CODING

Look out, Duke! How to build a Java game with JavaFX and the FXGL library

You'll be amazed at how little code is needed to create a 2D arcade game.

by Frank Delporte

June 25, 2021

Look out, Duke! Don't run into a cloud!

FXGL, the JavaFX Game Development Framework, is exactly what you need to extend your Java skills to become a game developer. FXGL is a dependency you add to your Java and JavaFX project; it doesn't need any additional installation or setup. It works out of the box on all platforms.

Thanks to the simple and clean FXGL API, you can build 2D games with minimal code and deliver them as a single executable `.jar` file or native image. Almas Baimagambetov, senior lecturer in game development at the University of Brighton, is the creator of FXGL, and the project is fully open source and has a clear description about how you can contribute to it.

You can see basic examples of using the FXGL library in the main GitHub project. More-complex games are provided in a separate project, FXGLGames. You can also use FXGL to build business applications with complex UI controls; there are also 3D interface controls in the library, but those are still experimental.

In this article, you will see all the code needed to build a fun game where Duke shoots at a circle while moving around—while avoiding floating cloud servers. Before proceeding, you might want to watch a one-minute video that shows how the game is played.

Oracle Java Magazine - FXGL

from **Frank Delporte**

Oracle Java Magazine—FXGL from Frank Delporte on Vimeo

In a follow-up article, you'll see how to control the game with a joystick on a Raspberry Pi device. For now, though, the goal is to create the game itself.

Source code and image assets

The finished project, called JavaMagazineFXGL, is available on GitHub. This is a Maven project you can build with `mvn package`. In the `pom.xml` file there is only one dependency, because FXGL itself depends on JavaFX.

```
<dependencies>
        <dependency>
            <groupId>com.github.almasb</grou
            <artifactId>fxgl</artifactId>
            <version>${fxgl.version}</versio
        </dependency>
    </dependencies>
```

The full code for the game consists of only a few classes, which include the `EntityFactory`, `CloudComponent`, and `PlayerComponent`, as well as the `Main` class and FXGL application overrides.

By default, FXGL loads images from the `src > main > resources > assets > textures` directory, which has the few images, such as Duke, that are used in the game. (See **Figure 1**, **Figure 2**, and **Figure 3**.)

Figure 1. The Duke character: duke.png

Figure 2. The cloud obstacles: cloud-network.png

Figure 3. Duke's bullet: sprite_bullet.png

The EntityFactory

All the game objects in an FXGL application are of type `Entity` and need to be defined in an `EntityFactory`. In this sample game, they are in a class called `GameFactory.java`.

```
public enum EntityType {
    BACKGROUND, CENTER, DUKE, CLOUD, BULLET
}
```

By defining an enum with the types, it becomes easier to reference them later, such as in collision detection. For each entity type, a `@Spawns` annotated method defines the layout and behavior of the object. Both the background and centered circle have a fixed size, which is defined in `SpawnData`. FXGL offers many components to control the entities, and in this case, the application uses the `IrremovableComponent` because these entities should never be removed.

```
@Spawns("background")
public Entity spawnBackground(SpawnData data
    return entityBuilder(data)
            .type(EntityType.BACKGROUND)
            .view(new Rectangle(data.<Intege
                data.<Integer>get("height"),
            .with(new IrremovableComponent()
            .zIndex(-100)
            .build();
}

@Spawns("center")
public Entity spawnCenter(SpawnData data) {
    return entityBuilder(data)
            .type(EntityType.CENTER)
            .collidable()
            .viewWithBBox(new Circle(data.<I
                 data.<Integer>get("y"), data
            .with(new IrremovableComponent()
            .zIndex(-99)
            .build();
}
```

For the Duke character and clouds, I gave the images a `viewWithBBox` for collision detection. I used the FXGL

`AutoRotationComponent` and custom `PlayerComponent` and `CloudComponent` because the game needs to add specific controls to these entities.

```
@Spawns("duke")
public Entity newDuke(SpawnData data) {
    return entityBuilder(data)
            .type(EntityType.DUKE)
            .viewWithBBox(texture("duke.png"
            .collidable()
            .with((new AutoRotationComponent
            .with(new PlayerComponent())
            .build();
}

@Spawns("cloud")
public Entity newCloud(SpawnData data) {
    return entityBuilder(data)
            .type(EntityType.CLOUD)
            .viewWithBBox(texture("cloud-netv
            .with((new AutoRotationComponent
            .with(new CloudComponent())
            .collidable()
            .build();
}
```

The bullet doesn't need a custom component, because FXGL's `ProjectileComponent` and `OffscreenCleanComponent` have all the needed functionality.

```
@Spawns("bullet")
public Entity newBullet(SpawnData data) {
    return entityBuilder(data)
            .type(EntityType.BULLET)
            .viewWithBBox(texture("sprite_bul
            .collidable()
            .with(new ProjectileComponent(dat
            .build();
}
```

The CloudComponent

The `CloudComponent` class illustrates the flexibility that's provided by an FXGL component. `OnUpdate` is called at each engine tick, allowing the game to fully control the behavior of the entity to which the component is attached.

In this case, the game moves the cloud in the direction that was randomly calculated upon initialization of the component. The game checks whether the cloud hits the border of the game and removes it from the game world if it does.

```
public class CloudComponent extends Component
    private final Point2D direction = new Poi
```

```
            FXGLMath.random(-1D, 1D),
            FXGLMath.random(-1D, 1D)
        );

        @Override
        public void onUpdate(double tpf) {
            entity.translate(direction.multiply(
            checkForBounds();
        }

        private void checkForBounds() {
            if (entity.getX() < 0) {
                remove();
            }
            if (entity.getX() >= getAppWidth()) 
                remove();
            }
            if (entity.getY() < 0) {
                remove();
            }
            if (entity.getY() >= getAppHeight())
                remove();
            }
        }

        public void remove() {
            entity.removeFromWorld();
        }
    }
```

The PlayerComponent

The `PlayerComponent` uses a `Point2D` object to define Duke's direction. Initially, Duke moves to the bottom right. The `up`, `down`, `left`, and `right` methods change the direction gradually defined by the `ROTATION_CHANGE` value. As with the `CloudComponent`, there is a check for the borders of the screen, but in this case, the code calls the `die` method when Duke hits the border.

The `die` method decreases the "number of lives" value and resets the direction and position to get back to the starting position. When the player doesn't have any lives left, the game shows a "Game Over" message box.

The `shoot` method spawns a new bullet at Duke's current position, giving the bullet the same direction that Duke is traveling in.

```
    public class PlayerComponent extends Componer
        private static final double ROTATION_CHAI
        private Point2D direction = new Point2D(1

        @Override
        public void onUpdate(double tpf) {
            entity.translate(direction.multiply(1
            checkForBounds();
        }
```

```
    private void checkForBounds() {
        if (entity.getX() < 0) {
            die();
        }
        if (entity.getX() >= getAppWidth()) {
            die();
        }
        if (entity.getY() < 0) {
            die();
        }
        if (entity.getY() >= getAppHeight()) {
            die();
        }
    }

    public void shoot() {
        spawn("bullet", new SpawnData(
                getEntity().getPosition().get
                getEntity().getPosition().get
                .put("direction", direction)
    }

    public void die() {
        inc("lives", -1);

        if (geti("lives") <= 0) {
            getDialogService().showMessageBo
                    () -> getGameController(
            return;
        }

        entity.setPosition(0, 0);
        direction = new Point2D(1, 1);
        right();
    }

    public void up() {
        if (direction.getY() > -1) {
            direction = new Point2D(directio
        }
    }

    public void down() {
        if (direction.getY() < 1) {
            direction = new Point2D(directio
        }
    }

    public void left() {
        if (direction.getX() > -1) {
            direction = new Point2D(directio
        }
    }

    public void right() {
        if (direction.getX() < 1) {
            direction = new Point2D(directio
        }
    }
}
```

Main class and FXGL application overrides

With the three previous classes, everything is prepared to create the game. All that's left is the `main` class. The `main` class combines everything and extends from FXGL `GameApplication`, which provides multiple override methods to configure your game. These methods are called during initialization in the following order:

1. Instance fields of your subclass of `GameApplication`
2. `InitSettings()`, which initialize the app settings
3. Services configuration, after which you can safely call any `FXGL.*` methods
4. The following, which are executed on a JavaFX UI thread
    - 1. `initInput()`, which initializes inputs, such as key presses and mouse buttons
    - 2. `onPreInit()`, which is called once per application lifetime, before `initGame()`
5. The following, which are not executed on the JavaFX UI thread
    - 1. `initGameVars()`, which can be overridden to provide global variables
    - 2. `initGame()`, which initializes each game's objects
    - 3. `initPhysics()`, which initializes collision handlers and physics properties
    - 4. `initUI()`, which starts the main game loop execution on the JavaFX UI thread

Because the game should run full-screen with the right dimensions, the application reads these values in the `main` method. The game also uses the `GameFactory`, created earlier, and requires an `Entity` for the player to contain the Duke entity.

```
private final GameFactory gameFactory = new (
private Entity player;

private static int screenWidth;
private static int screenHeight;

public static void main(String[] args) {
    Dimension screenSize = Toolkit.getDefaul
    screenWidth = (int) screenSize.getWidth(
    screenHeight = (int) screenSize.getHeigh
    launch(args);
}
```

FXGL overrides. `GameSettings` contains a long list of methods to configure your game. This example uses only those necessary to make the game full-screen and give it a title.

```
@Override
    protected void initSettings(GameSettings
```

```
        settings.setHeight(screenHeight);
        settings.setWidth(screenWidth);
        settings.setFullScreenAllowed(true);
        settings.setFullScreenFromStart(true
        settings.setTitle("Oracle Java Magaz
    }
```

Override `initInput` configures the input events that control the game. Because Duke should rotate while the player presses one of the arrow keys, the `onKey` method is used. Firing a bullet should happen only once each time the space bar is pressed, so this uses the `onKeyDown` method. `onPreInit` is not used in this example application.

```
@Override
    protected void initInput() {
        onKey(KeyCode.LEFT, "left", () -> thi
        onKey(KeyCode.RIGHT, "right", () -> t
        onKey(KeyCode.UP, "up", () -> this.pl
        onKey(KeyCode.DOWN, "down", () -> thi
        onKeyDown(KeyCode.SPACE, "Bullet", (
    }
```

The game needs a value for the number of lives that are left and a value for the score. Both are defined in `initGameVars` and used in `initUI`.

```
@Override
    protected void initGameVars(Map<String, O
        vars.put("score", 0);
        vars.put("lives", 5);
    }
```

Let's add some stuff to the game world! First, add the entity factory that generates the game entities. Then, it's time to add the spawns: first a full-screen background, then a circle in the middle of the screen and, finally, Duke as the player entity.

```
@Override
    protected void initGame() {
        getGameWorld().addEntityFactory(this

        // Background color
        spawn("background", new SpawnData(0,
                .put("height", getAppHeight(

        // Circle in the middle of the screen
        int circleRadius = 80;
        spawn("center", new SpawnData(
           (getAppWidth() / 2) - (circleRadiu
           (getAppHeight() / 2) - (circleRadi
                .put("x", (circleRadius / 2)
                .put("y", (circleRadius / 2)
                .put("radius", circleRadius)
```

```
        // Add the player
        this.player = spawn("duke", 0, 0);
    }
```

Next, define the collision handlers as follows:

- Whenever Duke hits a cloud or the centered circle, the player loses a life.
- Whenever a bullet hits a cloud, the player's score increases; both entities must be removed from the game.

Use lambdas to define the type of entities that need to be handled and the actions to be taken.

```
@Override
    protected void initPhysics() {
        onCollisionBegin(EntityType.DUKE, Ent
            this.player.getComponent(PlayerC

        onCollisionBegin(EntityType.DUKE, Ent
            this.player.getComponent(PlayerC

        onCollisionBegin(EntityType.BULLET, E
            inc("score", 1);
            bullet.removeFromWorld();
            cloud.removeFromWorld();
        });
    }
```

The player needs to see the score and lives-remaining variables defined in `initGameVars`, and this is done with `initUI`. By binding the `textProperty` to these values, the onscreen data will always be up to date.

```
@Override
    protected void initUI() {
        Text scoreLabel = getUIFactoryService
        Text scoreValue = getUIFactoryService
        Text livesLabel = getUIFactoryService
        Text livesValue = getUIFactoryService

        scoreLabel.setTranslateX(20);
        scoreLabel.setTranslateY(20);

        scoreValue.setTranslateX(90);
        scoreValue.setTranslateY(20);

        livesLabel.setTranslateX(getAppWidth
        livesLabel.setTranslateY(20);

        livesValue.setTranslateX(getAppWidth
        livesValue.setTranslateY(20);

        scoreValue.textProperty().bind(getWor
        livesValue.textProperty().bind(getWor
```

```
            getGameScene().addUINodes(scoreLabel
        }
```

Those were all the overrides called during the initialization needed for the game. There is one final initialization, but this is called every frame when the game is in play state: The game shall always have 10 clouds on the screen. By adding one cloud per frame, if needed, the clouds will appear one by one at the start of the game.

```
@Override
    protected void onUpdate(double tpf) {
        if (getGameWorld().getEntitiesByType
            spawn("cloud", getAppWidth() / 2
        }
    }
```

Conclusion

That's it! That's all the code you need to have a fully functional game in JavaFX. Thanks to the clever helper methods provided by FXGL, such as `FXGLMath.random`, you don't need to write a lot of code or include extra dependencies to achieve very nice results. This example is a nice starting point if you are new to game development. Have fun with the source code, use it as an inspiration, and please share what you created.

A follow-up article will take this game to the Raspberry Pi single-board computer and show how to add physical buttons and a joystick to create a true arcade-like experience.

Dig deeper

- Getting started with JavaFX on Raspberry Pi
- Creating 2D action games with the Game API
- Games, robots, and Java, oh my!
- The FXGL game library
- Getting started with Java on Raspberry Pi (ebook)

Frank Delporte

Frank Delporte is the author of *Getting Started with Java on Raspberry Pi* and is technical product lead at Televic Rail in Izegem, Belgium. He is also the lead coach for CoderDojo Belgium.

Share this Page