COMP20050 Software Engineering Project 2

14. LibGDX details (Part 1)

Assoc. Prof. Pavel Gladyshev

UCD School of Computer Science. **Scoil na Ríomheolaíochta UCD.**

LibGDX details

- Gdx package has several static objects, which are used for accessing different aspects of LibGDX functionality in a platform-independent fashion:
  - **Gdx.app** - platform-specific application object (e.g. Lwjgl3Application) It sets up a window and rendering surface and other aspects of the applicaiton
  - **Gdx.graphics** – encapsulates communication with the graphics processor.
  - **Gdx.audio** – encapsulates creation and use of sound resources.
  - **Gdx.input** – platform-independent interface to input facilities.
  - **Gdx.files** – provides access to the file system, jar file resources, etc.
  - **Gdx.net** - provides methods to perform networking operations.

LibGDX Application

Lwjgl3Applicaiton

```
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application;
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3ApplicationConfiguration;

public class Main {
    public static void main (String[] arg) {
        Lwjgl3ApplicationConfiguration config = new Lwjgl3ApplicationConfiguration();
        config.setTitle("Example1");
        config.setWindowedMode(600,400);
        config.setResizable(true);
        new Lwjgl3Application(new MyGame(), config);
    }
}
```

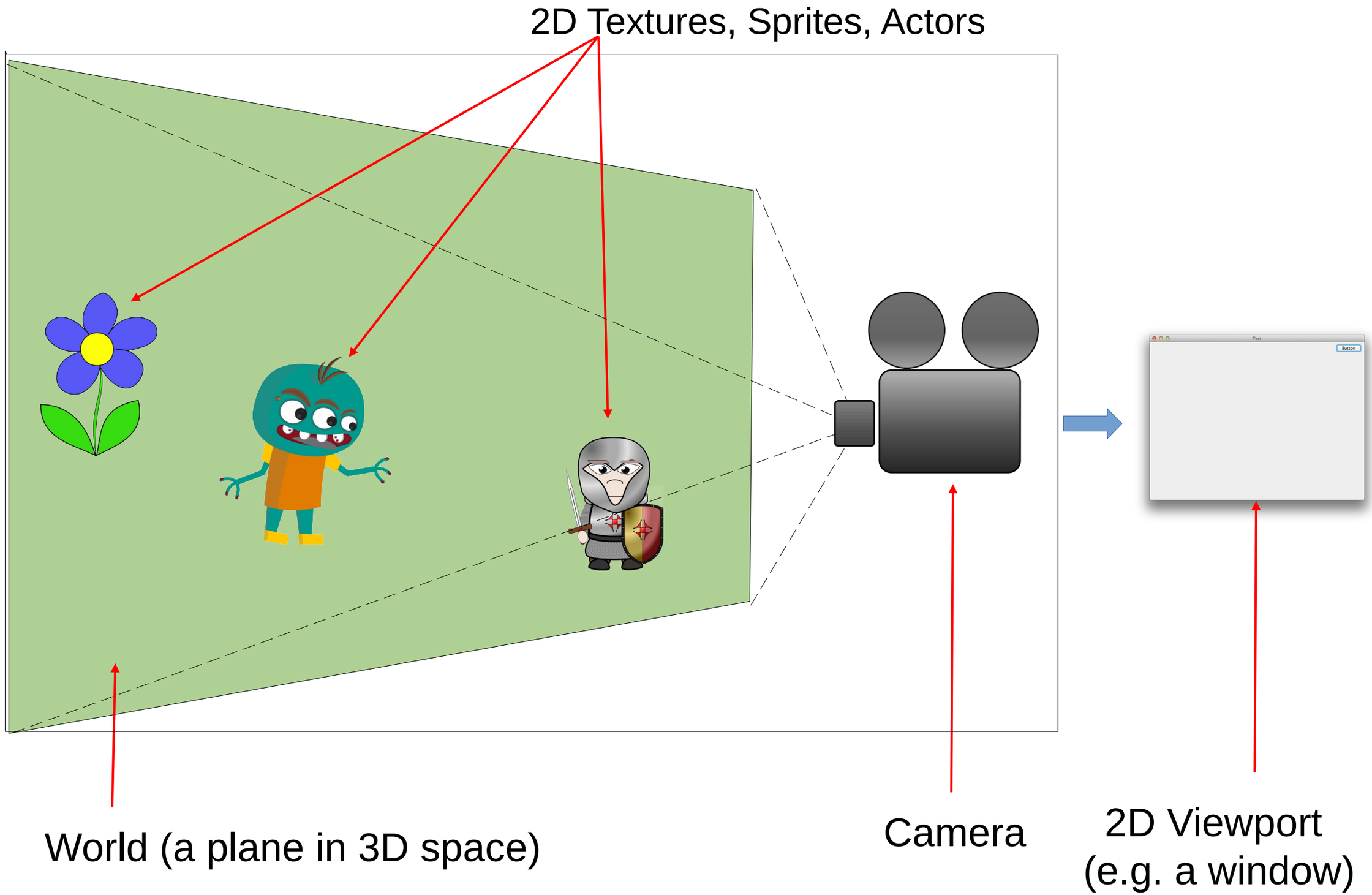
- setWindowedMode(width,heights) specifies the size of the game window in pixels
- SetResizeable(false) makes the game window non-resizeable
- For other config options see https://bit.ly/3GNs5TC

LibGDX Graphics

Batch drawing in LibGDX

- Computer game application redraws its game display several times per second.
- The process of “drawing” usually displays an arrangement of pre-created images, whose positions and orientations may change over time.
- LibGDX uses OpenGL ES API for graphics.
- OpenGL is a standard API for graphics acceleration hardware, such as Nvidia graphics cards, or graphics hardware embedded in modern Intel CPUs and AMD APUs.
- OpelGL is designed to work with 3-dimensional models. It takes a specification of 3D objects and textures, renders them in 3D space and then projects the resulting 3D scene to the 2D computer display.
- Since all rendering is done by graphics hardware, LibGDX drawing is done in batches: first, the CPU prepares a description of the textures and geometric models to be drawn, then it sends the prepared information to the graphics hardware, and the rendering is performed by the graphics hardware.

LibGDX uses 3D graphics for 2D games



Texture, Batch, and Camera

LibGDX performs drawing via objects implementing Batch interface. One such implementation is **SpriteBatch** class. To use it we need

- an instance of SpriteBatch
- a **Texture** (or TextureAtlas) to be displayed
- an **OrthographicCamera** configured to project the 3D “world” plane onto the game window

- The process of drawing is as follows:

  1. the camera’s combined matrix is given to the SpriteBatch object.
  2. the batch drawing starts with a call to SpriteBatch.begin()
  3. one or (many) more calls to SpriteBatch.draw() specify where on the world plne the texture (or its fragments) are to be diplayed
  4. the batch draw ends with a call to SpriteBatch.end(), which sends the data to the graphics card, which does the actual rendering

An example

```
import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.ScreenUtils;

public class MyGame extends ApplicationAdapter {

    Texture hacker;
    SpriteBatch batch;
    OrthographicCamera camera;

    @Override
    public void create() {
        hacker = new Texture(Gdx.files.internal("hacker.png"));
        batch = new SpriteBatch();
        camera = new OrthographicCamera();
    }

    @Override
    public void render() {
        ScreenUtils.clear(1.0f, 1.0f, 1.0f, 1.0f);

        camera.setToOrtho(false,Gdx.graphics.getWidth(),Gdx.graphics.getHeight());
        camera.update();
        batch.setProjectionMatrix(camera.combined);
        batch.begin();
        batch.draw(hacker,0,0);
        batch.end();
    }

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

setToOrtho(yDown, h, w)

This method of OrthigraphicCamera specifies the portion of the World plane “visible” through camera viewport:

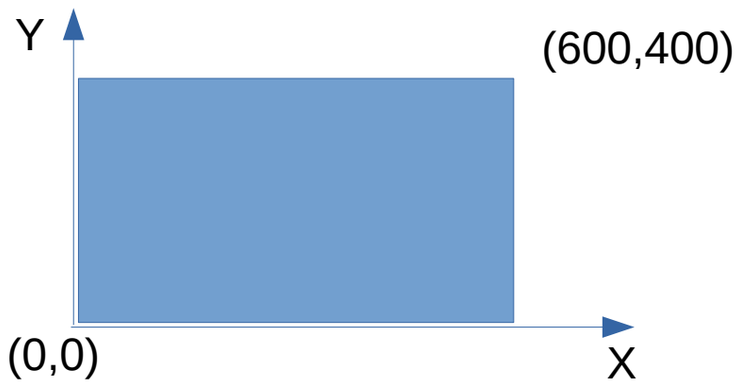
h – the height of the visible region (specified in the world coordinates).

w – the width of the visible region (specified in the world coordinates).

yDown – a boolean value. It specifies whether the Y axis of the world will be seen as pointing up (yDown = false) or down (yDown = true).

setToOrtho() requests the camera to “look” at the world rectangle (0,0)-(h,w)

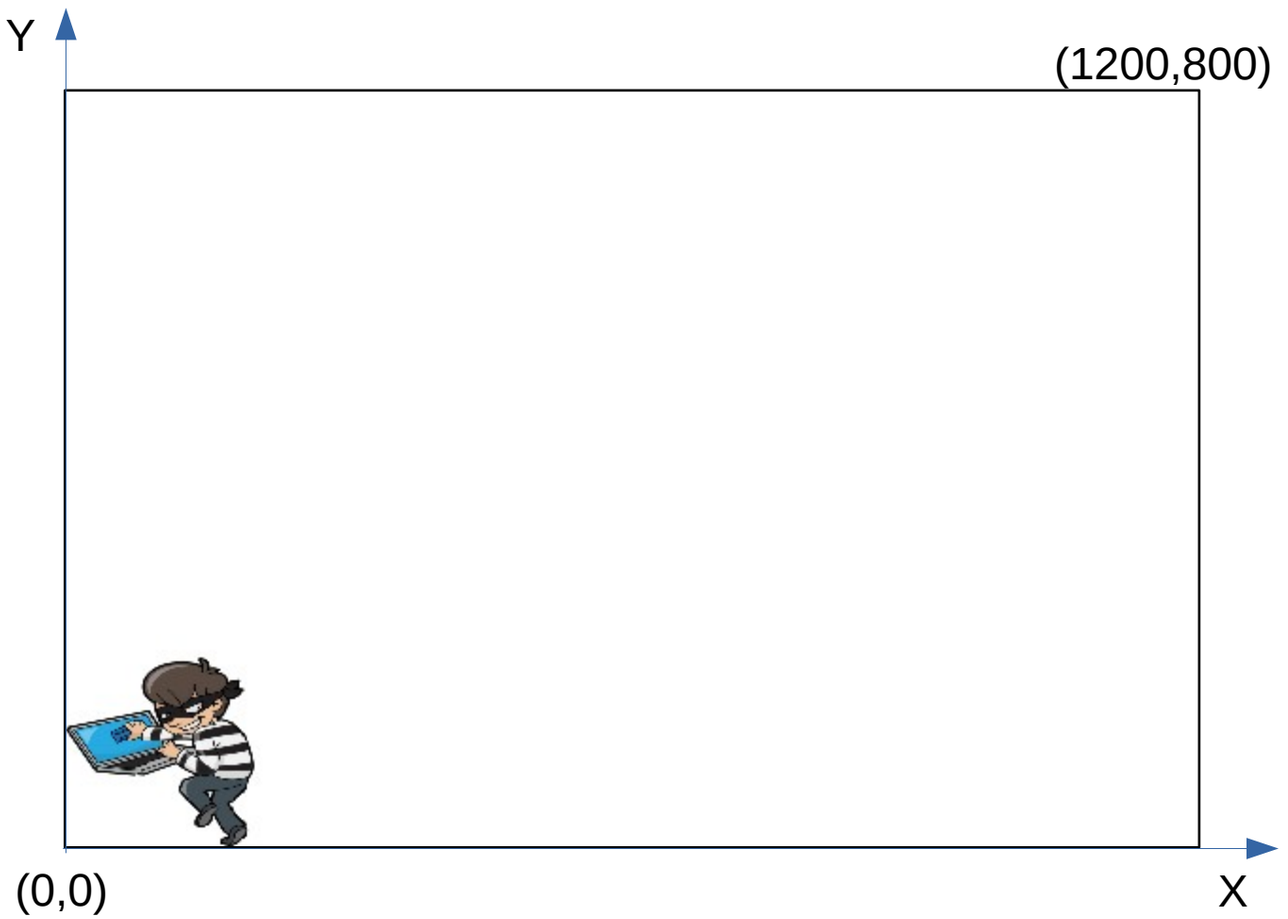
For example, camera.setToOrtho(false,600.0f,400.0f) requests the camera to display the following fragment of the world in the viewport:



By specifying appropriate setToOrtho() parameters we can “zoom in” or “zoom out”

camera.setToOrtho(**false**,600.0f,400.0f);

camera.setToOrtho(**false**,1200.0f,800.0f);



By specifying appropriate setToOrtho() parameters we can “zoom in” or “zoom out”

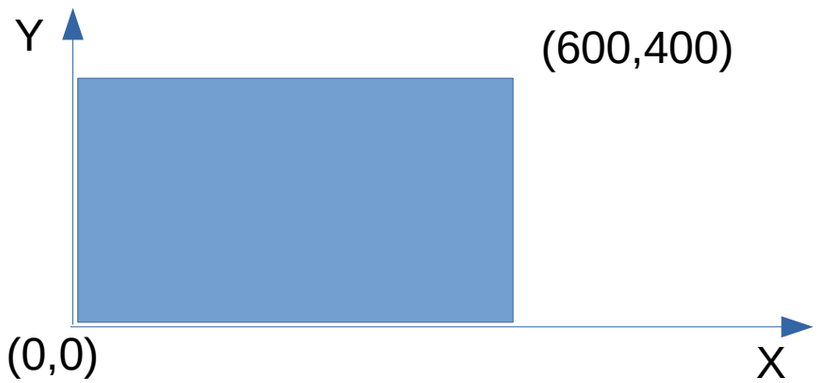
camera.setToOrtho(**false**,300.0f,200.0f);

camera.setToOrtho(**true**,300.0f,200.0f);

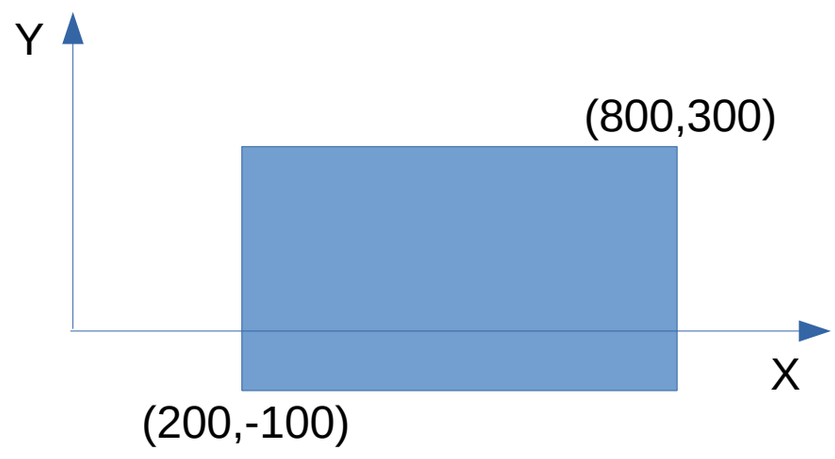
translate(deltaX, deltaY)

translate() moves the visible rectangle by the specified amounts along X and Y axes.

camera.setToOrtho(false,600.0f,400.0f);



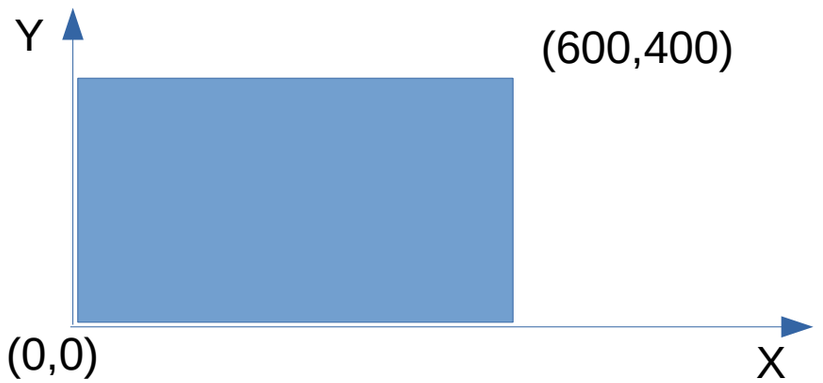
camera.translate(+200.0f, -100.0f);



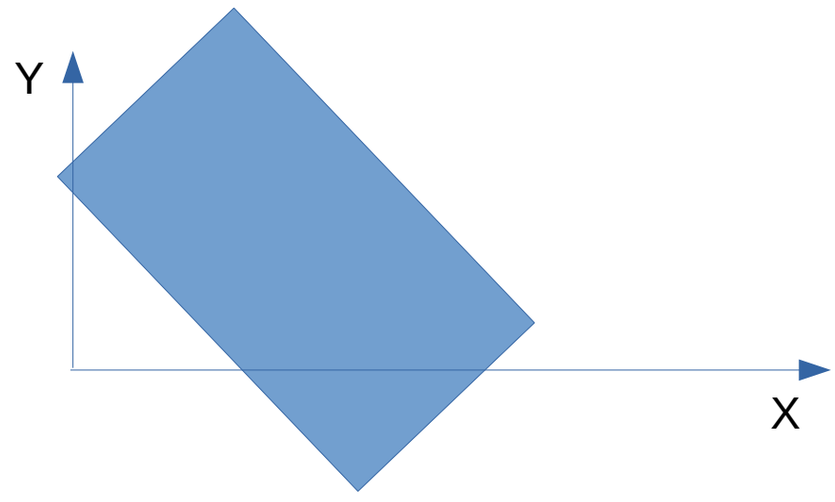
rotate(degrees)

rotate() rotates the camera (around the center of the visible region) by the specified amount of degrees.

camera.setToOrtho(false,600.0f,400.0f);



camera.rotate(45.0f);



update()

update() method takes the changes specified by setToOrtho(), translate(), and rotate() and calculates the so-called “combined” 4x4 matrix that transforms world points to screen pixel positions.

Important: any change of camera parameters must be followed by a call to update() in order to effect the requested changes:

```
camera.setToOrtho(false,600.0f, 400.0f);
camera.translate(+200.0f, -100.0f);
camera.rotate(45.0f);
camera.update();    // applies the changes to camera.combined matrix
```

batch.setProjectionMatrix(camera.combined)

The camera's combined matrix tells the SpriteBatch which part of the world to show to the player **after** the batch of sprites had been drawn.

- It possible to have multiple camera objects confiured to look at different parts of the world and use them interchangeably for consecutive batch draws.
- It is possible to have a camera that follows a game character and changes its position as the character moves around the game level.
- In the example supplied with this lecture, we continually change the dimensions of the view port to match the size of the game window [Gdx.graphics.getWidth(), Gdx.graphics.getHeight()] which allows the player to resize the game window without distorting the game display.

Successive batch draws are compounded

What would happen if we executed multiple batch draws using different camera settings in the same render() method?

The output of each batch draw is written on top of the output of the preceding batch draws.

This is roughly similar to how 1990s photo artisits took multiple photo exposures onto the same photographic film to produce LOMOgraphy:

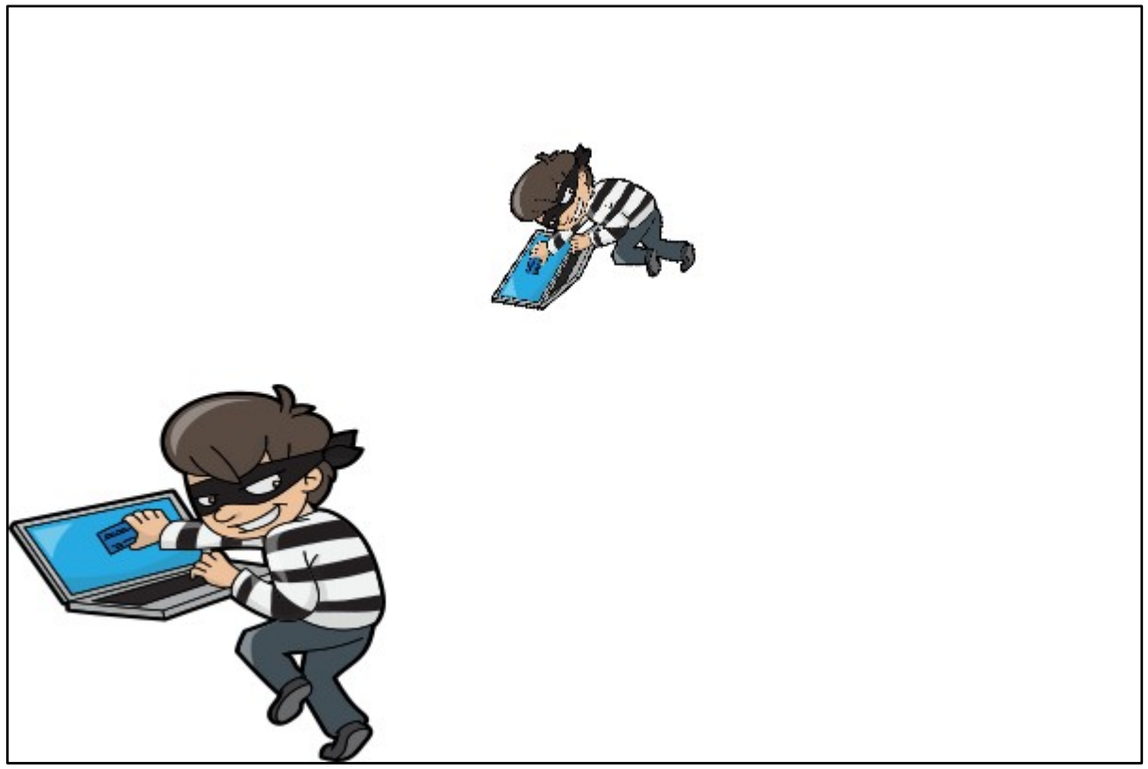
Photo by alnbbates (https://flic.kr/p/ejfPEg)

Successive batch draws with different camera settings on the same screen

```
camera.setToOrtho(false,Gdx.graphics.getWidth(),Gdx.graphics.getHeight());
camera.update();
batch.setProjectionMatrix(camera.combined);
batch.begin();
batch.draw(hacker,0,0);
batch.end();
```

```
camera.setToOrtho(false,Gdx.graphics.getWidth()*2,Gdx.graphics.getHeight()*2);
camera.translate(-Gdx.graphics.getWidth(), -Gdx.graphics.getHeight());
camera.rotate(45.0f);
camera.update();
batch.setProjectionMatrix(camera.combined);
batch.begin();
batch.draw(hacker,0,0);
batch.end();
```

Successive batch draws with different camera settings on the same screen



As an exercise, study the code on the preceding slide and try to explain how the smaller hacker image was produced: why does it have the size, the position, and the angle that it has?