling `setDead()` on itself). For a list of hooks available to AvatarEntities, see the javadocs. You should call these hooks wherever appropriate.

Cookbook

Look up an AvatarEntity based on an id

```
int avId = /* ... */;
World world = /* ... */;

AvatarEntity avEnt = AvatarEntity.lookupEntity(world, avId);
```

Send the AvatarEntity flying towards their owner

```
AvatarEntity avEnt = /* ... */;

EntityLivingBase owner = avEnt.getOwner();
if (owner != null) {
  Vector ownerPos = Vector.getEntityPos(owner);
  Vector avEntPos = avEnt.position(); // can also use Vector.getEntityPos
  Vector direction = avEntPos.minus(ownerPos);

  Vector velocity = direction.normalize().times(10); // 10 m/s
  avEnt.addVelocity(velocity);
}
```

Creating Avatar Entities

The below information only is useful if you are editing a class extending AvatarEntity.

Collisions

AvatarEntities that wish to handle collisions should do so in the `onCollidieWithEntity` method.

Common operations

There are many operations that multiple AvatarEntities need to perform. These often are relatively simple yet involve a significant amount of code; for example breaking a block also involves dropping items, playing sounds, and creating particles. Several protected methods are available to simplify these operations:

- `breakBlock(pos)` - break a block at the specified BlockPos
- `spawnExtinguishIndicators()` - plays effects to indicate something is extinguished

Cookbook

Extinguish when hit water

```
@Override
public boolean onLargeWaterContact() {
  spawnExtinguishIndicators();
  setDead();
  return true;
}
```

Bender interface

In AV2, there are many types of benders. Players, mobs, and even other possibilites (statues?) can access bending abilities. The bender interface exists to abstract behavior so it can be applied to players or mobs. A `Bender` object only represents a Bender that is currently in the world (i.e. excluding offline players).

Features that the Bender interface provides includes:

- Access to bending data
- Convenient methods like `isPlayer()`
- Logic that varies based on player/mob, like `consumeWaterLevel(amount)` - players have water in their pouches - mobs have infinite water
- Way to identify benders (see BenderInfo)

A Bender object can be obtained for an entity by calling `Bender#get(EntityLivingBase)`. Benders are to be used on both logical sides.

Only some types of entities can be Benders. It will throw an exception if Bender is not supported for that type of entity.

Implementations

The `Bender` interface is implemented in the following ways:

- For players, `PlayerBender` wraps a player entity and returns results based on that
- Current mobs directly implement the Bender interface - see superclass `EntityBender`

BenderInfo

The `Bender` interface describes a Bender which is present in the world, but sometimes the bender must be saved in various scenarios:

- Ability to identify offline players
- Saving information tied to offline players
- Sending Benders across the network (the `Bender` cannot be directly sent because there are different instances of that `Entity` on server/client)

BenderInfo allows you to identify Benders (either players or mobs) by using their UUID. It can be used both on the server and client.

Internally, BenderInfo uses several subclasses to simplify implementation. Therefore, you cannot instantiate a BenderInfo directly. There are several ways to get a BenderInfo object: - `Bender#getInfo()` - get BenderInfo about a particular bender - `BenderInfo#get(EntityLivingBase entity)` - get BenderInfo for a particular entity, provided that it is a Bender - `BenderInfo#readFromBytes(ByteBuf) or BenderInfo#readFromNbt(NBTTagCompound)` - read BenderInfo from disk or network. There are also writing methods that you can call once you have a BenderInfo instance. - `BenderInfo#get(boolean player, UUID id)` - get BenderInfo using your own parameters. This isn't recommended as there are simpler ways to do it (see above)

- BenderInfo finds the Bender by iterating through the world's loaded entities, and creating a new bender object off of that player
- In the case of players, uses account's UUID
- In the case of mobs, uses its randomly generated UUID

There is a subclass called `NoBenderInfo` which represents no bender found. This uses the null object pattern and can prevent NPEs.

BenderInfo is written to disk using `writeToNbt`, and can be read by calling the static method `readFromNbt`. It is written and read from network via the BenderInfo serializer found in `AvatarDataSerializers`.

Cookbook

Check if an entity (which implements Bender) is currently flying

```
EntityLivingBase entity = /* ... */;
Bender bender = Bender.create(entity);
boolean flying = bender.isFlying();
```

Store information about a Bender to disk so it can be accessed later

```
NBTTagCompound nbt = /* ... */;
Bender bender = /* ... */;
BenderInfo info = bender.getInfo();
info.writeToNbt(nbt);
```

Read the BenderInfo and create a new Bender

```
NBTTagCompound nbt = /* ... */;
World world = /* ... */;
BenderInfo info = BenderInfo.readFromNbt(nbt);
Bender bender = info.find(world); // can be null if Bender not found
```

Bending data

Information about *benders' <bender.html>* progress is stored in BendingData instances. This information includes unlocked *bending styles <bending-style.html>*, *ability data <ability-data.html>*, and more. Information here is persistent and synchronized between client and server.

Retrieval

A `BendingData` object can be obtained by calling the following methods:

- `BendingData.get(entity)` - gets data for that entity, if it's a bender
- `BendingData.get(world, accountId)` - gets data for the player who has that *account Id <account-uuids.html>*. The player can be offline.
- `BendingData.get(world, username)` - gets data for the player who has that username. The player can be offline.

If you have a *Bender <bender.html>* object, you can also use `Bender#getData()`.

Bending Styles

There are many different methods to use the large amount of data contained in BendingData.

Some useful methods relevant to the current unlocked bending styles include:

- `hasBending(UUID)` - whether the bending style with that Id is unlocked

- `addBending(UUID)` - unlocks the bending style with the given Id
- `removeBending(UUID)` - revokes the bending style with the given Id
- `getAllBendingIds()` - gets a list of the Ids of the currently unlocked bending styles

There are also overloads to avoid using a BendingStyle's Id as parameters and pass a BendingStyle instance instead. This can be convenient, but keep in mind a BendingStyle *may not currently be present <../offline.html>* so these methods should only be used when you know the BendingStyle isn't null.

Ability data

There are also methods pertaining to *ability data <ability-data.html>*. Ability data keeps track of a specific ability's level, Xp progress, and unlocked status.

The most relevant method for dealing with ability data is `getAbilityData(UUID)`, which gets ability data for the ability with that Id. As with methods for BendingData, there's an overload to pass in an actual `Ability` instance, but this should only be used in code where the ability is guaranteed to be loaded.

There are a few other methods for ability data, but this is the only useful one 90% of the time.

Status Controls

Status controls <status-control.html> are temporary key listeners that execute some code upon being pressed. (keeping StatusControls in synchronized BendingData is necessary because SCs need persistence)

Here are a few methods to access status controls:

- `hasStatusControl(StatusControl)` - whether the status control is present
- `addStatusControl(StatusControl)` - add the status control
- `removeStatusControl(StatusControl)` - remove the status control
- `getAllStatusControls()` - get all current status controls

Tick Handlers

Data can have multiple *tick handlers <tick-handler.html>*. As their name implies, a tick handler is an object which recieves updates every tick. Any tick handler in the BendingData will recieve updates if the Bender is online.

The method `addTickHandler(TickHandler)` activates the tick handler. An instance of a tick handler is found from a public static variable instance in the `TickHandler` class.

BendingData keeps track of how long a TickHandler was used for. This can be accessed using `getTickHandlerDuration(TickHandler)`, which represents the time in ticks. The time isn't synchronized between the server and client, and isn't saved to disk, so avoid using this timer for anything other than quick, short-term delays.

Miscellaneous

Some extra information is stored in BendingData. This information isn't very complex, but needs to be persistent so is stored in BendingData. One example is bison follow mode: whether tamed flying bison should follow the player. These bits of information are found from `getMiscData()` but there are delegates to MiscData in the BendingData class.

Registries

There are several places where instances of `Ability` and `BendingStyle` can be stored.

Vector

A `Vector` object represents an immutable, three-dimensional mathematical vector. Vectors can represent many things, such as velocity or position.

Warning: This refers to the class `com.crowsofwar.gorecore.util.Vector` class, not in any related openGL or JOML packages

Basic Usage

Vectors can be created using the appropriate constructor, such as `new Vector(x, y, z)`. Their coordinates can be retrieved via the `x()`, `y()`, and `z()` methods.

Vectors have many different methods to perform basic computations. As a vector is immutable, these methods create a new vector based on the old one and the parameters. Methods use the fluid interfaces naming convention. For example, addition is performed via the `plus` method.

Method chaining is encouraged. Consider the following code:

```
Vector motion = /* ... */;
motion = motion.times(2);
motion = motion.plusY(1);
motion = motion.times(-1);
```

The same could be accomplished much more concisely via method chaining:

```
Vector motion = /* ... */;
motion = motion.times(2).plusY(1).times(-1);
```

Trigonometry

There are also several methods that deal with physics and trigonometry.

- To convert from and to spherical (polar) coordinates, use `Vector#toSpherical()` or `Vector#toRectangular()`
- To get rotations between vectors, use `Vector#getRotationTo(from, to)`
- To reflect vectors off a normal, use `Vector#reflect(normal)`

Rotations

For vectors which represent rotation (Euler angles), the y-component represents yaw and the x-component represents pitch. The z-component is unused as Minecraft doesn't support roll. Methods will say whether the rotation is in degrees or radians.

Minecraft integration

Vectors were designed to be used in a Minecraft environment, so there are some methods which allow convenient usage of Vectors with entities.

- To get an entity's position, use `Vector#getEntityPos(entity)`. To get their eye position, use `Vector#getEyePos(entity)`
- To get an entity's motion, use `Vector#getVelocity(entity)`. The velocity is in meters per second (20 ticks).
- To get the direction in which an entity is looking in rectangular coordinates, use `Vector#getLookRectangular(entity)`
- To convert to a BlockPos, use `Vector#toBlockPos()`
- To find the direction to lob a projectile, use `Vector#getProjectileAngle(velocity, gravity, horizontalDist, verticalDist)`

Cookbook

Find the direction to other location

```
Vector us = /* ... */;
Vector them = /* ... */;
Vector directionTo = Vector.getRotationTo(us, them);
double yaw = directionTo.y();
double pitch = directionTo.x();
```

Check if two positions are within 10 blocks of each other

```
Vector pos1 = /* ... */;
Vector pos2 = /* ... */;
// Use Square Distance here to avoid a sqrt operation
// Sqrt is expensive so avoid it if possible
double distSq = pos1.sqrDist(pos2);
boolean within10Blocks = distSq <= 10 * 10; // if distance is squared, need to square␣
↪other side as well
```

Find the position of where the player is looking, 3 blocks ahead

```
EntityPlayer player = /* ... */;
Vector position = Vector.getEyePos(player); // use getEyePos not getEntityPos
Vector look = Vector.getLookRectangular(player);
Vector lookPosition = position.plus(look.times(3)); // using .plus() here since .
↪add() would modify position
// Note: this shouldn't be used for calculating the block player is looking at
//  since it doesn't take into account any blocks in the way of the player
// For that purpose, use Raytrace instead
```

Tutorials Index

Tutorial: New Ability

This tutorial will show you how to add a new ability to the mod. The ability in this example will be a Firebending ability that lights the targeted mob on fire.

Basic Setup

To make the new ability, extend the Ability class. You need to call the super constructor, which takes the name of the ability as well as its bending controller. The name of the ability will be "immolate", and the name of the bending controller is "firebending".

Also, the abstract methods `execute` and `getChiCost` need to be implemented - leave them blank and fill them in later.

```
public class AbilityImmolate extends Ability
```

Simply creating a class isn't enough, though - you'll need to register the ability to the ability registry so it's added to the game. In `AvatarMod#registerAbilities()`, call `Abilities#register` with a new instance of your ability (abilities are singletons).

Note: If you're developing an add-on, call these methods in your main mod class's `postInit` method. If your main mod class doesn't have a postInit, just add one which has the method header `@EventHandler public void postInit(FMLPostInitializationEvent e)`. FML will automatically call it, similar to how preInit is called.

The ability is now registered and visible in the Firebending radial menu.

Execute method

Now, it's time to fill in the `execute` method. It is called whenever the ability is used, only on the server side. This is where you define the ability's functionality.

This ability will light the targeted entity on fire. The basic logic flow will be to use a raytrace to find the targeted mob and then set it on fire.

Here is an example implementation of that ability.

```
@Override
public void execute(AbilityContext ctx) {
  final double range = 5;
  final float fireSeconds = 4;

  World world = ctx.getWorld();
  EntityLivingBase entity = ctx.getBenderEntity();

  Vector start = Vector.getEyePos(entity);
  Vector dir = Vector.getLookRectangular(entity);
  List<Entity> raytrace = Raytrace.entityRaytrace(world, start, dir, range);

  // only immolate 1 entity
  if (!raytrace.isEmpty()) {
    raytrace.get(0).setFire(fireSeconds);
  }

}
```

Here are links to detailed docs on each class used:

- AbilityContext
- Vector

- Raytrace

Improving the implementation

Although the ability works, there are still a lot of missing features common to other abilities. The ability doesn't require chi, so can be spammed continuously. It ignores experience and level 4 options, and therefore has no progression. Fortunately, adding these features are pretty simple.

Applying Xp cost is super easy. Just override the super method `getChiCost()` and return the correct amount of chi. To require more chi under other conditions (such as air bubbles consuming chi each second), see `Bender#consumeChi`.

Adding experience only requires a few method calls, but knowledge of the ability data class is necessary. An instance of ability data can be obtained through `AbilityContext#getAbilityData()`. Simply adjusting the numbers can be sufficent:

```java
@Override
public void execute(AbilityContext ctx) {
  AbilityData abilityData = ctx.getAbilityData();
  double range = 3 + abilityData.getLevel() * 0.5;
  float fireSeconds = 2 + abilityData.getLevel();

  if (abilityData.isMasterPath(AbilityTreePath.FIRST)) {
    range = 8;
    fireSeconds = 3;
  }
  if (abilityData.isMasterPath(AbilityTreePath.SECOND)) {
    range = 3;
    fireSeconds = 7;
  }

  // etc...
}
```

To award Xp, just call `abilityData.addXp` when appropriate.

Localization

For this ability, the following translations will be necessary:

- Name of the ability: `avatar.ability.immolate`
- Description of the ability: `avatar.ability.immolate.desc`
- Description of each level upgrade: `avatar.ability.immolate.lvlX` (see below)

For levels the following levels are used:

- `lvl1` - level I
- `lvl2` - level II
- `lvl3` - level III
- `lvl4_1` - level IV, first path
- `lvl4_2` - level IV, second path

Simply add the keys to en_US.lang, for example:

Icon

There are a few icons needed, one for radial menu, one for the skills menu card, and one for the skills menu background.

For the radial menu icon, use a 256x256 image and name it `icon_immolate.png`. Place it in the `textures/radial` folder of the assets.

For the skill menu card and skill menu background, don't bother because by the time someone else is reading this article, I'll have scrapped that approach as EduMC can't make skill menu pictures anymore.

AI

The ability is almost all set up - functionality, UI, translations... The only thing left is an Ai module, which tells bending mobs how to use this ability (this part is optional if bending mobs shouldn't use the ability).

For starters, override the method `getAi`, and see the `AiWaterArc` class for an example. Bending Ai is not covered in this tutorial; detailed documentation is available here.