libGDX

https://github.com/libgdx/libgdx/wiki/

Rafael Vieira Coelho

rafael.coelho@farroupilha.ifrs.edu.br

Tópicos

- The Application Framework
- **A Simple Game**
- File Handling
- Networking
- Preferences
- Input Handling
- Memory Management
- Audio
- Graphics

https://github.com/libgdx/libgdx/wiki/

A Simple Game

- Criaremos um exemplo simples nos quais abordaremos:

1. Acesso básico a arquivos
2. Limpar a tela
3. Desenhar imagens
4. Usar a câmera
5. Processar entrada de dados
6. Tocar efeitos sonoros

- A organização do projeto é a seguinte:
  - Application name: drop
  - Package name: com.badlogic.drop
  - Game class: Drop

- Como criar o projeto Gradle:
https://github.com/libgdx/libgdx/wiki/Project-Setup-Gradle

A Simple Game: Gameplay

- Objetivo: Capturar as gotas de chuva com um balde.
- Colocar o balde na parte de baixo da tela.
- Gotas de chuva caem randomicamente do topo da tela.
- O jogador pode movimentar o balde horizontalmente através do mouse ou teclado.
- O jogo não tem fim.

Os Assets

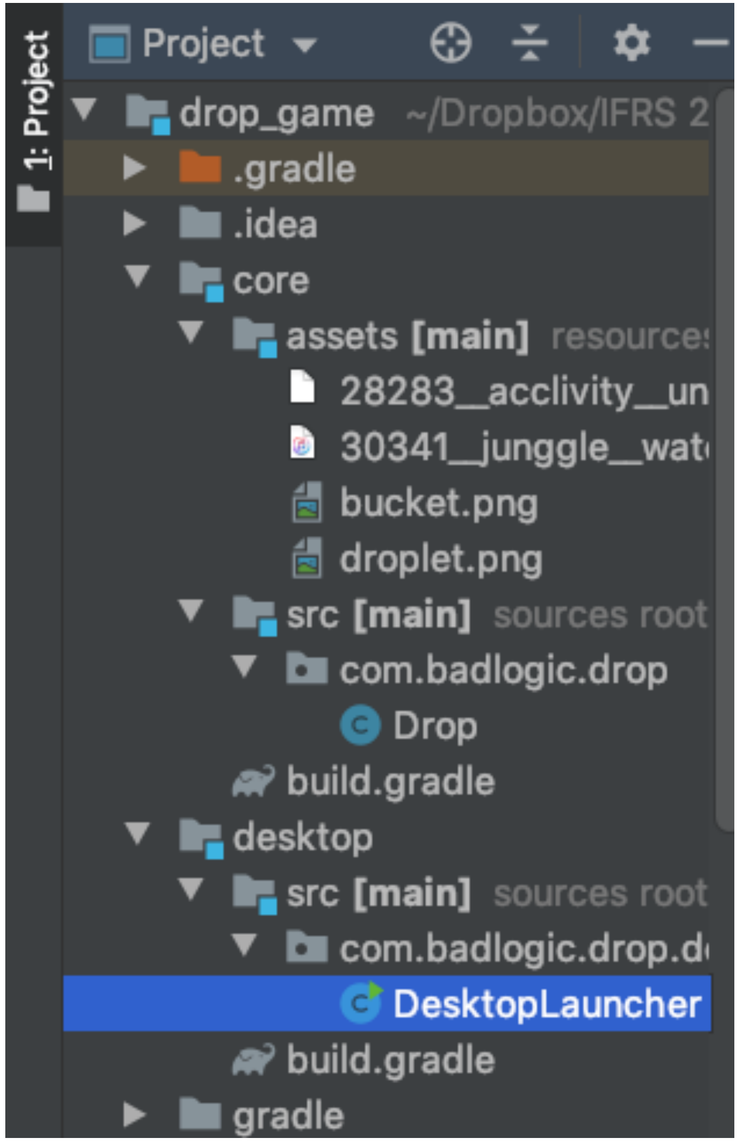
- Inicialmente, precisamos definir a resolução **L x A pixels**.
- É possível ter **assets** de diferentes tamanhos para se adaptar a resolução.
- No drop_game, as imagens do balde e da gota deve ter proporções pequenas (**64 x 64 pixels**).
- Podemos obter os assets (devem ser colocados na pasta **/core/assets/**) dos links abaixo:

・water drop sound by junggle:
http://www.freesound.org/people/junggle/sounds/30341/

・rain by acclivity:
http://www.freesound.org/people/acclivity/sounds/28283/

・droplet sprite by mvdv:
https://www.box.com/s/peqrdkwjl6guhpm48nit

・bucket sprite by mvdv:
https://www.box.com/s/605bvdlwuqubtutbyf4x



Configuração da Janela

- Precisamos definir o título (**title**), largura (**width**) e altura (**height**).

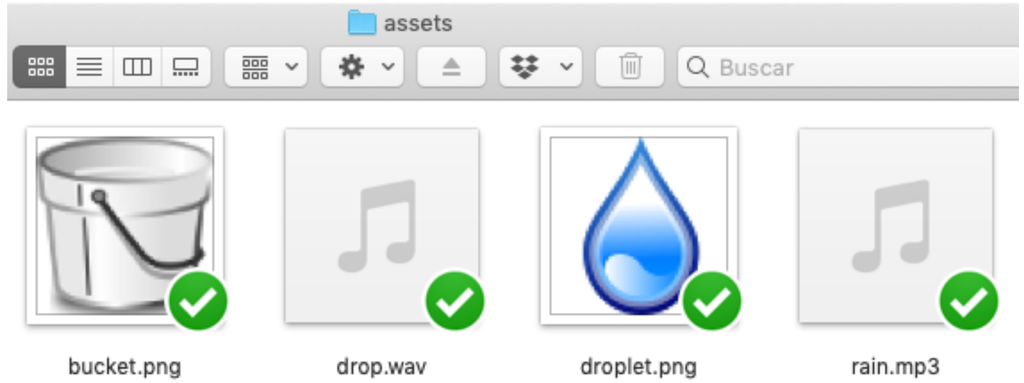
```
package com.badlogic.drop.desktop;

import com.badlogic.gdx.backends.lwjgl.LwjglApplication;
import com.badlogic.gdx.backends.lwjgl.LwjglApplicationConfiguration;
import com.badlogic.drop.Drop;

public class DesktopLauncher {
   public static void main (String[] arg) {
      LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
      config.title = "Drop";
      config.width = 800;
      config.height = 480;
      new LwjglApplication(new Drop(), config);
   }
}
```

Carregando os Assets

- Precisamos criar os atributos para as texturas e os sons.
- Precisamos importar as classes.
- E, por fim, criar os objetos correspondentes.
- Usamos dois módulos: **files** e **audio**.
- Para carregar arquivos, usa-se o método **internal**.
- Para carregar um efeito sonoro, o método **newSound**.
- Para carregar uma música, o método **newMusic**.




```java
package com.badlogic.drop;

import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.audio.Music;
import com.badlogic.gdx.audio.Sound;
import com.badlogic.gdx.graphics.Texture;

public class Drop extends ApplicationAdapter {
    private Texture dropImage;
    private Texture bucketImage;
    private Sound dropSound;
    private Music rainMusic;

    @Override
    public void create() {
        // load the images for the droplet and the bucket, 64x64 pixels each
        dropImage = new Texture(Gdx.files.internal("droplet.png"));
        bucketImage = new Texture(Gdx.files.internal("bucket.png"));

        // load the drop sound effect and the rain background "music"
        dropSound = Gdx.audio.newSound(Gdx.files.internal("drop.wav"));
        rainMusic = Gdx.audio.newMusic(Gdx.files.internal("rain.mp3"));

        // start the playback of the background music immediately
        rainMusic.setLooping(true);
        rainMusic.play();

        ... more to come ...
    }
```

Criando a Camera e a SpriteBatch

- A classe **SpriteBatch** é uma classe especial que é usada para desenhar imagens 2D.
- Precisamos adicionar dois novos atributos:

```
private OrthographicCamera camera;
private SpriteBatch batch;
```

- E no método create(), devemos criar os objetos:

```
camera = new OrthographicCamera();
camera.setToOrtho(false, 800, 480);
 batch = new SpriteBatch();
```

Criando o Balde e Renderizando-o

- Primeiramente, criamos o atributo do balde.

```
private Rectangle bucket;
```

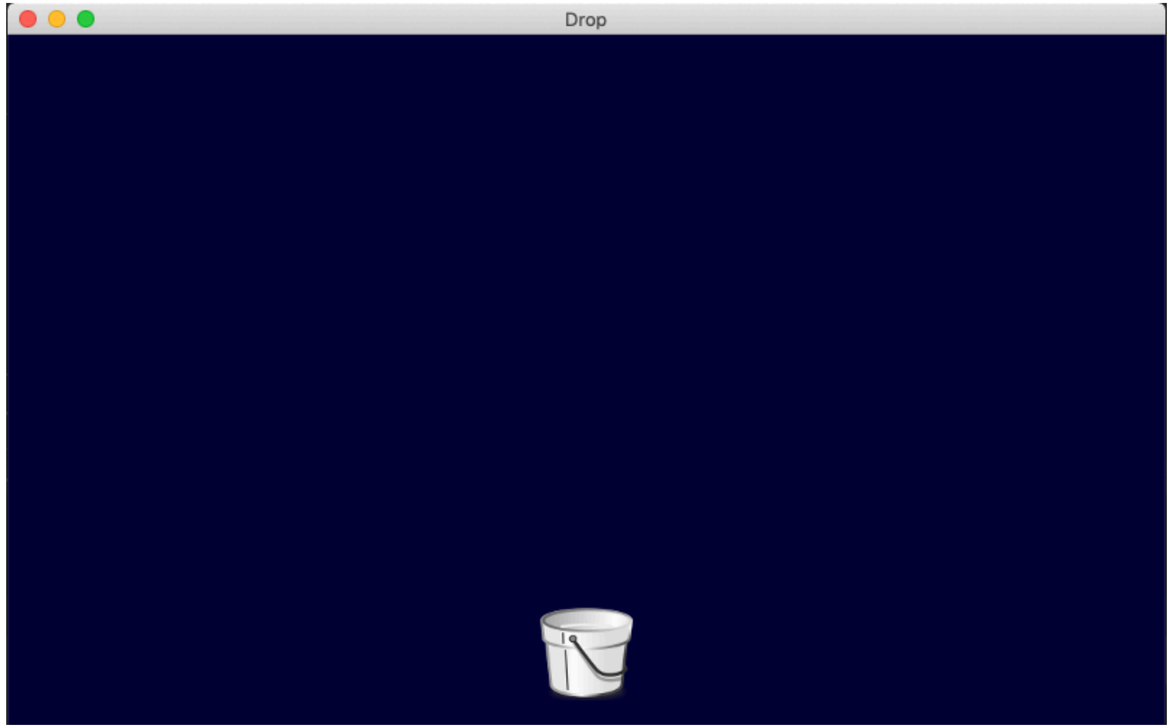
- E instanciamos no método create().

```
bucket = new Rectangle();
bucket.x = 800 / 2 - 64 / 2;
bucket.y = 20;
bucket.width = 64;
bucket.height = 64;
```

- Por fim, atualizamos a camera e desenhamos o balde no método render().

```
@Override
public void render() {
   Gdx.gl.glClearColor(0, 0, 0.2f, 1);
   Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    camera.update();
   batch.setProjectionMatrix(camera.combined);
   batch.begin();
   batch.draw(bucketImage, bucket.x, bucket.y);
   batch.end();
 }
```

- Colocar os comandos de desenho entre os métodos begin() e end() agiliza a renderização do OpenGL.



Movimentando o Balde com o Mouse

- No método **render()**, acessamos o módulo **input** e chamados o método **isTouched()** para verificar se o mouse foi usado.

```
if(Gdx.input.isTouched()) {
    Vector3 touchPos = new Vector3();
    touchPos.set(Gdx.input.getX(), Gdx.input.getY(), 0);
    camera.unproject(touchPos);
    bucket.x = touchPos.x - 64 / 2;
}
```

- Caso seja retornado true, podemos obter a **posição (x, y)** no plano cartesiano da tela através do método **getX()** e **getY()** do módulo input.

- Atualizamos o ponteiro do mouse e atualizamos a posição do balde com base na posição do cursor do mouse.

- O objeto **Vector3** encapsula um vetor 3D (posição x, y e z).

- Precisamos chamar o método **unproject()** a partir do objeto camera para traduzir as coordenadas do mouse para as coordenadas de nossa camera.

Movimentando o Balde com o Teclado

- Precisamos chamar o método **isKeyPressed()** do módulo **input** que retorna um valor lógico, indicando se uma tecla foi pressionada ou não.

```
if(Gdx.input.isKeyPressed(Input.Keys.LEFT))
    bucket.x -= 200 * Gdx.graphics.getDeltaTime();

if(Gdx.input.isKeyPressed(Input.Keys.RIGHT))
    bucket.x += 200 * Gdx.graphics.getDeltaTime();
```

- Precisamos adicionar ou remover **200 pixels na posição x** com base na tecla pressionada (seta esquerda ou seta direita).

- E também precisamos **limitar a posição x** para não sair da tela.

```
if(bucket.x < 0)
    bucket.x = 0;

if(bucket.x > 800 - 64)
    bucket.x = 800 - 64;
```

Adicionando as Gotas de Chuva

- Inicialmente, devemos definir um Array de objetos Rectangle que representam as gotas.

```
private Array<Rectangle> raindrops;
```

- E também precisamos saber o tempo decorrido da criação da última gota.

```
private long lastDropTime;
```

- Criaremos um método que cria uma gota.

```
private void spawnRaindrop() {
   Rectangle raindrop = new Rectangle();
   raindrop.x = MathUtils.random(0, 800-64);
   raindrop.y = 480;
   raindrop.width = 64;
   raindrop.height = 64;
   raindrops.add(raindrop);
   lastDropTime = TimeUtils.nanoTime();
}
```

- E criar uma gota e instanciar o vetor de gotas no método create().

```
raindrops = new Array<Rectangle>();
spawnRaindrop();
```

Adicionando as Gotas de Chuva

- No método render(), temos que testar quanto tempo passou da última gota criada.

```
if(TimeUtils.nanoTime() - lastDropTime > 1000000000)
    spawnRaindrop();
```

- E movimenta as gotas (200 pixels por segundo) no método render(). Caso passe da tela, remove-se a gota.

```
for (Iterator<Rectangle> it = raindrops.iterator(); it.hasNext(); ) {
    Rectangle raindrop = it.next();
    raindrop.y -= 200 * Gdx.graphics.getDeltaTime();
    if(raindrop.y + 64 < 0) it.remove();
}
```

- Desenhar as gotas.

```
batch.begin();
batch.draw(bucketImage, bucket.x, bucket.y);
for(Rectangle raindrop: raindrops) {
   batch.draw(dropImage, raindrop.x, raindrop.y);
}
batch.end();
```

- E verificar se houve uma colisão (método **overlaps()**) entre o balde e as gotas para removê-las no laço do método render().

```
if(raindrop.overlaps(bucket)) {
   dropSound.play();
   iter.remove();
}
```

Limpando os Componentes

- Precisamos liberar o espaço que estava sendo usado.

```
@Override
public void dispose() {
    dropImage.dispose();
    bucketImage.dispose();
    dropSound.dispose();
    rainMusic.dispose();
    batch.dispose();
}
```

Como Pausar/Retomar o Jogo

- Primeiramente, criamos uma **enum** para representar os estados possíveis do jogo.

```
public enum State {
    PAUSE,
    RUN,
}
```

- Reimplementar os métodos para atualizar o estado do jogo.

```
@Override
public void pause() {
    this.state = State.PAUSE;
}

@Override
public void resume() {
    this.state = State.RUN;
}
```

- E criar o atributo de estado.

```
private State state;
```

- Criar no método create().

```
state = State.RUN;
```

- E como pode ser visto ao lado, verificar o estado do jogo (iniciados pela **tecla P** e restaurado pela **tecla R**).

- Caso seja **State.RUN**, deve-se fazer o que já fazíamos para atualizar a tela. Mas o desenho estático dos componentes deve ser realizado independente do estado.

```
@Override
public void render() {

    if (Gdx.input.isKeyPressed(Input.Keys.P))
        pause();
    if (Gdx.input.isKeyPressed(Input.Keys.R))
        resume();

    Gdx.gl.glClearColor( red: 0,  green: 0,  blue: 0.2f,  alpha: 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    camera.update();
    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    batch.draw(bucketImage, bucket.x, bucket.y);
    for (Rectangle raindrop : raindrops) {
        batch.draw(dropImage, raindrop.x, raindrop.y);
    }
    batch.end();

switch (state) {
    case RUN:
        //check mouse input
        if (Gdx.input.isTouched()) {...}
        //check keyboard input
        if (Gdx.input.isKeyPressed(Input.Keys.LEFT))
            bucket.x -= 200 * Gdx.graphics.getDeltaTime();
        if (Gdx.input.isKeyPressed(Input.Keys.RIGHT))
            bucket.x += 200 * Gdx.graphics.getDeltaTime();
        //check screen limits
        if (bucket.x < 0)
            bucket.x = 0;
        if (bucket.x > 800 - 64)
            bucket.x = 800 - 64;
        //check time to create another raindrop
        if (TimeUtils.nanoTime() - lastDropTime > 1000000000)
            spawnRaindrop();
        //move raindrops created
        for (Iterator<Rectangle> it = raindrops.iterator(); it.hasNext();
        break;
    case PAUSE:
```

Como Pausar/Retomar o Jogo

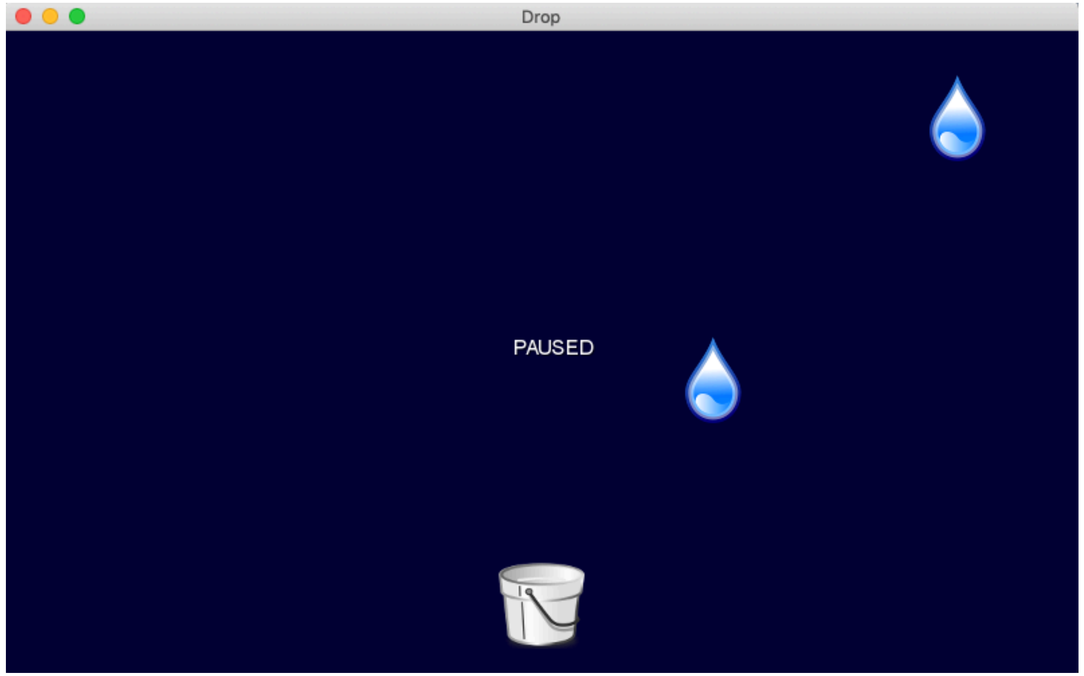
- No momento que está pausado, iremos escrever a palavra '**PAUSE**' na tela.

- Para isto, precisamos de um atributo da classe **BitmapFont** para conseguir escrever na tela.

```
private BitmapFont font;
```

- E no método render() testar o estado da enum State e escrever na tela com o método **draw()**.

```
    case PAUSE:
        batch.begin();
        font.draw(batch, str: "PAUSED", x: 380, y: 250);
        batch.end();
        break;
```



Extendendo o Jogo: contador de gotas

- Inicialmente, podemos adicionar um contador de gotas obtidas pelo balde no canto da tela.

- Precisamos criar um contador como atributo que deve ser inicializado em zero no método **create()**.

```
private int count_raindrops;
```

- E somar 1 nele quando houver uma colisão entre o balde e uma gota no método **render()**.

```
//move raindrops created
for (Iterator<Rectangle> it = raindrops.iterator(); it.hasNext()
    Rectangle raindrop = it.next();
    raindrop.y -= 200 * Gdx.graphics.getDeltaTime();
    //check if it is beyond the screen
    if (raindrop.y + 64 < 0)
        it.remove();
    //check collision between bucket and raindrops
    if (raindrop.overlaps(bucket)) {
        count_raindrops++;
        dropSound.play();
        it.remove();
    }
}
```

Extendendo o Jogo: tela inicial

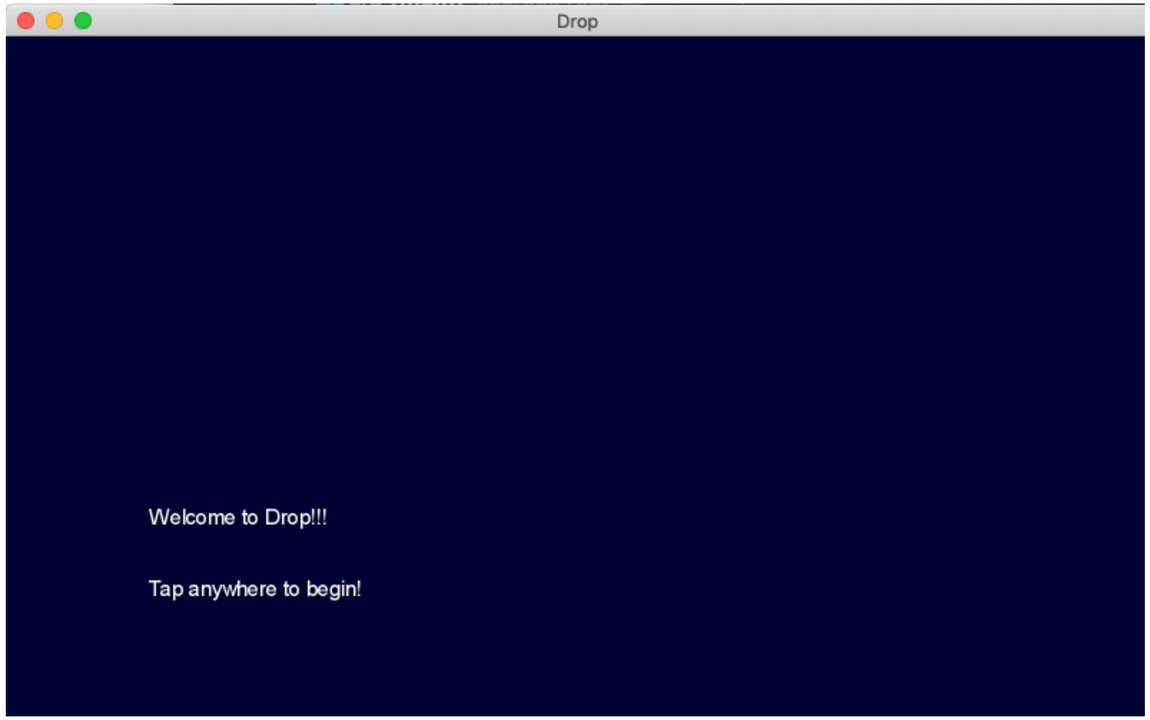
- Teremos 3 classes no pacote core:

- Classe **Drop** extende classe Game: tem os objetos para desenhar (SpriteBatch) e escrever (BitmapFont) na tela. Inicia a tela inicial (instancia objeto MainMenuScreen).

- Classe **MainMenuScreen** implementa a interface Screen: cria a tela inicial e espera por um clique para criar um objeto GameScreen.

- Classe **GameScreen** implementa a interface Screen: tem a lógica do jogo e atualização do mesmo.

- A interface **Screen** é fundamental para qualquer jogo com componentes múltiplos. Ela contém métodos usados por objetos ApplicationListener e tem alguns métodos próprios (ex: **show** e **hide** que são usados quando a tela ganha ou perde foco).



1) Classe Drop

- Cria os objetos **SpriteBatch** e **BitmapFont**.
- Define a tela inicial (objeto **MainMenuScreen**).

```java
package com.badlogic.drop;

import com.badlogic.gdx.Game;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;


public class Drop extends Game {

	public SpriteBatch batch;
	public BitmapFont font;

	public void create() {
		batch = new SpriteBatch();
		//Use LibGDX's default Arial font.
		font = new BitmapFont();
		this.setScreen(new MainMenuScreen(this));
	}

	public void render() {
		super.render(); //important!
	}

	public void dispose() {
		batch.dispose();
		font.dispose();
	}

}
```

2) Classe MainMenuScreen

- Recebe o **objeto Drop** e cria a camera.

- A cada atualização da tela (**método render()**), atualiza-se a tela inicial e espera-se por um clique do mouse para criar a tela principal do jogo (**objeto GameScreen**).

```
package com.badlogic.drop;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;

public class MainMenuScreen implements Screen {

        final Drop game;

        OrthographicCamera camera;

        public MainMenuScreen(final Drop game) {
                this.game = game;

                camera = new OrthographicCamera();
                camera.setToOrtho(false, 800, 480);

        }
```

```
@Override
public void render(float delta) {
        Gdx.gl.glClearColor(0, 0, 0.2f, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        camera.update();
        game.batch.setProjectionMatrix(camera.combined);

        game.batch.begin();
        game.font.draw(game.batch, "Welcome to Drop!!! ", 100, 1
        game.font.draw(game.batch, "Tap anywhere to begin!", 100
        game.batch.end();

        if (Gdx.input.isTouched()) {
                game.setScreen(new GameScreen(game));
                dispose();
        }
}

//Rest of class still omitted...
```

3) Classe GameScreen

- Tem os atributos do jogo.
- Na criação, recebe uma instância de **Drop** e cria instancia os objetos necessários para o jogo (carrega assets, define proporções, etc.).

```
public class GameScreen implements Screen {
        final Drop game;

        Texture dropImage;
        Texture bucketImage;
        Sound dropSound;
        Music rainMusic;
        OrthographicCamera camera;
        Rectangle bucket;
        Array<Rectangle> raindrops;
        long lastDropTime;
        int dropsGathered;

        public GameScreen(final Drop game) {
                this.game = game;

                // load the images for the droplet and the bucket, 64x64
                dropImage = new Texture(Gdx.files.internal("droplet.png"
                bucketImage = new Texture(Gdx.files.internal("bucket.png

                // load the drop sound effect and the rain background "m
                dropSound = Gdx.audio.newSound(Gdx.files.internal("drop.
                rainMusic = Gdx.audio.newMusic(Gdx.files.internal("rain.
                rainMusic.setLooping(true);

                // create the camera and the SpriteBatch
                camera = new OrthographicCamera();
                camera.setToOrtho(false, 800, 480);

                // create a Rectangle to logically represent the bucket
                bucket = new Rectangle();
                bucket.x = 800 / 2 - 64 / 2; // center the bucket horizo
                bucket.y = 20; // bottom left corner of the bucket is 20
                                                // the bottom screen edg
                bucket.width = 64;
                bucket.height = 64;

                // create the raindrops array and spawn the first raindr
                raindrops = new Array<Rectangle>();
                spawnRaindrop();

        }
```

3) Classe GameScreen

- No método render() tem a lógica do jogo para atualizar o mesmo.

```
@Override
public void render(float delta) {
        // clear the screen with a dark blue color. The
        // arguments to glClearColor are the red, green
        // blue and alpha component in the range [0,1]
        // of the color to be used to clear the screen.
        Gdx.gl.glClearColor(0, 0, 0.2f, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        // tell the camera to update its matrices.
        camera.update();

        // tell the SpriteBatch to render in the
        // coordinate system specified by the camera.
        game.batch.setProjectionMatrix(camera.combined);

        // begin a new batch and draw the bucket and
        // all drops
        game.batch.begin();
        game.font.draw(game.batch, "Drops Collected: " + dropsGathered,
        game.batch.draw(bucketImage, bucket.x, bucket.y, bucket.width, b
        for (Rectangle raindrop : raindrops) {
                game.batch.draw(dropImage, raindrop.x, raindrop.y);
        }
        game.batch.end();

        // process user input
        if (Gdx.input.isTouched()) {
                Vector3 touchPos = new Vector3();
                touchPos.set(Gdx.input.getX(), Gdx.input.getY(), 0);
                camera.unproject(touchPos);
                bucket.x = touchPos.x - 64 / 2;
        }
        if (Gdx.input.isKeyPressed(Keys.LEFT))
                bucket.x -= 200 * Gdx.graphics.getDeltaTime();
        if (Gdx.input.isKeyPressed(Keys.RIGHT))
                bucket.x += 200 * Gdx.graphics.getDeltaTime();

        // make sure the bucket stays within the screen bounds
        if (bucket.x < 0)
                bucket.x = 0;
        if (bucket.x > 800 - 64)
                bucket.x = 800 - 64;

        // check if we need to create a new raindrop
        if (TimeUtils.nanoTime() - lastDropTime > 1000000000)
                spawnRaindrop();
```