FREE eBook

LEARNING libgdx

Free unaffiliated eBook created from **Stack Overflow contributors.**

#libgdx

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: libgdx

It is an unofficial and free libgdx ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official libgdx.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with libgdx

Versions

Version	Release Date
1.9.3	2016-05-16
1.9.5	2016-12-11

Examples

Installation or Setup

LibGDX has a fairly simple setup, with the help of a simple Java program. You can find the download here. When you startup the application, it will look something like this:

Note: This screenshot have been taken on Linux and shows path that differs from a Windows installation. However, the form is the same on any OS compatible with this setup application

Input boxes

In the "Name" input box, is the name of the game for your project. The "Package" input box, is the package to your main class. The "Game Class" input box, is the main class that is called when your game is run. The "Destination" input box, is the destination to where your project is going to be generated. The "Andriod SDK" input box, the path to where your android sdk is. This input box is completely optional, so if you do not want to deploy your application to Android, you do not have to worry about this.

Sub Projects

Sub projects are just the platforms that are to be deployed to. That part is pretty self explanatory. If you do not want to deploy it to HTML for instance, just simply uncheck the check box.

Extensions

Extensions are the official LibGDX extensions. Here is a table telling you what each one is:

Extension name	Description
Bullet	Bullet is a 3D Collision Detection and Rigid Body Dynamics Library.
Freetype	Freetype allows you to use .ttf fonts, rather than having to use Bitmap fonts
Tools	Allows you to implement the output of LibGDX tools.
Controllers	Allows you to implement controllers like XBox 360 controllers.
Box2d	A physics library for 2d games.
Box2dlights	Allows easy way of adding soft dynamic lights to a physics game.
Ashley	A tiny entity framework
Ai	An artificial intelligence framework.

You can add Third Party Extensions, but their details or names will not be shown here.

Advanced

In the Advanced section you can set several settings and generate extra project files for Eclipse and IDEA IDE.

Setting name	Description
Maven Mirror URL	Replaces Maven Central with the provided Maven URL
IDEA	Generates Intellij IDEA project files
Eclipse	Generates Eclipse project files
Offline mode	Don't force download dependencies

Generation

Once you have all of your settings correct, you can push the "Generate" button. This may take a couple of seconds, but it will generate the basic files, and Gradle needed for your project. Once your done with that, it is time to import the project to your IDE.

LibGDX Hello World

Basics

The generated projects contain a basic Hello World-like application already implemented.

The main project is the core project, that contains all platform-independent code. This is mandatory, but based on your generation settings you can have several more projects for every platform that you selected.

The example

Open `com.mygdx.game.MyGdxGame.java` in the `core` project. You will see the following code:

```
public class MyGdxGame extends ApplicationAdapter {
    SpriteBatch batch;
    Texture img;

    @Override
    public void create () {
        batch = new SpriteBatch();
        img = new Texture("badlogic.jpg");
    }

    @Override
    public void render () {
        Gdx.gl.glClearColor(1, 0, 0, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        batch.begin();
        batch.draw(img, 0, 0);
        batch.end();
    }

    @Override
    public void dispose () {
        batch.dispose();
        img.dispose();
    }
}
```

Although this is your main project , you will not execute it directly, you always have to run the platform-specific Launcher, for desktop it is called `com.mygdx.game.desktop.DesktopLauncher.java` in the `desktop` project.

```
public class DesktopLauncher {
    public static void main (String[] arg) {
        LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
```

```
        new LwjglApplication(new MyGdxGame(), config);
    }
}
```

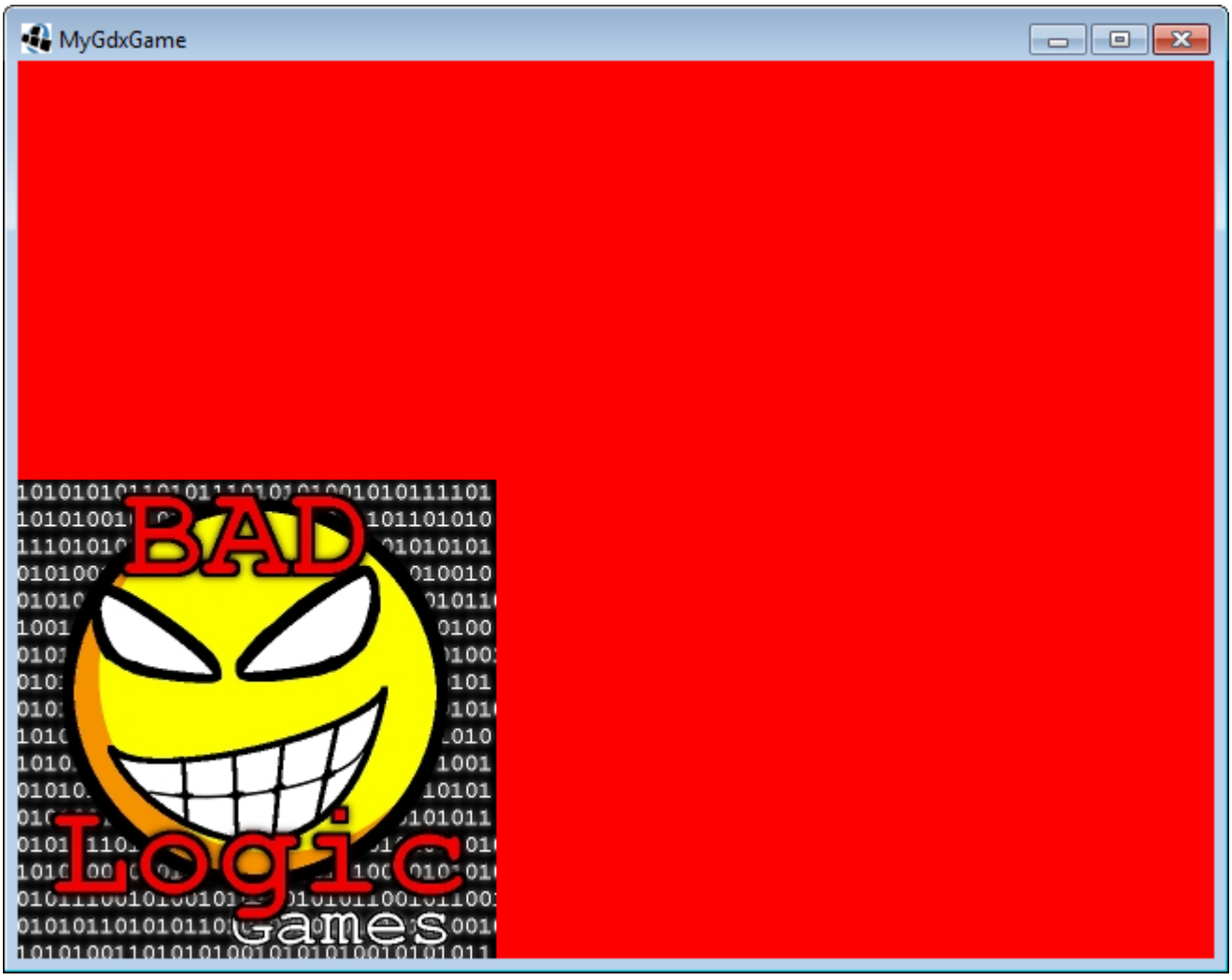
In this class you can set platform-specific settings.

Running your game

Eclipse and Intellij use two different methods, to run your project. Find the IDE your using below.

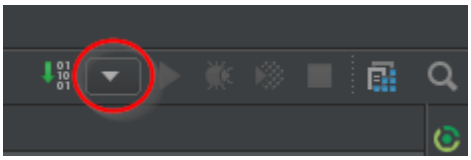
Running using Eclipse

Using eclipse, you can run your application by running this class as a Java Application(Right-click on project -> Run as -> Java Application). You will see the following window:

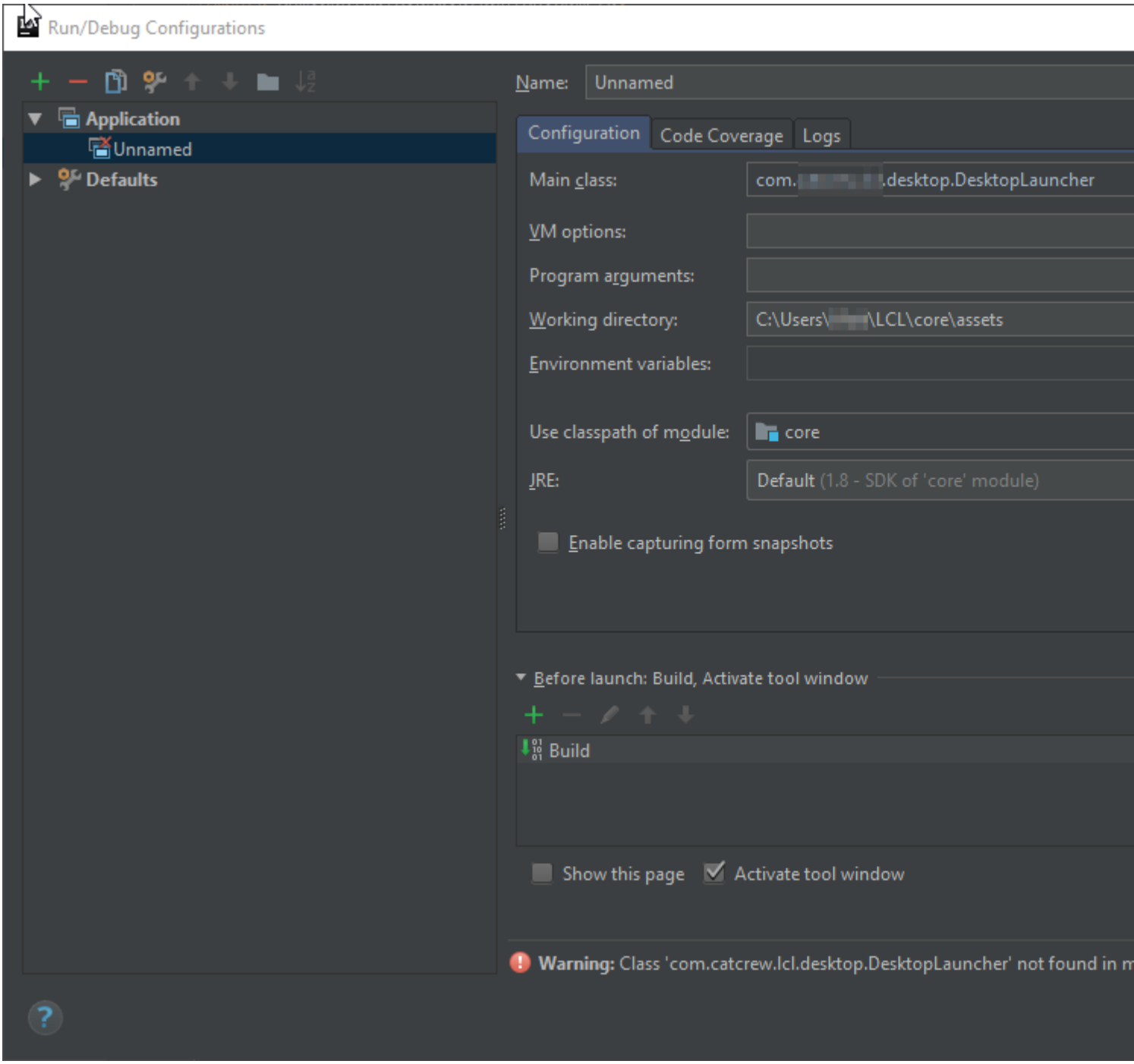


Running using Intellij Idea

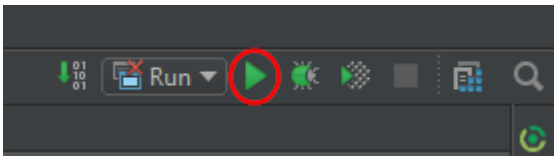
In Intellij, you will need to make a run configuration. To do this, you must click the button in the top right that looks like a downward carrot:



Then click the "Edit Configurations..." button. You will be presented with a screen of all of your current run configurations. Click the "+" in the top left of that window, and select the "Application" option. From there, select the "DesktopLauncher" file for the "Main class" option, set the "Use classpath of module" to the core option, and set "Working directory" to the assets folder in your core folder. The end product should look something like this:



Once you have done that, you can select a name for your run configuration, and click "Apply", then "OK". Once your done with that, you can click the green run icon in the top right:



What is happening in MyGdxGame class?

First the `create` method is called, that will initialize the batch that draws to the screen. Then the method loads the badlogic.jpg into the memory.

After this the `render` method is called repeatedly until the application is stopped. This method will reset to background color to red and draws the image on the screen. As you can see you always have to begin and end the batch drawing. Lastly when the application is about to stop the `dispose` method is called, that will free the memory space used by the texture and the batch.

(It's good to know that disposing can happen at runtime as well on other platforms e.g. Android, because Android might free up memory space when applications are in the background, but this is a more advanced topic)

Remark: If you get an error on running like below check this question for answer!

> File not found: badlogic.jpg (Internal)

LibGDX General Overview

LibGDX is a free, open-source game-development library developed in Java. It's goals are to allow users to develop cross-platform games that run on desktop, Android, iOS, and web browsers. Write code once, deploy it to any of the major platforms.

Adding support for platform-specific code

LibGDX is designed in a way that you can write the same code and deploy it on several different platforms. Yet, there are times when you want to get access to platform specific code. For an instance, if you have leaderboards and achievements in your game, you may want to use platform-specific tools (like Google Play Games) in addition to storing them locally. Or you want to use a database, or something completely different.

You can't add this kind of code into the core module. So the first step is to create an Interface. Create it in the core module. This first one is a utility to manage other interfaces:

```
public interface PlatformWrapper{
    //here you can also add other things you need that are platform specific.
    //If you want to create your own file saver for an instance, a JSON based database,
    //achievements, leaderboards, in app purchases and anything else you need platform
specific code for.
```

```
    SQLWrapper getSQLSaver();//This one will be used in this example
    AchievementWrapper getAchievementHandler();//this line is here as an example
}
```

Then, we need to create a second interface, the SQLWrapper. This one also goes in the core module.

```
public interface SQLWrapper{
    void init(String DATABASE);
    void saveSerializable(int id, Object o);
    Object loadSerializable(int id, Object o);
    void saveString(int id, String s);
    //.... and other methods you need here. This is something that varies
    //with usage. You may not need the serializable methods, but really need a custom method
for saving
    //an entire game world. This part is entirely up to you to find what methods you need

    //With these three methods, always assume it is the active database in question. Unless
    //otherwise specified, it most likely is
    String getActiveDatabase();
    void deleteDatabase();
    void deleteTable(String table);//delete the active table
}
```

Now, you need to go into every project and create a class to implement PlatformWrapper and one to implement SQLWrapper. In each project you add the necessary code, like instances, constructors and so on.

After you have overridden all of the interfaces you made, make sure they all have an instance in the class that implements PlatformWrapper (and that there is a getter). Finally, you change the constructor in the main class. The main class is the class you reference from the launcher. It either extends ApplicationAdapter, implements ApplicationListener or extends Game. Edit the **constructor** and add PlatformWrapper as an argument. Inside the platform wrapper you have some utilities (if you added any) in addition to all the other wrappers (sql, achievements, or any else you added).

Now, if everything is correctly set up, you can make a call to the PlatformWrapper and get any of the cross-platform interfaces. Call any method and (assuming the executed code is correct) you will see on any platform, it does what it is supposed to do with platform specific code

AssetManager

The AssetManager is a class that helps you manage your assets.

First off, you need to create an instance:

```
AssetManager am = new AssetManager();
```

After this is initialized, and before you render anything, you want to get the resources:

```
am.load("badlogic.jpg", Texture.class);//Texture.class is the class this asset is of. If it is
```

```
a
//sound asset, it doesn't go under Texture. if it is a 3D model, it doesn't go under
Texture.class
//Which class added depends on the asset you load

//... other loading here ...//

//when finished, call finishLoading:
am.finishLoading();
```

Now, whereever you feel like getting `badlogic.jpg`:

```
Texture texture = am.get("badlogic.jpg");
//Ready to render! The rendering itself is in the normal way
```

Using AssetManager allows you to load them once into the memory of the AssetManager, and then get them as many times as you want.

Why use AssetManager? (from the wiki):

AssetManager (code) helps you load and manage your assets. It is the recommended way to load your assets, due to the following nice behaviors:

- Loading of most resources is done asynchronously, so you can display a reactive loading screen while things load
- Assets are reference counted. If two assets A and B both depend on another asset C, C won't be disposed until A and B have been disposed. This also means that if you load an asset multiple times, it will actually be shared and only take up memory once!
- A single place to store all your assets.
- Allows to transparently implement things like caches (see FileHandleResolver below)

The 2D scene graph

When you get started with Java or Android, you quickly learn that (0,0) is in the top-left corner. In LibGDX, however, (0,0) is by default in the bottom left corner.

Using an Orthographic camera, you can get (0, 0) to be in the top-left corner. Though by default, (0, 0) is in the bottom-left corner. This is something that is important to know, as it also changes which corner of the textures have the X and Y coordinates.

Read Getting started with libgdx online: https://riptutorial.com/libgdx/topic/1368/getting-started-with-libgdx

Chapter 2: Ashley Entity System

Remarks

Ashley Entity System is an Entity System library that's managed under the LibGDX organization and is and well-suited for game development. It depends on LibGDX utility classes, but can be used with other Java game frameworks not based on LibGDX with some work.

Entity systems provide a different way to manage data and functionality towards large sets of objects without having to make the object classes rich with inheritance.

Utilizing Ashley might be a helpful approach for those looking for an object modeling approach like Unity provides, but with the scope of a framework instead of game engine.

Examples

Creating a Component

Components are simply instances that implement the Ashley component class.

```
import com.badlogic.ashley.core.Component;
import com.badlogic.ashley.core.ComponentMapper;

public class Position implements Component {
    public static final ComponentMapper<Position> Map =
                                ComponentMapper.getFor(Position.class);

    public float x = 0f,
                 y = 0f;
}
```

Component maps provide a fast way to access components on entities. Two common ways to manage your component maps is to either keep a static instance within your component's class, or to have a class/enum that contains all the mappers for all your components.

There's no need to declare a mapper for a component type more than once in your application.

Creating an Entity System

Entity systems are how you perform functional operations on sets of entities. Components should typically not have logic associated with them that involves awareness of data or state of other component instances, as that is the job of an entity system. An entity system can not be registered to more than one engine at a time.

Entity systems should not perform more than one type of function. A MovementSystem should only handle positioning and the like, while something like a RenderSystem should handle the drawing of entities.

```
import com.badlogic.ashley.core.Entity;
import com.badlogic.ashley.core.EntitySystem;
import com.badlogic.ashley.core.Family;

public class MovementSystem extends EntitySystem {
    //the type of components necessary for entities to have to be operated on
    private static final Family FAMILY = Family.all(Position.class).get();

    public MovementSystem () {
        super();
    }

    /**
     * The update method called every tick.
     * @param deltaTime The time passed since last frame in seconds.
     */
    public void update (float deltaTime) {
        for (Entity e : this.getEngine().getEntitiesFor(FAMILY)) {
            Position pos = Position.Map.get(e);

            // do entity movement logic on component
            ...
        }
    }
```

Sometimes it's helpful to extend your EntitySystems with additional functionality like that found in EntityListeners so you keep track of only the types of entities you wish to operate on, instead of iterating over all entities in the engine every cycle. EntityListeners are triggered whenever an entity is added to the same engine that the system is registered to.

```
import com.badlogic.ashley.core.EntityListener;
import com.badlogic.gdx.utils.Array;

public class MovementSystem extends EntitySystem implements EntityListener {
    Array<Entity> moveables = new Array<>();
    ...

    @Override
    public void entityAdded(Entity entity) {
        if (FAMILY.matches(entity)) {
            moveables.add(entity);
        }
    }

    @Override
    public void entityRemoved(Entity entity) {
        if (FAMILY.matches(entity)) {
            moveables.removeValue(entity, true);
        }
    }

    public void update (float deltaTime) {
        for (Entity e : this.moveables) {
            Position pos = Position.Map.get(e);

            // do entity movement logic on component
            ...
        }
    }
```

```
}
```

Keeping track of a subset of entities that the system processes at all times can also be optimal, as you can remove a particular entity from that system's processing without having to remove the associating component or removing them from the engine as a whole if so desired.

Creating a Sorted Entity System

> A simple `EntitySystem` that processes each entity of a given family in the order specified by a `comparator` and calls `processEntity()` for each entity every time the `EntitySystem` is updated. This is really just a convenience class as rendering systems tend to iterate over a list of entities in a sorted manner. Adding entities will cause the entity list to be resorted. Call `forceSort()` if you changed your sorting criteria. For more info, please see SortedIteratingSystem

In the below code example, the best usage for this the rendering of your sprites in a sorted order by zindex.

```
public class SpriteComponent implements Component {
     public TextureRegion region;
     public int z = 0;
}

public class Mapper {
     public static ComponentMapper<SpriteComponent> sprite =
ComponentMapper.getFor(SpriteComponent.class);
}

public class RenderingSystem extends SortedIteratingSystem {

    public RenderingSystem () {
         super(Familly.all(SpriteComponent.class).get(), new ZComparator())
    }

    public void processEntity(Entity entity, float deltaTime) {
         if(checkZIndexHasChangeValue()) {
              forceSort();
         }
    }

    private static class ZComparator implements Comparator<Entity> {
        @Override
        public int compare(Entity entityA, Entity entityB) {
            return (int)Math.signum(Mapper.sprite.get(entityA).z -
Mapper.sprite.get(entityB).z);
        }
    }

}
```

Creating an Interval Iterating System

> A simple `EntitySystem` that processes a Family of entities not once per frame, but after a given interval. Entity processing logic should be placed in `processEntity(Entity)`. For

more info, please see IntervalIteratingSystem

In the below code example, the best usage for this is the physics world step.

```
public class Constants {
     public final static float TIME_STEP = 1 / 60.0f; // 60 fps
     public final static int VELOCITY_ITERATIONS = 6;
     public final static int POSITION_ITERATIONS = 2;
}

public class PhysicsSystem extends IntervalIteratingSystem {
    public PhysicsSystem () {
         super(Family.all(PhysicsComponent.class), Constants.TIME_STEP);
    }

    @Override
    protected void processEntity(Entity entity) {
         // process the physics component here with an interval of 60fps
    }

    @Override
    protected void updateInterval() {
        WorldManager.world.step(Constants.TIME_STEP, Constants.VELOCITY_ITERATIONS,
Constants.POSITION_ITERATIONS);
        super.updateInterval();
    }

}
```

Read Ashley Entity System online: https://riptutorial.com/libgdx/topic/2620/ashley-entity-system

Chapter 3: Box2D

Examples

Create Box2D Bodies from Tiled Map

The objects created within a Tiled Map (.tmx), can be simply loaded as bodies into a Box2D world using the Libgdx MapObject class as following:

```
public void buildBuildingsBodies(TiledMap tiledMap, World world, String layer){
    MapObjects objects = tiledMap.getLayers().get(layer).getObjects();
    for (MapObject object: objects) {
        Rectangle rectangle = ((RectangleMapObject)object).getRectangle();

        //create a dynamic within the world body (also can be KinematicBody or StaticBody
        BodyDef bodyDef = new BodyDef();
        bodyDef.type = BodyDef.BodyType.DynamicBody;
        Body body = world.createBody(bodyDef);

        //create a fixture for each body from the shape
        Fixture fixture = body.createFixture(getShapeFromRectangle(rectangle),density);
        fixture.setFriction(0.1F);

        //setting the position of the body's origin. In this case with zero rotation
        body.setTransform(getTransformedCenterForRectangle(rectangle),0);
    }
}
```

The following functions helps to map the Tiled object coordinates to Box2D shape.

```
public static final float TILE_SIZE = 16;
//Also you can get tile width with:
Float.valueOf(tiledMap.getProperties().get("tilewidth",Integer.class));

public static Shape getShapeFromRectangle(Rectangle rectangle){
    PolygonShape polygonShape = new PolygonShape();
    polygonShape.setAsBox(rectangle.width*0.5F/ TILE_SIZE,rectangle.height*0.5F/ TILE_SIZE);
    return polygonShape;
}
```

And this functions helps to map the center of a Tiled object to the Libgdx's rectangle shape.

```
public static Vector2 getTransformedCenterForRectangle(Rectangle rectangle){
    Vector2 center = new Vector2();
    rectangle.getCenter(center);
    return center.scl(1/TILE_SIZE);
}
```

So, the first function can be used as following:

```
public static final float GRAVITY = 9.8F;
```

```
public void createBodies(AssetManager assetManager){
    TiledMap tiledMap = assetManager.get("tiledMap.tmx");
    //create a Box2d world will contain the physical entities (bodies)
    World world = new World(new Vector2(0,GRAVITY),true);

    String layerName = "BuildingsLayers";
    buildBuildingsBodies(tiledMap,world,layerName);
}
```

Read Box2D online: https://riptutorial.com/libgdx/topic/5052/box2d

Chapter 4: Life-cycle

Remarks

Create

This method is called once when the Application is started. In this method resources should be loaded and variables should be initialized.

Render

The method is called every frame, and is used to display whatever needs to be displayed. It is also used to update any variables/classes that may need to be updated, such as a camera.

Dispose

This method is called when the application is destroyed, and is used to free any resources, for examples `Texture`s or the `SpriteBatch`. You will know that an object has to be disposed of if it implements the `Disposable` interface.

Pause

This method is called when the application is paused. Usually when the application looses focus.

Resume

This method is called when the application should be resumed. Usually when the application regains focus.

Resize

This method is called when the application is resized. This method is normally used to resize a viewport.

Examples

Main Game file

```
class Tutorial extends Game {

    public ScreenNumberOne screenNumberOne;

    public void create(){
        screenNumberOne = new ScreenNumberOne(this);

        this.setScreen(screenNumberOne);
    }

    public void render() {
        super.render();
    }
}
```

That is the basic file to allow you to make multiple screens. Notice that it extends `Game`.

Read Life-cycle online: https://riptutorial.com/libgdx/topic/5053/life-cycle

Chapter 5: Moving actors on path with constant speed

Examples

Simple movement between two locations

For this the best solution is using `actions`. To add a new action to an actors in `Scene2D` just call:

```
Action action = Actions.moveTo(x,y,duration);
actorObject.addAction(action);
```

Where x and y is the target location and duration is the speed of this movement in seconds(`float`).

If you want to stop this action(and the actor by it) you can do it by calling:

```
actorObject.removeAction(action);
```

or you can remove all actions by calling:

```
actorObject.clearActions();
```

This will immediately stop the running of the action(s).

The moveTo action manipulates the x and y property of the actor, so when you draw the actor to the screen always use getX() and getY() to draw textures. Just like in the following example:

```
public class MovingActor extends Actor {

    private Action runningAction;
    private float speed = 2f;

    public void moveTo(Vector2 location) {
       runningAction = Actions.moveTo(location.x, location.y, speed);
       this.addAction(runningAction);
    }

    public void stopAction() {
       this.removeAction(runningAction);
    }

    public void stopAllActions() {
       this.clearActions();
    }

    @Override
    public void draw(Batch batch, float parentAlpha){
        batch.draw(someTexture, getX(), getY());
    }
}
```

Read Moving actors on path with constant speed online:
https://riptutorial.com/libgdx/topic/6384/moving-actors-on-path-with-constant-speed

Chapter 6: Supporting Multiple Resolutions

Examples

Viewports

To support multiple resolutions and aspect ratios Libgdx uses the so called `Viewports`.
There are a few types of `Viewports` which use different strategies to handle multiple resolutions and aspect ratios.
A `Viewport` uses a `Camera` under the hood and manages it's `viewportHeight` and `viewportWidth`. You can optionally give the `Viewport` a `Camera` in it's constructor, otherwise it will use an `OrthographicCamera` by default:

```
private Viewport viewport;
private Camera camera;

public void create() {
   camera = new PerspectiveCamera();
   viewport = new FitViewport(8f, 4.8f, camera);
}
```

Additionally, you must specify the `worldWidth` and `worldHeight` to the viewport's constructor. This space will represent the virtual coordinate system that you will use for specifying objects' position to be drawn and sizes. The viewport transformation, that can be applied to a SpriteBatch, for example, will automatically take care of transforming the logical coordinates to actual screen coordinates, with a way that conforms to the actual type of viewport you are using. For example, when using an orthographic projection (`OrthographicCamera`, which is the default): if your world size is 12.8 by 7.8 meters and your device screen is 2560x1560 pixels, then your world will be dynamically projected with 200 pixels per meter.

When the `Viewport`s size changes (for example if the Screen Orientation of the Smartphone is changing), you need to inform the `Viewport` about that change. it will then automatically update the `Camera`s `viewportHeight` and `viewportWidth`:

```
public void resize(int width, int height) {
    viewport.update(width, height);
}
```

The `Viewport`s also manage the `OpenGL Viewport` and modify the drawable area as needed.

The `Viewport`s also take care about converting screen-coordinates to game-coordinates, which is needed especially for picking:

```
private Vector2 worldPos = new Vector2();
public boolean touchDown (int x, int y, int pointer, int button) {
    worldPos.set(x, y);
    viewport.unproject(worldPos);
    processPicking(worldPos);
```

```
}
```

Builtin Viewports

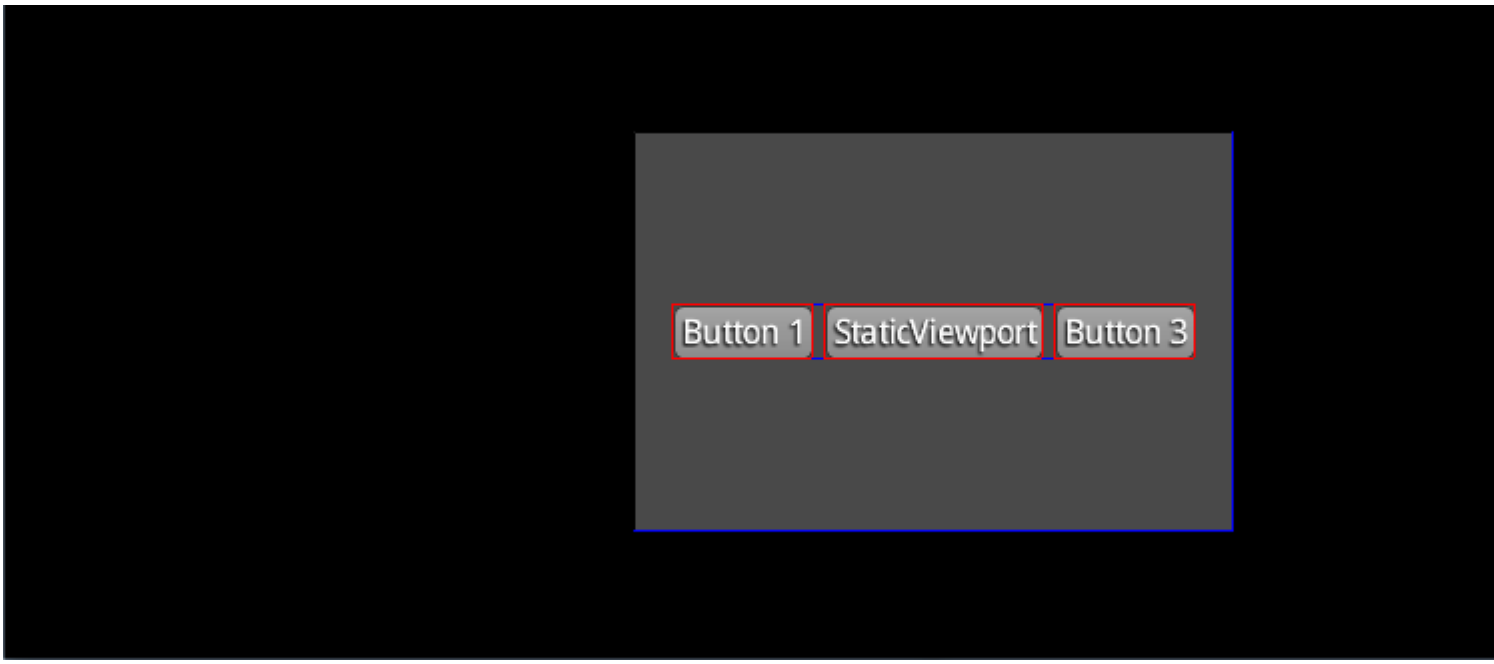
There are a couple of builtin viewports, that each do different things. Here is a list of the names, and their descriptions:

Viewport Name	Description
StretchViewport	Will stretch the screen. No black bars, but aspect ratio might be off.
FitViewport	Will maximize it's size based on the aspect ratio. Could have black bars.
FillViewport	Quite the same as a FitVieport, but always fills the entire screen.
ScreenViewport	Always fills the entire screen, but does not resize any children.
ExtendedViewport	Keeps the world aspect ratio without black bars by extending the world in one direction
CustomViewport	A custom programmed Viewport. Might have black bars, and might keep aspect ratio.

Custom Viewports

You can make your very own custom viewport. It may have black bars, and it may or may not keep it's aspect ratio, depending on how you program it. A custom viewport would look something like this:
`public class Tutorial extends Viewport`. You would need to override `update(width, height)`, but that's it.

Another attempt would be to extend the generic `ScalingViewport`, and supplying another Scaling which is not already supplied. Suppose that you set it to `Scaling.none`. That would look something like this:



If you want some reference, you can find the builtin viewport classes right over here.

StretchViewport

The `StretchViewport` is a `Viewport`type, which supports a virtual screen size.
This allowes you to define a fixed (resolution independent) width and height.
The `StretchViewport`, as the name suggests, stretches the virtual screen, if the virtual aspect ratio does not match the real aspect ratio. The OpenGL Viewport won't be modified and there won't appear black bars.

Usage:

```
private Viewport viewport;
private Camera camera;

public void create() {
    camera = new OrthographicCamera();
    viewport = new StretchViewport(80, 48, camera);
}

public void resize(int width, int height) {
    viewport.update(width, height);
}
```

FitViewport

FitViewports are viewports that always maintain the aspect ratio. It does this by creating black bars on the edges where there is space left. This is one of the most commonly used viewports.

Usage:

```
private Viewport viewport;
private Camera camera;
```

```
public void create() {
    camera = new OrthographicCamera();
    viewport = new FitViewport(80, 48, camera);
}

public void resize(int width, int height) {
    viewport.update(width, height);
}
```

Read Supporting Multiple Resolutions online: https://riptutorial.com/libgdx/topic/4219/supporting-multiple-resolutions

Credits

S. No	Chapters	Contributors
1	Getting started with libgdx	Aryan, Community, Deniz Yılmaz, gaRos, Jesse Lawson, mttprvst13, Winter, Wyatt, Zoe
2	Ashley Entity System	nhydock, ronscript
3	Box2D	Alex Sifuentes, Alvaro Hernandez
4	Life-cycle	Benedikt S. Vogler, mttprvst13, nhydock, Skrelp
5	Moving actors on path with constant speed	gaRos
6	Supporting Multiple Resolutions	Eames, Lake, mttprvst13, Springrbua, Xoppa