Zeta

Jun 12, 2019

Contents

Zeta is an open-source top-down fully moddable 2D RPG singleplayer game engine, written in Java using LibGDX, under the Apache 2.0 License. Games are written in Java as mods to the base engine, which ensures that every aspect of Zeta can be tweaked to the design of your project.

Zeta is written using LibGDX, an open-source Java-based game development framework created by badlogicgames licensed under the Apache License 2.0.

CHAPTER 1

Setup

This guide will show you how to set up a project with the Zeta engine.

1.1 Compiling the Zeta library

Zeta uses the Gradle build tool. If you're on Windows, open a command line and run `./gradlew desktop:dist` in Zeta's main folder. If you're on a Unix system, run `./gradle desktop:dist` instead. This will create the `zeta-1.0.jar` binary, which will be located in the `desktop/build/libs` folder.

1.2 Writing code with Zeta

In your IDE, make sure to add the Zeta jar file as a dependency to your project so that it can detect Zeta's functions and provide hints. Alternatively, link it to Zeta's source code so that you also get quick access to documentation.

1.3 Linking to Zeta for a base game

When compiling and running your project, make `com.thirds.zeta.desktop.DesktopLauncher` the main class, and put the Zeta binary in your game's classpath. You should now be able to test Zeta. You don't need to write any code at all at this point, Java should automatically invoke Zeta's initialisation methods.

1.4 Linking to Zeta with mods

Zeta detects all mods in its classpath, so if you want to allow support for end-user mods as well as your game mod, make sure to run your game with a command like `java -cp * com.thirds.zeta.desktop.DesktopLauncher` rather than referencing a jar directly. End users can then place their own mods in the folder you added to the classpath.

1.5 Next Steps

Check out the modding guide page for more information on how to get started with Zeta!

CHAPTER 2

Modding

Zeta's engine is designed around the core functionality of mods. Anything you make on top of Zeta is a mod; even a game is just a mod running on Zeta. This modularity allows for greater flexibility in the engine's internal workings. Please read the setup page first - make sure you can run the Zeta engine before trying to mod it.

2.1 Creating a mod

To create a mod for Zeta, simply create a class anywhere which implements `com.thirds.zeta.mod.Mod`. When Zeta initialises, it will search for all mods in its classpath and load them. You need to implement the methods `getName()` and `getVersion()`. The mod name should be written entirely in ASCII lower case, with digits and hyphens. Version number is arbitrary, but will be used for dependency management in the future.

2.2 How to use a mod

2.2.1 Creating entities

To make your own entities for use in Zeta's levels, or to see how to use Zeta's existing entities, visit the entities page.

2.2.2 Injecting code into Zeta

Zeta has lots of interesting ways to hack into its engine. One commonly used method is called a *hook*; these functions allow you to execute your own code when Zeta calls certain functions. They are documented here.

CHAPTER 3

Entities

In the Zeta engine, an entity is any object in a level. This can be as simple as a collision brush or a point light, or as complicated as a player character or an AI-driven NPC. All entities must follow the same framework:

- They all inherit from `com.thirds.zeta.entity.Entity`
- All non-abstract entity classes must have only a single, public constructor, which takes an EntityData object. This data is the position, life, physics information and entity ID loaded from level files and save files.

3.1 Creating entities

3.1.1 In a level file

Coming soon - the level editor isn't implemented yet!

3.1.2 Programmatically

Sometimes, you will want to create entities on the fly during a level. Never call the entity's constructor directly since it won't get registered properly into the level; instead, use the EntityFactory. Provide it a class and some entity data and it will handle calling relevant constructors and registering the entity into the level for you.

The entity will instantly become available for you to call methods on, but its update/render methods won't fire until the next frame. This ensures that each entity always gets one complete frame's worth of updates at a time - if update was called, you know that preUpdate was also called just before.

Never reuse the same EntityData for initialising multiple entities! This will mess with the ID system.

3.2 EntityData in detail

Here is a detailed description of the EntityData class. You can get any entity's EntityData by calling getData.

3.2.1 Coordinates

Entities use a 3-dimensional vector to store position data. The first two components comprise the x- and y-coordinates of the entity. The bottom left of the screen is (0, 0). The z-coordinate is used to control which entities get updated and rendered before which others, but you shouldn't need to worry about it in most cases. This is discussed in more detail below.

3.2.2 Size

An entity's size expands upwards and to the right (fitting with the coordinate system). Size has little in-built usage in Zeta apart from with physics and collision detection. The method getCentre finds the coordinates of the centre of the entity by using the size.

3.2.3 Physics

Entities have no collision detection by default. By setting the physics object, you can enable this functionality. You should ensure that any physics object you create matches the size of the entity. By calling setPhysicsObjectToAABB, Zeta can create a rectangular physics object that matches the size of the object. If a physics object is solid, players can't walk through them. Otherwise it is only used for custom collision detection, including triggers.

3.2.4 Life

By calling getLife, it tells you how many seconds the entity has been alive (in the level) for.

3.2.5 Name

You can give entities a custom name when in the level editor. This has no intrinsic functionality, apart from being useful when trying to find specific entities programmatically.

3.3 Event methods and invocation order

3.3.1 Event methods

Entities are event-driven. Once they are created, they will have a number of methods automatically called by the level that they're in.

preInit, init, postInit Called as soon as the entity is created. Always use these methods instead of constructors to initialise data, because constructors can be quite volatile when dealing with save files (and they're harder to work with when modding), so Zeta provides this useful alternative.

preUpdate, update, postUpdate Called once every frame while the game is not paused. Takes in a single argument, delta, which is the time in seconds since the last frame.

preRender, render, postRender Called once every frame whether or not the game is paused. Takes in a single argument, delta, which is the time in seconds since the last frame. Uses the level's coordinate system, drawing below all UI objects.

preRenderUI, renderUI, postRenderUI Called once every frame whether or not the game is paused. Takes in a single argument, delta, which is the time in seconds since the last frame. Uses the screen coordinate system, drawing above all game objects.

dispose Called when the entity is removed from the level and when you might need to clean up data or child entities.

resize Called when the window changes size.

For detailed descriptions on how to use the render methods, look at LibGDX's documentation on how to use the SpriteBatch, since Zeta uses their libraries for rendering. Don't use your own sprite batch, just use the one provided in the render methods! If tinting the sprite batch, reset it to white after you're done with rendering your entity.

3.3.2 Event rules

As a rule of thumb, put game mechanics in the update method family and put purely graphical mechanics in the render method family. Try not to mix-and-match; weird things may happen when the game is paused or running on a slow machine!

3.3.3 Invocation order

Every frame, the event based methods are called on every entity (as in, all preUpdates happen before any updates or postUpdates) in the following order:

- preUpdate (and updateMovement if applicable)
- update
- postUpdate
- preRender
- render
- postRender

3.3.4 Highest first z-order

The order in which entities get updated within this list is determined by their z-order, more specifically a highest first order. For example, an entity with a z-coordinate of 100 will be rendered before one with a z-coordinate of -250. When changing the position of an entity, Zeta automatically adjusts its z-coordinate to match its y-coordinate (you can do this manually using the setZFromY method). This means that entities are updated and rendered from the top of the screen downwards, ensuring that entities at the bottom of the screen correctly visually overlay entities that should appear behind them.

3.4 Other methods

3.4.1 Die

By calling this method, the entity is removed from the game at the start of the next frame, and its dispose method is called. You can also call LevelScreen.getInstance().getLevel().removeEntity(ent).

3.5 Save files

Zeta's save file system serialises all entities, and allows you to choose which fields to serialise and which to freshly initialise yourself when loading from a save file. By marking a field as transient (e.g. `private transient int`

`variableName;`), this tells Zeta not to save it into the save file. This allows the save file to be smaller and faster to load. Here is a list of object types you may want to make transient:

- Graphics objects, because they are easy to initialise in your init method, for example:
  - AtlasRegions
  - Textures
- Other asset files like sounds and music, for the same reason.
- Other entities as fields

Do not reference another entity as a field in your entity (TODO warn/error when this occurs). When loading back from a save file, it won't load into the level properly and may even create two copies of the entity; one from the level itself and one from the field. In the worst case, two entities reference each other and produce an infinite loop trying to save all the data! Instead, always use EntityRefs to reference other entities.

Calling the method ref on an entity gives you a reference that can be queried to retrieve the original entity, and can also be saved as a field. This EntityRef object has a method, get, which returns the original entity that was referenced. Another (perhaps more useful) method is refs, or "reference specific". This gives you an EntityRef.Specific which enforces that the entity it references is of the correct class.

Bear in mind that if you want to put an Entity in a transient field, that is fine because transient fields don't get saved.

3.6 Queued actions

QueuedActions are like alarms that fire events at some specified time in the future, during one (or more) of the entity's update and render methods. To create a queued action, always use a concrete class not a lambda, since save files can't reconstruct anonymous classes. Use the queue method to install it into the entity. The life parameter in the QueuedAction constructor shows how many seconds it should wait for before executing the action.

CHAPTER 4

Levels

In Zeta, a level contains and manages every entity, and levels themselves are handled by a LevelScreen.

4.1 How to access the level objects

Level functionality is split between several classes: the Level, the LevelRenderer and the LevelPhysics. They all have accessors in the LevelScreen class. For example, to get the level renderer, write *LevelScreen.getInstance().getLevelRenderer()*. You can hook the constructors of both Level and LevelRenderer.

4.2 Level object

4.2.1 User functions

Here are some useful methods in the level object.

find Allows you to find entities that match a class or superclass, and optionally also a name.

entityExists Allows you to tell whether an entity is currently in the level. If not, that entity won't be saved properly.

4.2.2 Internal functions

These functions are called internally and shouldn't normally be used. Alternatives are listed.

addEntity Use EntityFactory.generate.

removeEntity Use Entity.die.

reorder Use Entity.setPos to let Zeta reorder the entity within the z-order.

update Called automatically every frame by Zeta internally.

4.3 Level renderer object

4.3.1 User functions

unproject Allows you to project UI coordinates (where (0, 0) is the bottom left, and the window dimensions dictate the size) into in-game coordinates (which is, for example, the coordinate system entities are stored in).

getMouseCoords Returns the UI coordinates of the mouse.

getUnprojectedMouseCoords Allows you to find the location of the mouse pointer in game coordinates.

4.3.2 Internal functions

render Called automatically every frame by Zeta internally.

resize Called automatically whenever the window changes size by Zeta internally.

dispose Called automatically when the level is exiting by Zeta internally.

4.4 Level physics object

4.4.1 User functions

collideAny(Solid) Returns the first entity whose physics object intersected the provided entity's physics object. If the solid variant is called, only solid physics objects count. It returns null if no object was hit.

collideAll(Solid) Returns the list of all entities whose physics objects intersected the provided entity's physics object. It returns an empty list if no object was hit.

collideAnyCharacter Returns the first GameCharacter whose physics object intersected the provided entity's physics object. It returns null if no object was hit.

CHAPTER 5

Hooks

Hooks allow mods to inject their own code into some of Zeta's functions.

5.1 Example

For example, Zeta provides a hook into the constructor of the `LevelRenderer` class. In order to hook into that constructor, you need to use the `HookManager` object in that class, called `LevelRenderer.constructorHM`. It has two methods that you can use, `addPreHook` and `addPostHook`. If you add a hook using `addPreHook`, your code will be executed just before the default behaviour in the constructor. If you instead use `addPostHook`, your code will be executed just after the object initialises. Here is an example of using this hook to change the background colour of the render engine.

```
LevelRenderer.constructorHM.addPostHook(args -> {

    LevelRenderer levelRenderer = args.getA1();
    Level level = args.getA2();

    levelRenderer.setGlClearColour(Color.MAGENTA);

    return null;
});
```

Add the above code to your mod's init method to try it out. Note that the arguments to the function (including the 'this' pointer to the LevelRenderer) are encapsulated inside an args variable.

CHAPTER 6

Indices and tables

- genindex
- search