Android Game Development 101

How I Learned to Love the Canvas

Badlogic Games

Agenda

1. Android – A Gaming Platform
2. Game Design 101
3. Game Programming 101
4. Android for Game Developers
5. Lunch Break (Ohm Nom Nom Nom)
6. Droidanoid

Before We Start

- Eclipse http://www.eclipse.org/
- Android SDK & ADT
  http://developer.android.com/sdk/installing.html
- Google Code Page (slides & sources):
  http://code.google.com/p/agd-101/
- See video on how to perform the setup.

1 - Why Develop for Android?

- **~300k activations per day == huge audience.**
- **Instant push to gamers via Android Market.**
- **One time fee of 25$ to become a publisher.**
- **No walled garden.**
- **Free Development Tools.**
  - Eclipse, Java (erm Dalvik…), Android SDK, ADT
- **Hardware rivaling dedicated gaming platforms, e.g. Gameboy, PSP.**
- **Because it‘s awesome!**

1 – Mobile Gaming is Different

- **Mobile Phones have become the new GameBoys.**
- **Covers an even larger demographic!**
  - Hardcore gamers.
  - Casual gamers.
  - Mom, Pop and even Grandma play games now.
- **Always connected.**
- **New distribution channels.**
- **Big Market, Small Developers.**

Badlogic Games

1 – Genres: to Each One's Taste

Casual Games

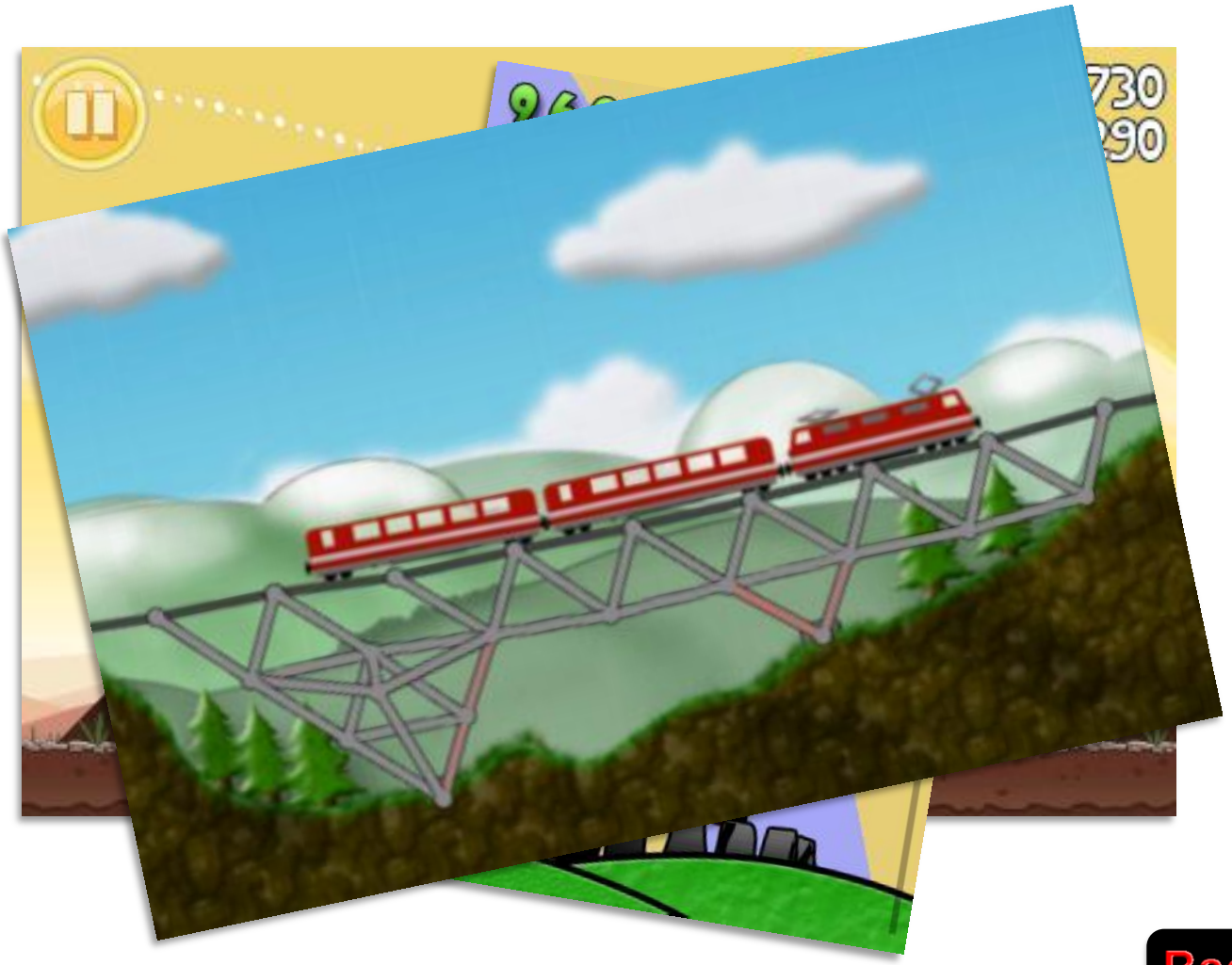
1 – Genres: to Each One's Taste

Casual Games

Badlogic Games

1 – Genres: to Each One's Taste

Puzzle Games



Badlogic Games

1 – Genres: to Each One's Taste

Puzzle Games

Badlogic Games

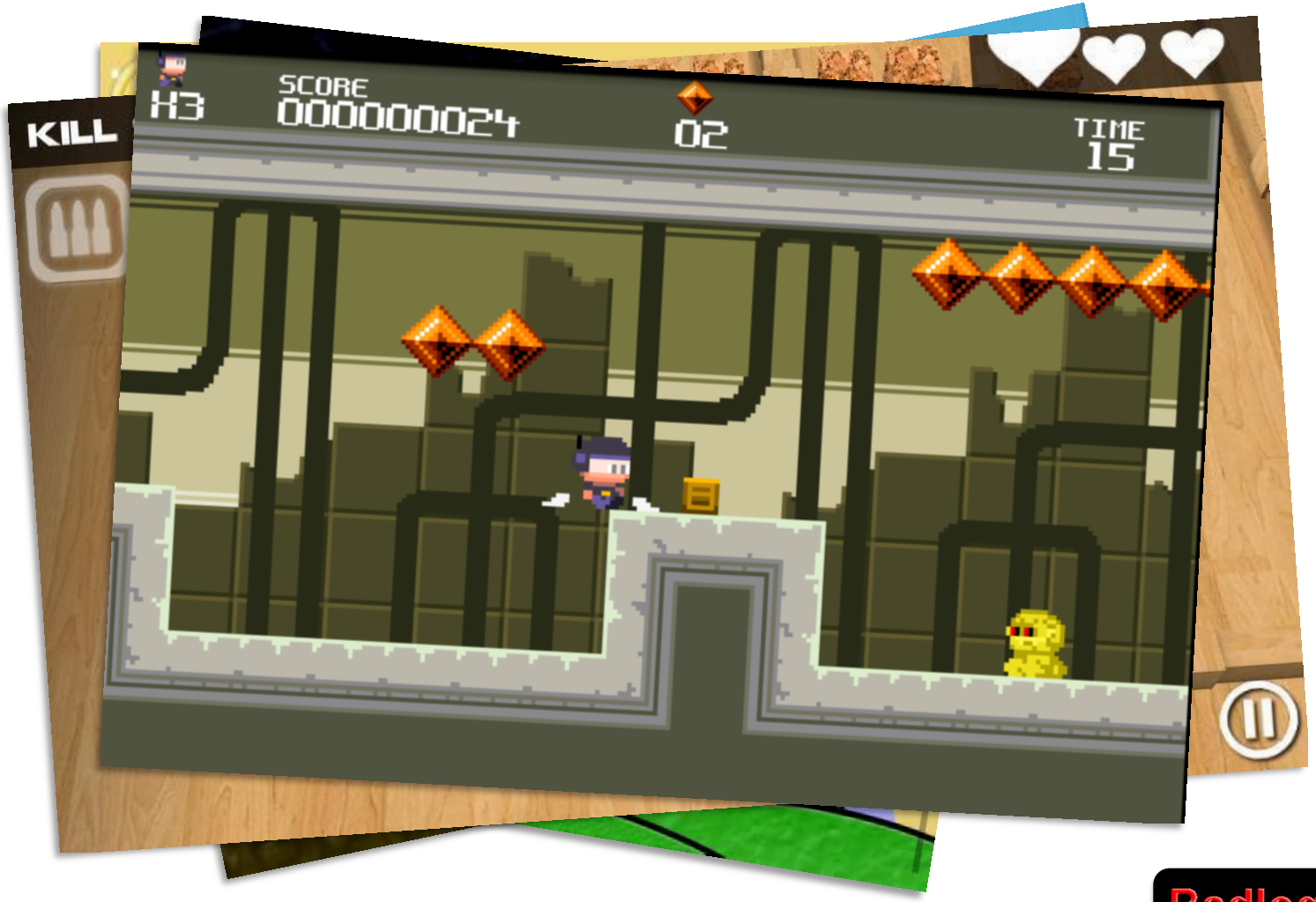
1 – Genres: to Each One's Taste

Arcade & Action Games

Badlogic Games

1 – Genres: to Each One's Taste

Arcade & Action Games



Badlogic Games

1 – Hardware

- **ARM CPU (500Mhz to 1Ghz dual cores).**
- **Plenty of RAM (128MB to 1GB).**
  - Only 20-25MB per application though.
- **Internal storage (256MB to 8GB).**
- **External storage (Usually a couple GB).**
- **Capacitive Touch Screen.**
- **System Keys (Home, Back, Menu, Search)**
  - Not all devices have all four keys! WTF?
- **Accelerometer.**

1 – Hardware

Fragmentation Much?

- **Screen resolutions and sizes.**
  - 320x240, 480x320, 800x480, 854x480, …
  - 3.2“, 4.1“, 4.3“…
- **Audio Latencies.**
  - Samsung Galaxy S, Y U HAVE SO HIGH LATENCY?
- **Shoddy OpenGL ES drivers.**
  - „Qualcooooooomm“ (in Captain Kirk voice).
  - Not relevant for this workshop.
- **Keyboards, Trackballs etc.**
  - Some have them, some don‘t.
- **Looks worse than it really is.**

1 - Hardware

First Generation

- **HVGA (320x480)**
- **Qualcomm CPU/GPU**
  - MSM720xA
- **No true multi-touch**
- **Comes with Android 1.5/1.6**
- **Upgradable to 2.1**
- **If you see one of those, RUN!**

Badlogic Games

1 - Hardware

Second Generation

- **QVGA, HVGA, WVGA.**
  - 320x240, 480x320, 800x480
- **Diverse CPU/GPU.**
  - Qualcomm Snapdragon
  - PowerVR SGX 530/535
- **Still no true multi-touch for some.**
- **Android 2.1, 2.2.**
- **Bigger, Better, Faster.**

1 - Hardware

Next Generation

- **Dual Core CPUs**
- **Nvidia Tegra GPUs (Yay!)**
- **Finally proper multi-touch.**
- **New form factors.**
  - Tablets, e.g. Xoom, Toshiba Pad.
- **Android 2.2, 2.3, 3.0**
- **Holy S%&$t that‘s fast!**

Badlogic Games

1 - Software

- **Android 1.5/1.6**
  - Low-End API, no multi-touch.
  - Dalvik stop-the-world GC pauses, no JIT.
- **Android 2.1**
  - Multi-touch API. (But broken touch screens make kitty sad)
  - Dalvik improvements, GC still s%&$t, no JIT either.
- **Android 2.2**
  - Finally JIT!
  - Slightly better GC
- **Android 2.3**
  - Concurrent GC, pauses less intrusive.
- **Android 3.0**
  - Renderscript, not that important for game devs.

Agenda

1. ~~Android – A Gaming Platform~~
2. Game Design 101
3. Game Programming 101
4. Android for Game Developers
5. Lunch Break (Ohm Nom Nom Nom)
6. Droidanoid

Badlogic Games

2 – Plan Of Attack

What's a game made of?

- **Core Game Mechanics.**
- **Story & Art Style.**
- **Screens and Transitions.**

Note: We do not write any code at this point! It's all about design. The code will emerge from the design naturally.

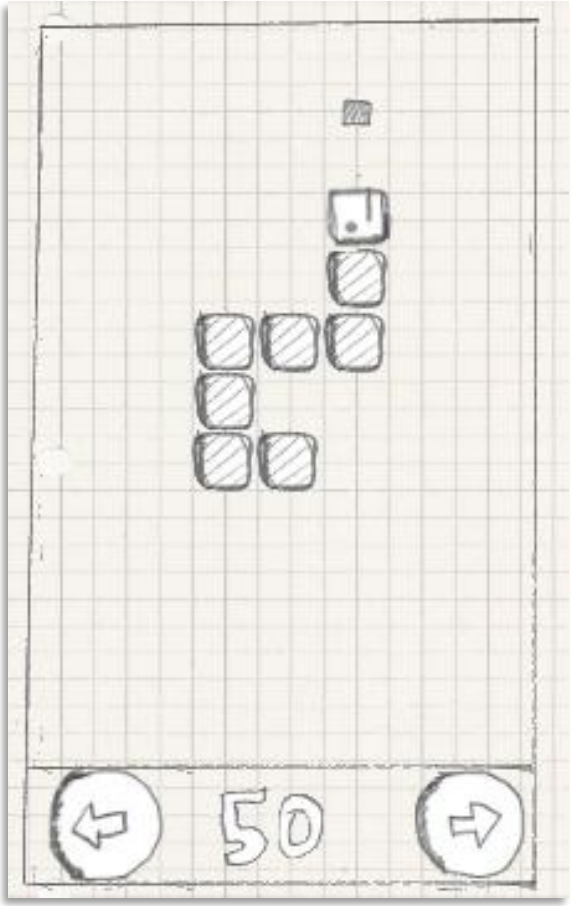
2 – Core Game Mechanics

- What objects are there in our world?
- How do they behave?
- How do they interact?
- How does the player interact with the game world?

Let‘s look at an old classic: Snake.

2 – Core Game Mechanics

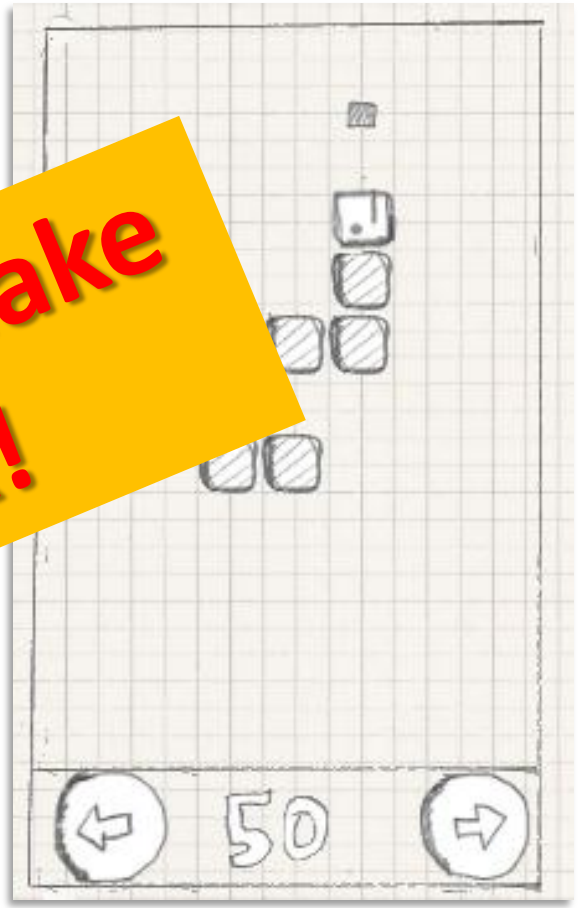
- The world is split up into a grid of cells.
- The snake is composed of cell sized parts (one head and a number of tail parts).
- The snake advances in the direction its head is pointed .
- The snake can only advance from cell to cell.
- If the snake goes outside the world boundaries, it reenters on the opposite side.
- If the right or left button is pressed, the snake takes a 90 degree clock-wise (right) or counter-clockwise (left) turn.
- If the snake hits a piece with its head, the piece disappears, the score is increased by 10 points, and a new piece appears on the playing field in a location that is not occupied by the snake itself. The snake also grows by one tail part. That new tail part is attached to the end of the snake.
- If the snake hits itself (e.g., a part of its tail), the game is over.



Badlogic Games

2 – Core Game Mechanics

- The world is split up into a grid of cells.
- The snake is composed of cell sized parts (one head and a number of tail parts).
- The snake advances in the direction its head is pointed .
- The snake can only advance from cell to
- If the snake goes outside the world boundaries, it reenters on the
- If the right or left button snake takes a 90 d counter-clock
- If the snake piece disapp ed by 10 points, an rs on the playing field i at is not occupied by the snake it snake also grows by one tail part. Th new tail part is attached to the end of the snake.
- If the snake hits itself (e.g., a part of its tail), the game is over.



Holy S&%$t! Snake is complex!

Badlogic Games

2 – Story & Art Style

A simple story

“Enter the world of Mr. Nom. Mr. Nom is a paper snake, always eager to eat drops of ink that fall down from an unspecified source on his paper land. Mr. Nom is utterly selfish and has only a single, not-so-noble goal: becoming the biggest ink-filled paper snake in the world!”

Badlogic Games

2 – Story & Art Style

- **The art style could be derrived from the story**
- **We chose a „doodle“ graphics style**
  - Easy to create, just use a pen, some paper and a scanner
- **The audio has to fit the graphics style**
  - Some silly sound effects when a stain is eaten.
  - Funny Music is neat as well.

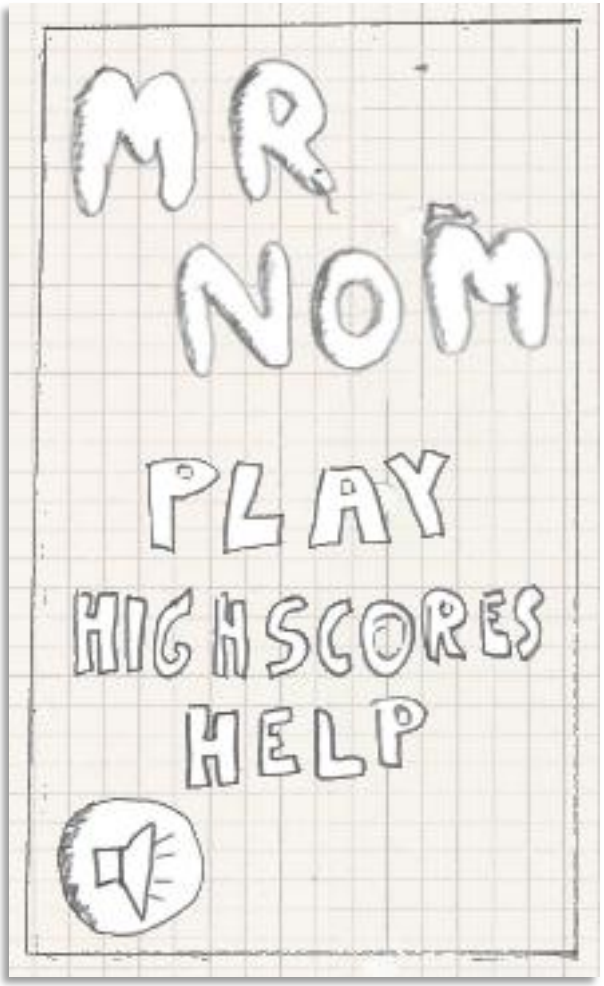
2 – Screens & Transitions

- **A game is more than just its mechanics**
- **We have different screens**
  - Main menu, highscore screen, game screen, etc.
- **Different events in a screen trigger a transition to another screen.**
- **We can also have screens within screens!**
  - Mind blown

2 – Screens & Transitions

Main Menu Screen

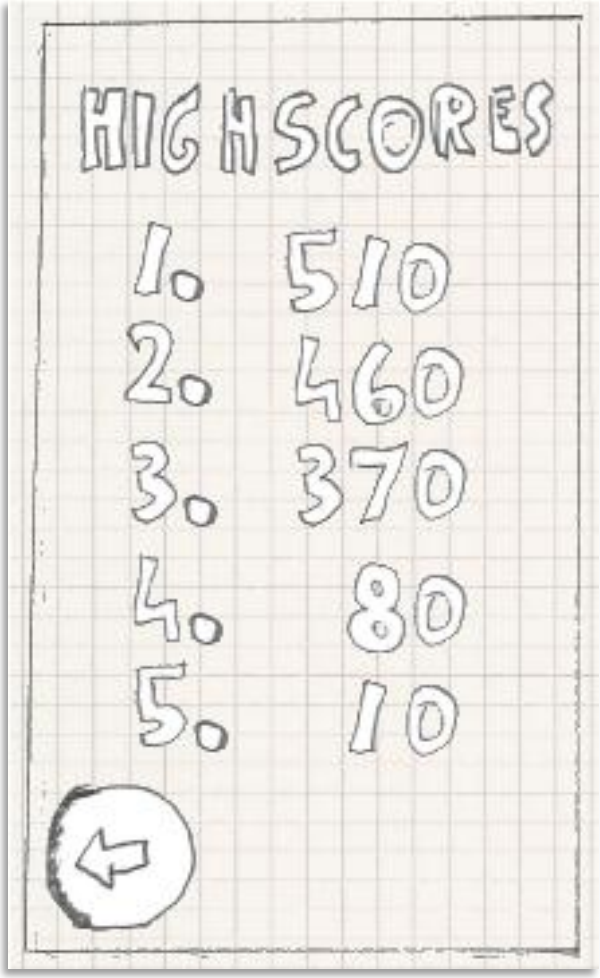
- Displays logo, menu items.
- Clicking on a menu item triggers a transition to another screen.
- The icon in the bottom is used to enable/disable audio.
- The audio configuration needs to be saved for the next session.



2 – Screens & Transitions

Highscore Screen

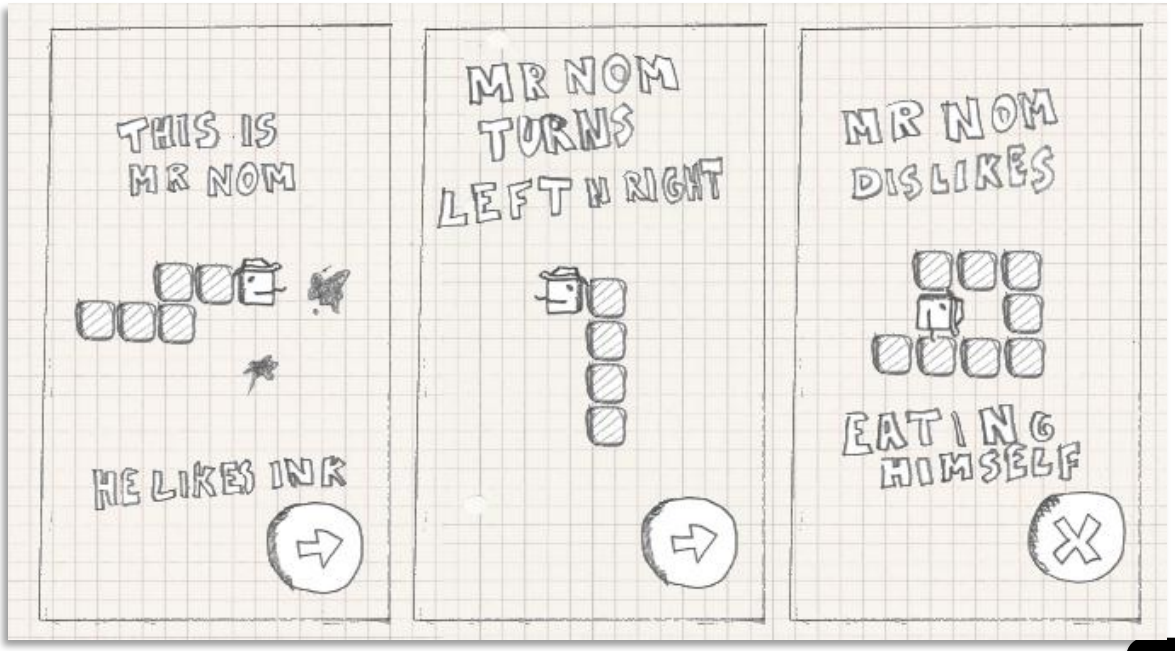
- Displays highscores.
- Highscores need to be loaded from somewhere.
- The button in the bottom will trigger a transition to the main menu screen.



2 – Screens & Transitions

Help Screens

- Display images, illustrating game mechanics.
- Buttons to transition to the next screen.

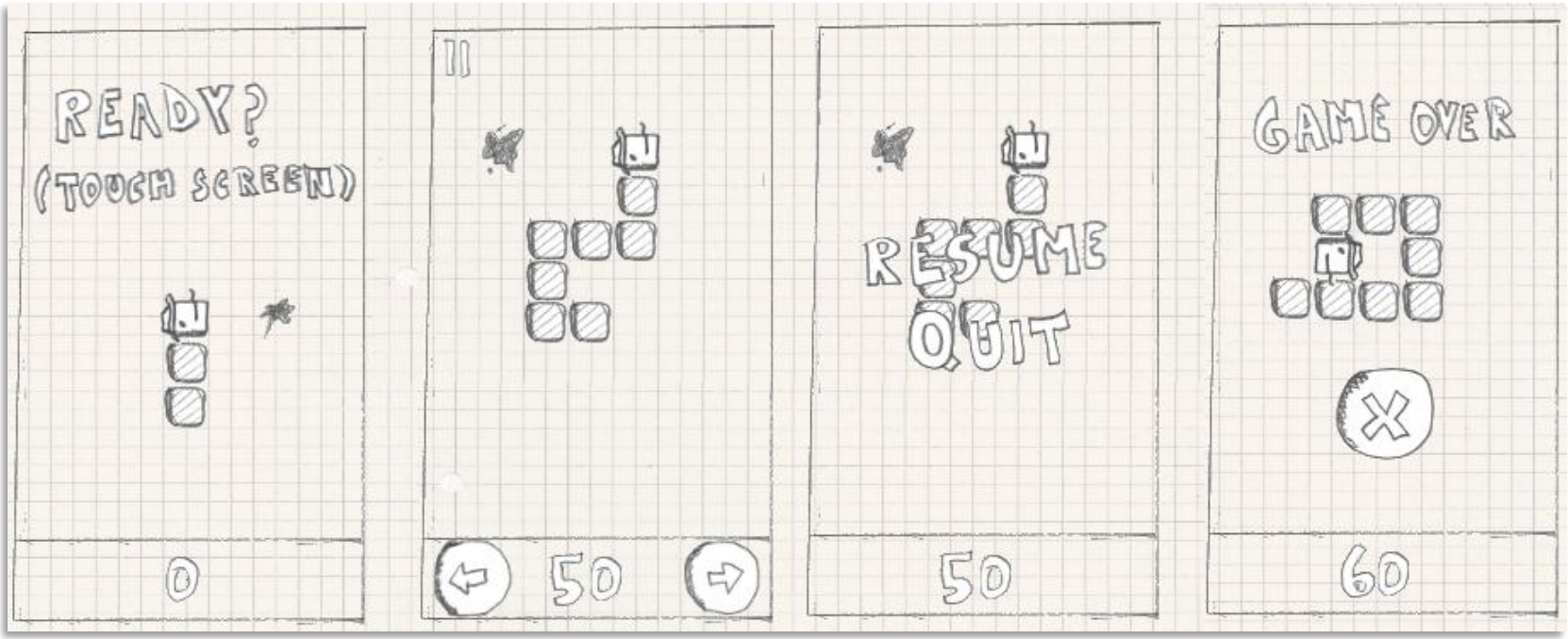


Badlogic Games

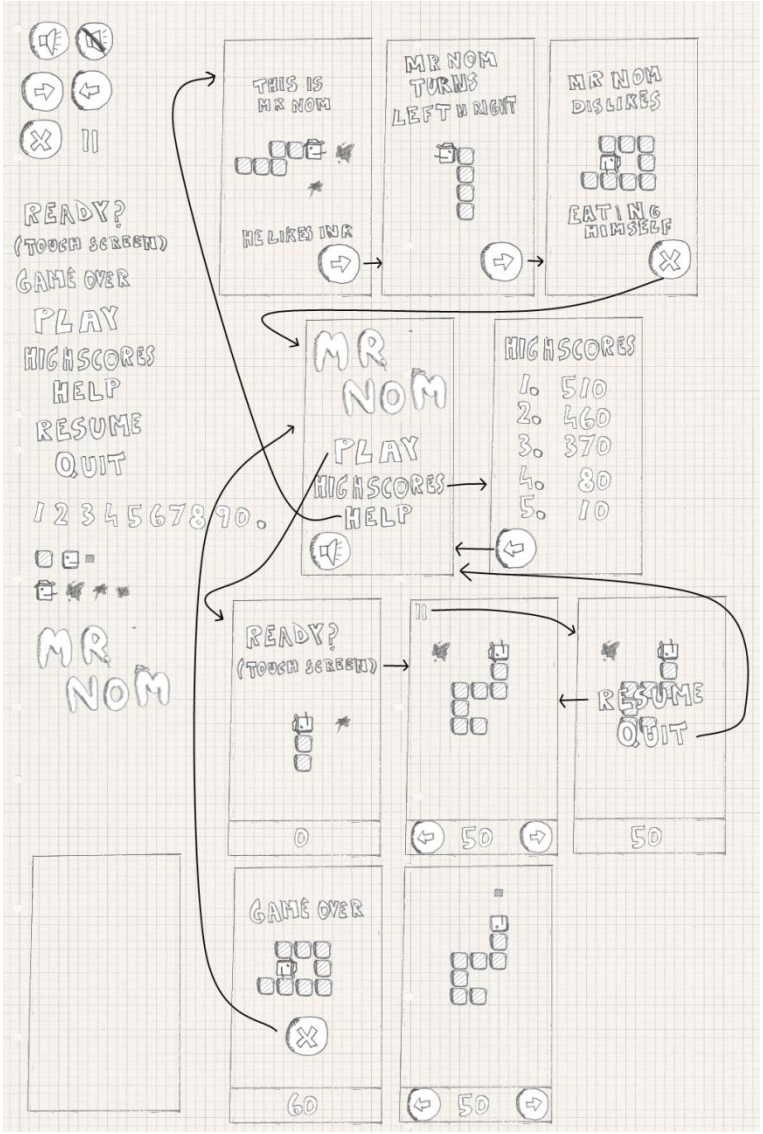
2 – Screens & Transitions

Game Screen

- Displays and runs the core game
- Has states: Ready, Running, Paused, Game Over



2 – Screens & Transitions



Badlogic Games

Agenda

1. ~~Android – A Gaming Platform~~
2. ~~Game Design 101~~
3. Game Programming 101
4. Android for Game Developers
5. Lunch Break (Ohm Nom Nom Nom)
6. Droidanoid

3 – Game Programming 101

- **Design tells us what we need code-wise.**
- **Low-Level**
  - Application Managment (Activity, Life-Cycle).
  - Loading and rendering graphics and audio.
  - Getting user input.
  - Persisting state (highscores).
- **High-Level**
  - Implementing the screens/UI.
  - Code that simulates our game world.

3 – What's Application Managment?

- **Activity on Android, Window on the desktop.**
- **Has to obey platform conventions.**
- **Life-Cycle!**
  - Game has to react to pause/resume gracefully.
  - Player can receive a call at any time and wants to pick up where she left.
- **Responsible for running the actual game.**
  - Main loop ahead!

3 – What's a Main Loop?

- **Our game is a big old endless loop.**
- **In each iteration we…**
  - Update the current game screen based on user input.
  - Draw the game screen's graphics.
  - Playback the game screen's audio.
- **We want to do this as often as possible!**
  - Low input latency.
  - Smooth Animations.
- **One such iteration is called a frame.**
- **We want high framerates (frames per second)**

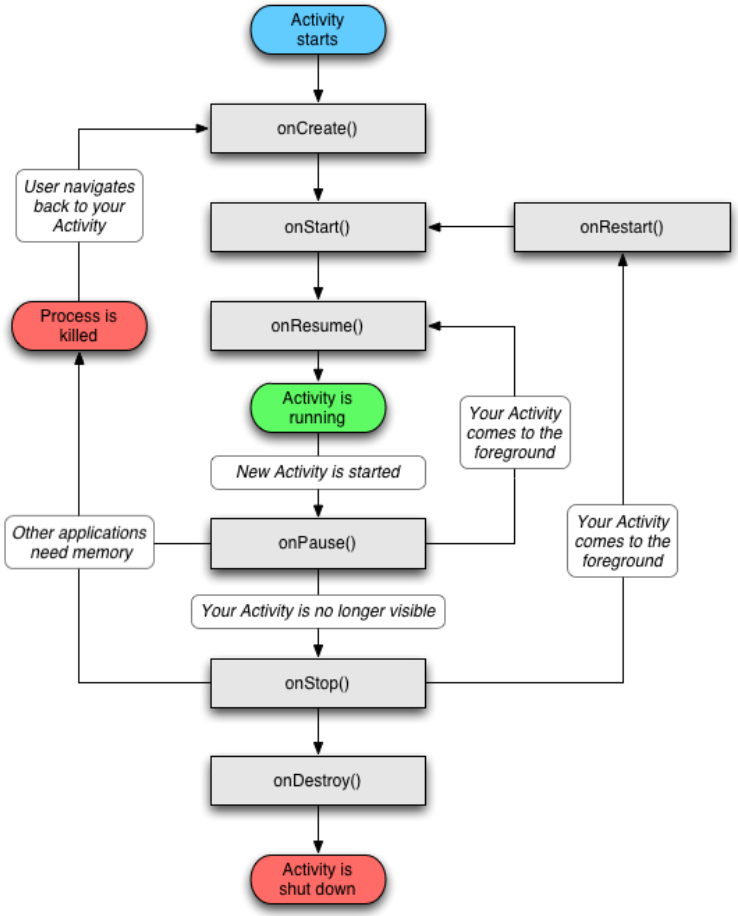
3 – What's a Main Loop?

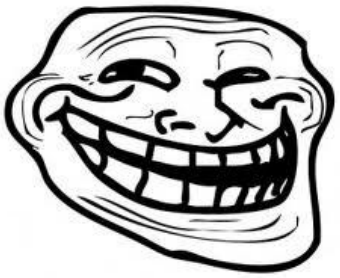
```
while(!gameOver) {
    InputState inputState = readInput();
    screen.update(inputState);
    screen.render();
}
```

3 – What's a Main Loop?

Problem?

- Where should we put our main loop?
- Android Activities work on UI thread, we only get callbacks for events!
- Can't block the UI thread!



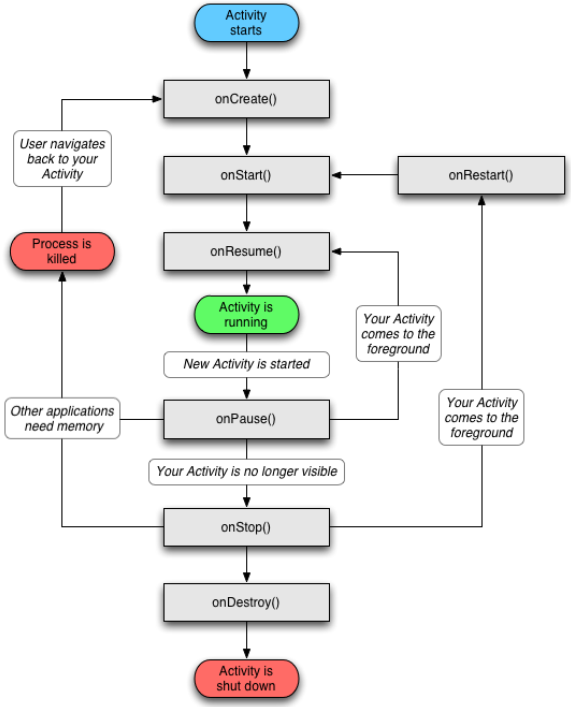


Badlogic Games

3 – What's a Main Loop?

Solution: we create our own main loop thread!

UI Thread



Main Loop Thread

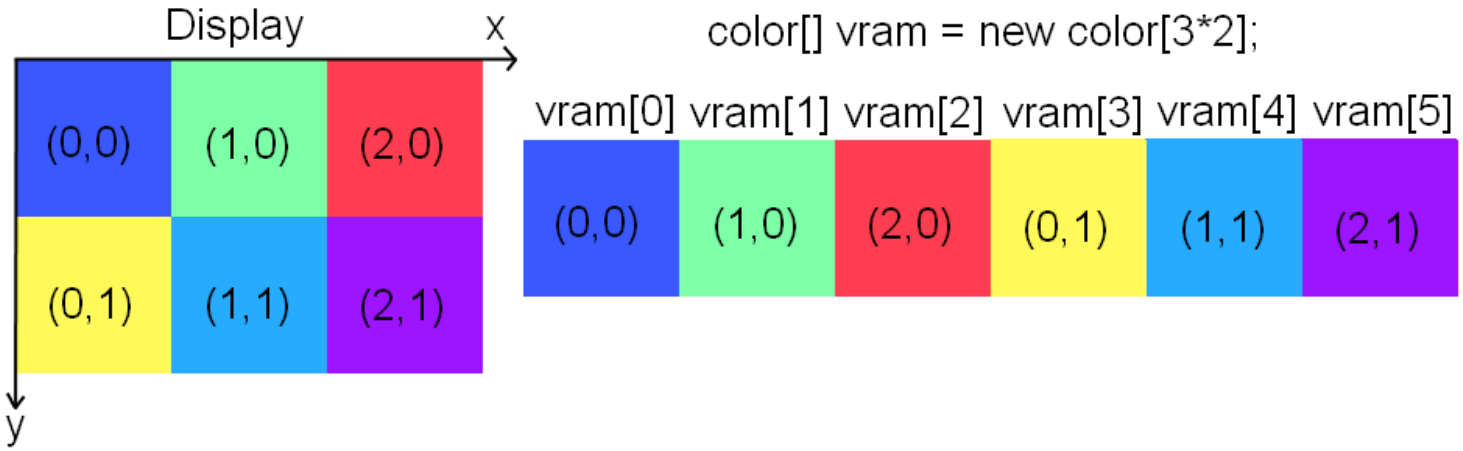
```
while(!gameOver && !activityDestroyed) {
    if(!paused) {
        InputState inputState = readInput();
        screen.update(inputState);
        screen.render();
    }
}
```

Just need to inform ML thread of UI events

Badlogic Games

3 – 2D Graphics Programming

- The display presents us the contents of something called the framebuffer.
- The framebuffer is an area in (V)RAM
- For each pixel on screen there‘s a corresponding memory cell in the framebuffer
- Pixels are addressed with 2D coordinates.

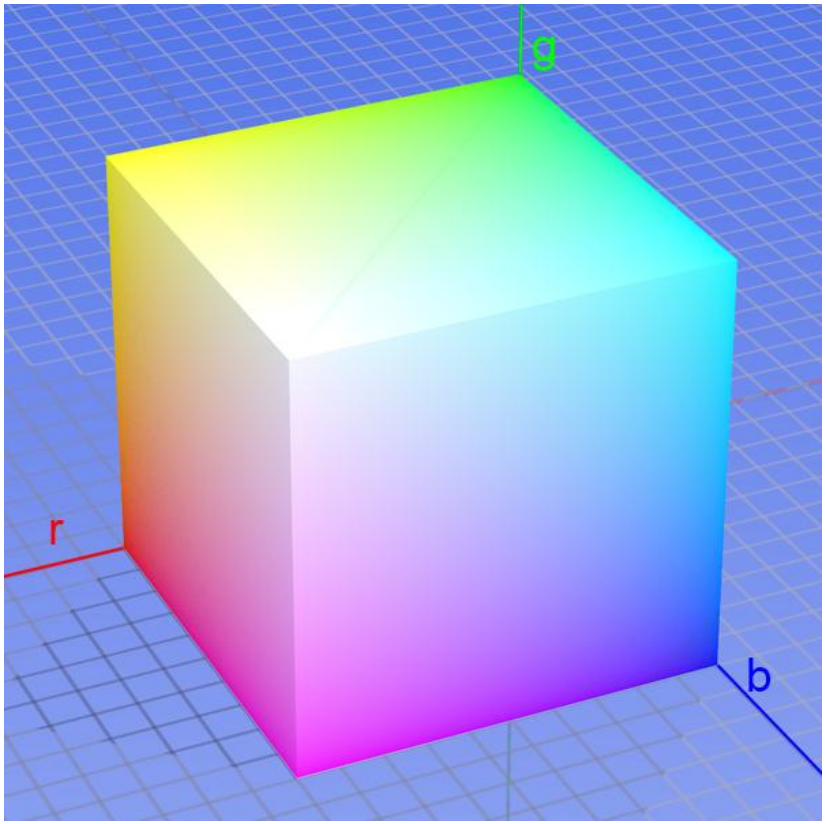


3 – 2D Graphics Programming

- **To change what's displayed we change the colors of pixels in (V)RAM.**
- **Pixel colors are encoded as RGB or RGBA**
  - R -> red, G -> green, B -> blue, A -> alpha (transparency)
- **Different bit-depths / encodings**
  - RGB565: 16-bit, 5 bits red, 6 bits green, 5 bits blue.
  - RGB888: 24-bit, 8 bits for each channel.
  - RGBA8888: 32-bit, 8 bits for each channel.

float: (1.0, 0.5, 0.75)
24-bit: (255, 128, 196) = 0xFF80C4
16-bit: (31, 31, 45) = 0xFC0D

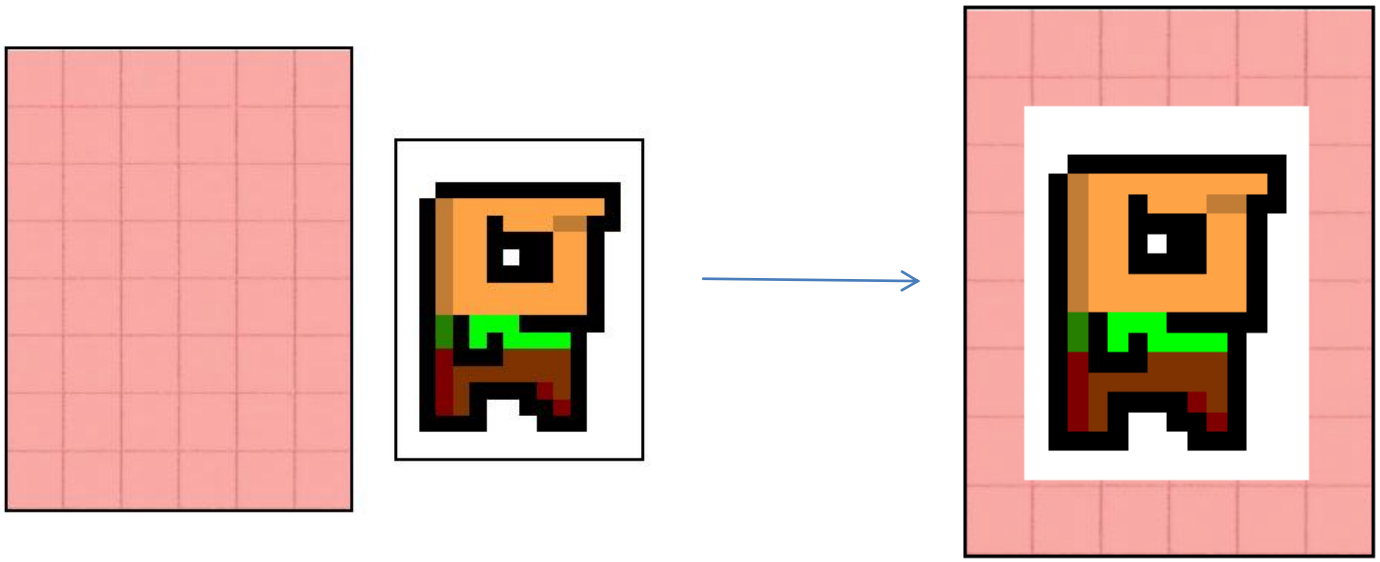
Behold the color cube!



3 – 2D Graphics Programming

- **To draw shapes we simply need to figure out which framebuffer pixels we have to set.**
- **Images (==bitmaps) aren‘t special either!**
  - Pixels of the bitmap get stored in a memory area, just like we store framebuffer pixels.
  - To draw a bitmap to the framebuffer we simply copy the pixels! (Blitting)
  - We can perform the same operations on bitmaps as we perform on the framebuffer, e.g. draw shapes or other bitmaps.

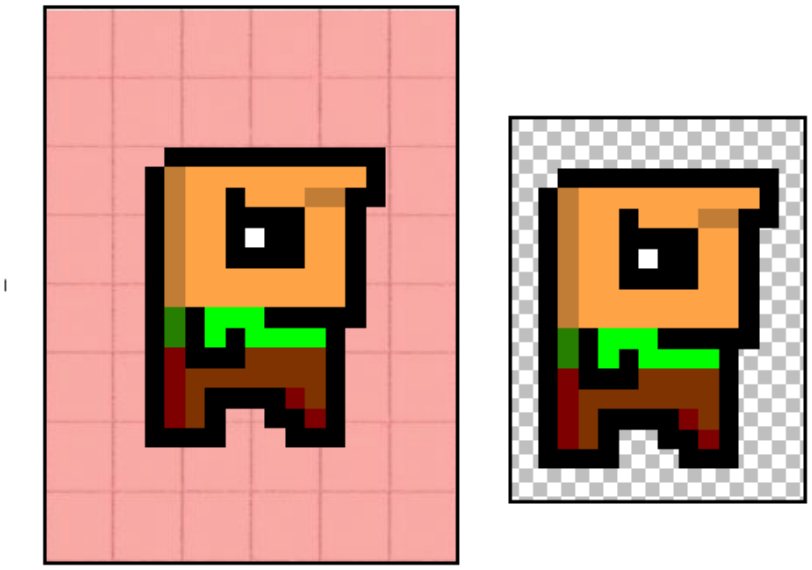
Blitting: copy (parts of) one bitmap to another.



3 – 2D Graphics Programming

Alpha Compositing: blitting + alpha blending.

- **Alpha value of a pixel governs transparency**
- **Instead of overwritting a destination pixel we mix its color with the source pixel.**



3 – 2D Graphics Programming

- **Two Options:**
  - SurfaceView & Canvas APIs.
  - OpenGL ES.
- **For many 2D games Canvas is enough.**
  - Hardware accelerated on latest Android!
  - Orange Pixel games are poster children (Meganoid!)
- **OpenGL ES comes with its own problems:**
  - Shoddy drivers.
  - Not a silver bullet for performance if you don‘t know what you do. Au Contraire!

3 – Audio Programming

- **Sound effects**
  - Short audio snippets like explosions, shots.
  - Completely loaded into RAM.
  - Can be played back multiple times, simultaniously.
- **Music**
  - Streamed on demand from the disk.
  - Too big to fit into RAM.
  - MP3 file of 3MB takes ~30MB decoded in RAM.

Input Processing

- **Polling**
  - Read the current state of the input devices.
    - Current touch points and positions.
    - Currently pressed keys.
    - Accelerometer status.
  - No chronology.
- **Events**
  - Callback when an input event happened (key down, etc.)
  - Chronological.

3 – Persisting State & File I/O

- **Reading/Writing Files**
- **Other means of state persistency**
  - Database
- **Not exactly rocket surgery.**

3 – Game & Screens

- **We should wrap all low-level stuff.**
- **A nice Game class would do!**
- **It would be responsible for:**
  - Managing the UI and main loop thread.
  - Keeping track of, update & render the current screen.
  - Allowing to transition from one screen to the next.
  - Passing on application events to the current screen.
  - Providing access to input events.
  - Loading and drawing bitmaps/playing sound & music
- **Let‘s devise a simple interface for that!**

3 – The Game Class

```
public class Game extends Activity {
    public abstract Screen createStartScreen();
    public void setScreen(Screen screen) { }

    public Bitmap loadBitmap(String fileName) { return null; }
    public Music loadMusic(String fileName) { return null; }
    public Sound loadSound(String fileName) { return null; }

    public void clearFramebuffer(int color) { }
    public void getFramebufferWidth() { return 0; }
    public void getFramebufferHeight() { return 0; }
    public void drawBitmap(Bitmap bitmap, int x, int y) { }
    public void drawBitmap(Bitmap bitmap, int x, int y,
                           int srcX, int srcY,
                           int srcWidth, int srcHeight) { }

    public boolean isKeyPressed(int keyCode) { return false; }
    public boolean isTouchDown(int pointer) { return false; }
    public int getTouchX(int pointer) { return 0; }
    public int getTouchY(int pointer) { return 0; }
    public List<com.badlogic.agd.KeyEvent> getKeyEvents() { return null; }
    public List<com.badlogic.agd.KeyEvent> getKeyEvents() { return null; }
    public float[] getAccelerometer();
}
```

3 – The Game Class

- **It‘s an Activity!**
- **It sets up the main loop thread and updates the current screen there.**
- **Screen transitions are triggered via Game.setScreen().**
- **The current screen can use its methods to:**
  - Load assets.
  - Render bitmaps and playback audio.
  - Get user input.
- **The createStartScreen() method must be implemented for each new game. It‘s the main entry point and returns the first screen!**

3 – The Screen Class

- **A screen implements one phase of our game.**
- **See design!**
- **It will be managed by the Game instance and updated as often as possible on the main loop thread.**
- **It‘s responsible for:**
  - Updating the state, drawing and playing back audio.
  - Triggering transitions by calling Game.setScreen().
- **Game reports life-cycle events to it.**

3 – The Screen Class

```
public abstract class Screen {
    protected final Game game;

    public Screen(Game game) {
        this.game = game;
    }

    public abstract void update(float deltaTime);
    public abstract void pause();
    public abstract void resume();
    public abstract void dispose();
}
```

3 – The Screen Class

- We store a Game instance reference.
- The update() method gets called as often as possible.
- The pause()/resume() methods get called based on life-cycle events.
- The dispose() method gets called when the app quits or a screen transition happens.
- All this methods are called on the main loop thread!
- Ignore the delta time parameter for now!

3 – A Simple Example

```
public class SimpleGame extends Game {
    public Screen createStartScreen() {
        return new SimpleScreen(this);
    }
}
```

- On startup createStartScreen() is called, returning the Screen the Game should handle.
- Next the SimpleScreen.update() method is called by the SimpleGame on the main loop thread.

```
class SimpleScreen extends Screen {
    int x = 0, y = 0;
    Bitmap bitmap;

    public SimpleScreen(Game game) {
        super(game);
        bitmap = game.loadBitmap("bitmap.png");
    }

    public void update(float deltaTime) {
        if(game.isTouchDown(0)) {
            x = game.getTouchX(0);
            y = game.getTouchY(0);
        }

        game.clearFramebuffer(Color.BLACK);
        game.drawBitmap(bitmap, x, y);
    }

    public void pause() { }
    public void resume() { }
    public void dispose() { }
}
```

3 – Run Through

1. The SimpleGame instance is created and will setup the main loop thread.
2. It then calls createStartScreen() and will update the returned Screen on the main loop thread as often as possible.
3. If the application is paused the Screen is informed and the main loop thread is paused.
4. If the application is resumed the Screen is informed and the main loop thread will continue updating the Screen again.
5. If the application is quit the Screen is informed and the main loop thread is destroyed.

3 – Code!

Let‘s implement all these classes for Android!

Agenda

1. ~~Android – A Gaming Platform~~
2. ~~Game Design 101~~
3. ~~Game Programming 101~~
4. Android for Game Developers
5. Lunch Break (Ohm Nom Nom Nom)
6. Droidanoid

Badlogic Games

4 – Demystifying the Manifest File

- The AndroidManifest.xml file governs how the Market filters our app.
- It also specifies the permissions we need.
- Additionally it lets Android know which Activities there are in our application.
- Here are 10 easy steps to create a Android game project in Eclipse.

4 – Demystifying the Manifest File

1. Create a new Android project in Eclipse by opening the New Android Project dialog
2. Specify your project's name and set the build target to the latest available SDK version.
3. Specify the name of your game, the package all your classes will be stored in, and the name of your main activity.
4. Set the minimum SDK version to 3.
5. Add the installLocation attribute to the <manifest> element in your manifest file and set it to "preferExternal".
6. Add the debuggable attribute to the <application> element and set it to "true".
7. Add the screenOrientation attribute to the <activity> element and specify the orientation you want ("portrait" or "landscape").
8. Set the configChanges attribute of the <activity> element to "keyboard|keyboardHidden|orientation".
9. Add two <uses-permission> elements to the <manifest> element and specify the name attributes "android.permission.WRITE_EXTERNAL_STORAGE" and "android.permission.WAKE_LOCK".
10. Add the targetSdkVersion attribute to the <uses-sdk> element and specify your target SDK.

4 – Demystifying the Manifest File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.badlogic.agd"
      android:versionCode="1"
      android:versionName="1.0"
      android:installLocation="preferExternal">

    <application android:icon="@drawable/icon"
     android:label="@string/app_name"
     android:debuggable="true">
        <activity android:name=".MyAwesomeGame"
                  android:label="@string/app_name"
                  android:configChanges="keyboard|keyboardHidden|orientation"
                  android:screenOrientation="landscape">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

    <uses-permission android:name="android.permission.WRITE_EXTERNALSTORAGE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="11"/>
</manifest>
```

Badlogic Games

4 – Plan of Attack

- We‘ll now implement the classes we just discussed.
- We‘ll implement the features iteratively, one step at a time.
- For each feature we implement we have an example that derives from Game.
- We declare each example as a startable Activity in the manifest file so we can test it.

4 – Wake Locks

- The screen has a time-out after which it gets dimmed.
- A no-go for most games.
- We can fix this by using a wake lock.
- Remember: we have to add a permission to the manifest file!

4 – Wake Locks

```
public abstract class Game extends Activity {
    private WakeLock wakeLock;

    public void onCreate(Bundle instanceBundle) {
        super.onCreate(instanceBundle);
        PowerManager powerManager = (PowerManager)
getSystemService(Context.POWER_SERVICE);
        wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK,
"Game");
    }

    public void onPause() {
        super.onPause();
        wakeLock.release();
    }

    public void onResume() {
        super.onResume();
        wakeLock.acquire();
    }

...
```

4 – Wake Locks

- Create a new class called WakeLockExample.
- Derrive it from Game.
- Implement the createStartScreen() method by returning null.
- Add the Activity to the manifest file and run it.

```
<activity android:name=".examples.WakeLockExample"
        android:label="Wake Lock Example"
        android:configChanges="keyboard|keyboardHidden|orientation"
        android:screenOrientation="portrait">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
    </intent-filter>
</activity>
```

4 – Going Fullscreen

- That title bar and the info bar take away precious screen space.
- Let‘s get rid of these by going fullscreen!

4 – Going Fullscreen

```
public abstract class Game extends Activity {
    ...

    public void onCreate(Bundle instanceBundle) {
        ...
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                             WindowManager.LayoutParams.FLAG_FULLSCREEN);
    }

...
```

- Create a new example called FullscreenExample analogous to WakeLockExample.
- Run it!

4 – Adding the Main Loop Thread

- **Time to add the main loop thread.**
- **It will be part of our Game instance.**
- **It can be notified of the following states:**
  - Running
  - Paused
  - Resumed
  - Disposed
- **We‘ll use the Java Thread class which executes a Runnable.**
- **The Game will implement the Runnable interface and thus the main loop.**

4 – Adding the Main Loop Thread

- The life-cycle events will be reported on the UI thread via calls to onPause() and onResume().
- We have to pipe those events to the main loop thread so it can react to them, e.g. inform the current Screen.
- We create an enum called State encoding the four states.
- We have an ArrayList<State> we put all state changes from the UI thread into.
- The main loop thread reads that list in each iteration and reacts to the change.

4 – Adding the Main Loop Thread

```
public abstract class Game extends Activity implements Runnable {
    ...
    private Thread mainLoopThread;
    private State state = State.Paused;
    private List<State> stateChanges = new ArrayList<State>();

    ...

    public void run() {
        while(true) {
            synchronized (stateChanges) {
                for(int i = 0; i < stateChanges.size(); i++) {
                    state = stateChanges.get(i);
                    if(state == State.Disposed) { Log.d("Game", “disposed"); return; }
                    if(state == State.Paused) { Log.d("Game", "paused"); return; }
                    if(state == State.Resumed) { state = State.Running; Log.d("Game", "resumed");
                }
                stateChanges.clear();
            }
        }
    }
...
```

```
public enum State {
    Running,
    Paused,
    Resumed,
    Disposed
}
```

4 – Adding the Main Loop Thread

```
public abstract class Game extends Activity implements Runnable {
    ...
    public void onResume() {
        super.onResume();
        wakeLock.acquire();

        mainLoopThread = new Thread(this);
        mainLoopThread.start();
        synchronized (stateChanges) {
            stateChanges.add(stateChanges.size(), State.Resumed);
        }
    }

    public void onPause() {
        super.onPause();
        wakeLock.release();

        synchronized (stateChanges) {
            if(isFinishing()) {
                stateChanges.add(stateChanges.size(), State.Disposed);
            } else {
                stateChanges.add(stateChanges.size(), State.Paused);
            }
        }
        try { mainLoopThread.join(); } catch(InterruptedException e) { };
    }
...
```

4 – Adding the Main Loop Thread

- OMG, THREADING!
- We create the main loop thread each time Game.onResume() is called and sent a State.Resume event to the thread.
- We send a State.Dispose or State.Pause message in Game.onPaused() to thread which will quit the thread.
- There is no main loop thread while our application is running!

4 – Adding a SurfaceView

- We need a View to render to.
- For this purpose we can use the SurfaceView class.
- It‘s specifically tailored for our task (rendering in a separate thread).
- We create a SurfaceView and set it as the content view of our Game Activity.
- We get a SurfaceHolder from the SurfaceView.
- With the SurfaceHolder we first check if the surface is valid, then lock it, render to the surface and unlock it which makes the changes to our surface visible.

4 – Adding a SurfaceView

```
public abstract class Game extends Activity implements Runnable {
    ...
    private SurfaceView surfaceView;
    private SurfaceHolder surfaceHolder;

    public void onCreate(Bundle instanceBundle) {
        surfaceView = new SurfaceView(this);
        setContentView(surfaceView);
        surfaceHolder = surfaceView.getHolder();
    }

    public void run() {
        while(true) {
            …
            if(state == State.Running) {
                if(!surfaceHolder.getSurface().isValid()) continue;
                Canvas canvas = surfaceHolder.lockCanvas();
                // all drawing happens here! E.g.
                canvas.drawColor(Color.RED);
                surfaceHolder.unlockCanvasAndPost(canvas);
            }
        }
    }
...
```

4 – Adding Screen Managment

- **We need to manage the current Screen in our main loop thread.**
- **Easy, we just need to call its respective methods where appropriate.**
  - Screen.pause()/Screen.resume()/Screen.dispose() when we receive a life-cycle event in the stateChanges list.
  - Screen.update() in between the SurfaceHolder.lockCanvas()/Surfaceholder.unlockCanvasAndPost() calls.

4 – Adding Screen Managment

```
public abstract class Game extends Activity implements Runnable {
    ...
    private Screen screen;

    public void onCreate(Bundle instanceBundle) {
        ...
        screen = createStartScreen();
    }

    public void run() {
        while(true) {
            synchronized (stateChanges) {
                for(int i = 0; i < stateChanges.size(); i++) {
                    if(state == State.Disposed) { if(screen != null) screen.dispose(); ... }
                    if(state == State.Paused) { if(screen != null) screen.pause(); ... }
                    if(state == State.Resumed) { if(screen != null) screen.resume(); ... }
                }
                stateChanges.clear();
            }

            if(state == State.Running) {
                if(!surfaceHolder.getSurface().isValid()) continue;
                Canvas canvas = surfaceHolder.lockCanvas();
                if(screen != null) screen.update(0);
                surfaceHolder.unlockCanvasAndPost(canvas);
            }
        }
    }
```

Badlogic Games

4 – Adding Screen Managment

```
public abstract class Game extends Activity implements Runnable {
    ...

    public void setScreen(Screen screen) {
        if(this.screen != null) screen.dispose();
        this.screen = screen;
    }
    ...
}

public class ScreenManagmentExample extends Game {
    public Screen createStartScreen() {
        return new TestScreen(this);
    }

    class TestScreen extends Screen {

        public TestScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) { }
        public void pause() { Log.d("TestScreen", "pause"); }
        public void resume() { Log.d("TestScreen", "resume"); }
        public void dispose() { Log.d("TestScreen", "dispose"); }
    }
}
```

4 – Application Managment Complete!

- **We are done with our application managment:**
  - Setting up the main loop thread and a render surface.
  - Handling life-cycle events.
  - Managing screens.
- **One tiny little bit is missing: the delta time**
- **We‘ll add that later on once we understand why we need it!**
- **On to the graphics methods of Game.**

4 – Implementing the Graphics Methods.

- **Game has methods for:**
  - Clearing the framebuffer (our SurfaceView!).
  - Getting the framebuffer dimensions.
  - Loading & drawing Bitmaps.
- **Let‘s start with the first two, clearing the screen and getting the framebuffer dimensions.**

4 – Implementing the Graphics Methods.

- In Game.run() we get a Canvas instance.
- Canvas is a class provided by the Android API that let‘s us draw to a Bitmap or Surface(View).
- It has a ton of methods to draw shapes, bitmaps and so on.
- The drawing methods of Game will use this Canvas to execute.
- The method to clear the target of the Canvas is Canvas.drawColor(int color).
- The color argument is a 32-bit ARGB color.

4 – Implementing the Graphics Methods

```
public abstract class Game extends Activity implements Runnable {
    ...
    private Canvas canvas = null;

    ...
    public void run() {
        while(true) {
            ...
            if(state == State.Running) {
                if(!surfaceHolder.getSurface().isValid()) continue;
                canvas = surfaceHolder.lockCanvas(); // we store the Canvas in the member now!
                if(screen != null) screen.update(0);
                surfaceHolder.unlockCanvasAndPost(canvas);
                canvas = null;
            }
        }
    }

    public void clearFrameBuffer(int color) {
        if(canvas != null) canvas.drawColor(color);
    }

    public void getFramebufferWidth() { return surfaceView.getWidth(); }
    public void getFramebufferHeight() { return surfaceView.getHeight(); }
    ...
}
```

4 – Implementing the Graphics Methods

```
public class ClearExample extends Game {
    public Screen createStartScreen() {
        return new ClearScreen(this);
    }

    class ClearScreen extends Screen {
        Random rand = new Random();

        public ClearScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) {
            game.clearFramebuffer(rand.nextInt());
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Implementing the Graphics Methods

- Next we want to implement the Bitmap loading method.
- We will store all our images in the assets/ folder.
- Android messes with images that are stored in the res/drawable-XXX folders, we want more control.
- We‘ll use the BitmapFactory to load images.

4 – Implementing the Graphics Methods

```
public abstract class Game extends Activity implements Runnable {
    ...
    public Bitmap loadBitmap(String fileName) {
        InputStream in = null;
        Bitmap bitmap = null;
        try {
            in = getAssets().open(fileName);
            bitmap = BitmapFactory.decodeStream(in);
            if (bitmap == null)
                throw new RuntimeException("Couldn't load bitmap from asset '"
                        + fileName + "'");
            return bitmap;
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load bitmap from asset '"
                    + fileName + "'");
        } finally {
            if (in != null) try { in.close(); } catch (IOException e) { }
        }
    }
    ...
}
```

4 – Implementing the Graphics Methods

- **Drawing Bitmaps is just as easy as loading.**
- **We just need to invoke one of the methods of the Canvas class.**
- **The method is called Canvas.drawBitmap() and comes in two flavors:**
  - One for blitting the complete source Bitmap to the target.
  - One for blitting parts of the source Bitmap to the target, potentially stretched.

4 – Implementing the Graphics Methods

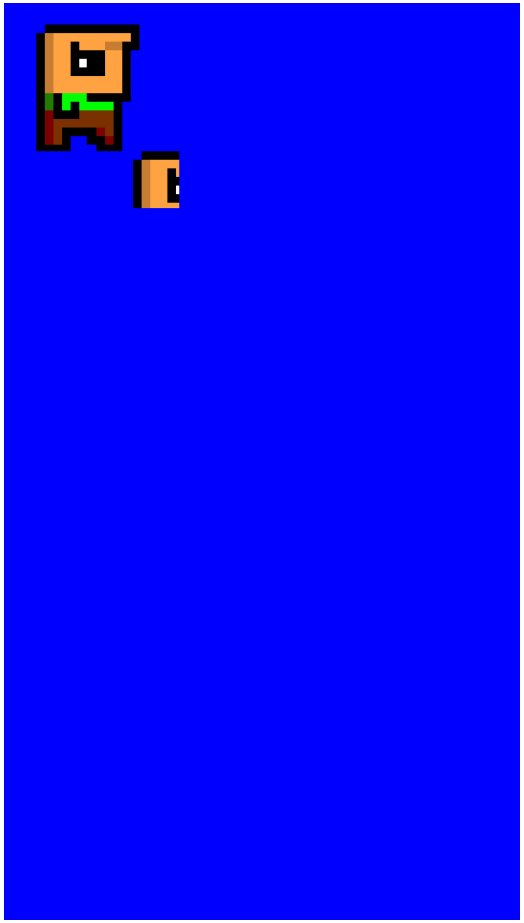
```
public abstract class Game extends Activity implements Runnable {
    ...
    public void drawBitmap(Bitmap bitmap, int x, int y) {
        if(canvas != null) canvas.drawBitmap(bitmap, x, y, null);
    }

    Rect src = new Rect();
    Rect dst = new Rect();
    public void drawBitmap(Bitmap bitmap, int x, int y, int srcX, int srcY,
                           int srcWidth, int srcHeight) {
        if(canvas == null) return;
        src.left= srcX;
        src.top = srcY;
        src.right = srcX + srcWidth;
        src.bottom = srcY + srcHeight;

        dst.left = x;
        dst.top = y;
        dst.right = x + srcWidth;
        dst.bottom = y + srcHeight;

        canvas.drawBitmap(bitmap, src, dst, null);
    }   ...
}
```

4 – Implementing the Graphics Methods

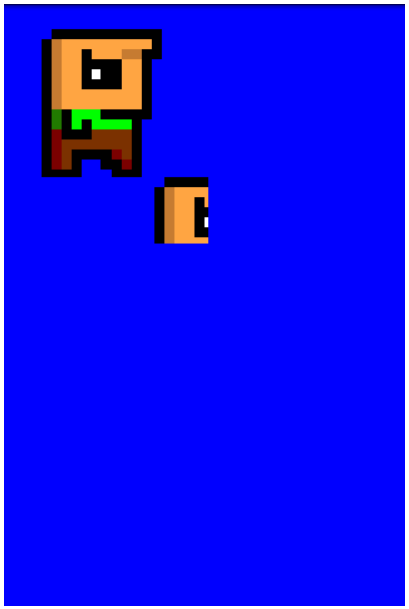
```
public class BitmapExample extends Game {
    public Screen createStartScreen() {
        return new BitmapScreen(this);
    }

    class BitmapScreen extends Screen {
        Bitmap bob;

        public BitmapScreen(Game game) {
            super(game);
            bob = game.loadBitmap("bob.png");
        }

        public void update(float deltaTime) {
            game.clearFrameBuffer(Color.BLUE);
            game.drawBitmap(bob, 10, 10);
            game.drawBitmap(bob, 100, 128, 0, 0, 64, 64);
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Different Screen Resolutions

Ouch!



480x854



320x480

Badlogic Games

4 – Different Screen Resolutions

- **We want all players to see the same.**
- **Problem:**
  - Different aspect ratios (width / height).
  - Different screen resolutions.
- **Simple solution:**
  - Assume a target resolution, e.g. 320x480.
  - Create a temporary bitmap of that size.
  - Direct all drawing to that bitmap.
  - At the end of a frame blit the bitmap to the real framebuffer, stretching it if necessary.
- **Copes with different screen resolutions, produces some stretching due to difference in aspect ratio.**
- **We work in a unified coordinate system, no matter the actual screen resolution!**
- **Good enough for most games!**

4 – Implementing the Graphics Methods

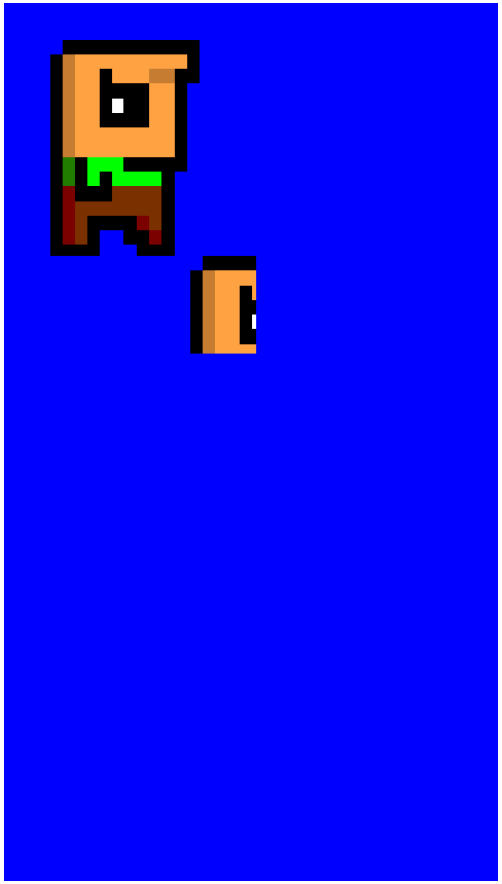
```
public abstract class Game extends Activity implements Runnable {
    ...
    private Bitmap offscreenSurface;

    public void onCreate(Bundle instanceBundle) {
        ...
        if(surfaceView.getWidth() > surfaceView.getHeight()) {
            setOffscreenSurface(480, 320);
        } else {
            setOffscreenSurface(320, 480);
        }
    }

     public void setOffscreenSurface(int width, int height) {
        if(offscreenSurface != null) offscreenSurface.recycle();
        offscreenSurface = Bitmap.createBitmap(width, height, Config.RGB_565);
        canvas = new Canvas(offscreenSurface);
    }

    public int getFramebufferWidth() {
        return offscreenSurface.getWidth();
    }

    public int getFramebufferHeight() {
        return offscreenSurface.getHeight();
    }
...
}
```

4 – Implementing the Graphics Methods

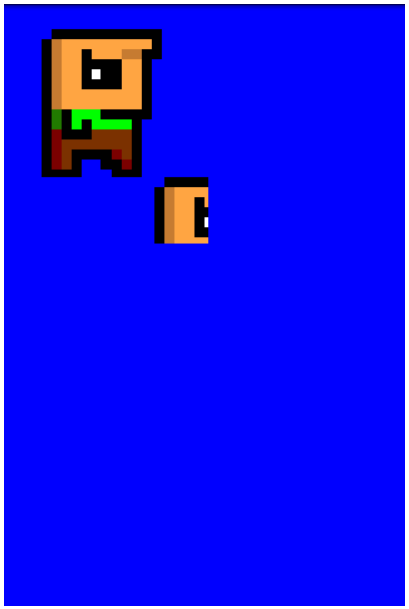
```
public abstract class Game extends Activity implements Runnable {
    ...
    public void run() {
        while(true) {
            ...
            if(state == State.Running) {
                if(!surfaceHolder.getSurface().isValid()) continue;
                Canvas canvas = surfaceHolder.lockCanvas();
                if(screen != null) screen.update(0);
                src.left = 0;
                src.top = 0;
                src.right = offscreenSurface.getWidth() - 1;
                src.bottom = offscreenSurface.getHeight() - 1;
                dst.left = 0;
                dst.top = 0;
                dst.right = surfaceView.getWidth();
                dst.bottom = surfaceView.getHeight();
                canvas.drawBitmap(offscreenSurface, src, dst, null);
                surfaceHolder.unlockCanvasAndPost(canvas);
            }
        }
    }...
}
```

4 – Different Screen Resolutions

Not perfect but better!



480x854



320x480

Badlogic Games

4 – Different Screen Resolutions

- We could also use a larger target resolution (e.g. 800x480) and scale down on lower resolution devices.
- Just call Game.setOffscreenSurface() with whatever resolution you want to work with.
- There‘s another solution to this but it has the downside of showing more/less of the game to a player depending on the aspect ratio.
- For our purposes the offscreen method is more than sufficient.

4 – Implementing the Input Methods

- We need to hook up listeners to the SurfaceView and SensorManager in order to get key and touch events as well as accelerometer readings.
- We need to pipe those events to the main loop thread since our Screen.update() method is called there.
- Polling is easy: we just store the status of the input devices in members of Game.
- Event-based input is harder due to threading.

4 – Implementing the Input Methods

- For touch events we register an OnTouchListener with the SurfaceView.
- For key events we register a OnKeyListener with the SurfaceView.
- For accelerometer readings we register a SensorEventListener with the SensorManager.
- In our implementation of the methods of these interfaces we simply store the current states.
- Let‘s begin with the (polled) key events.

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...
    private boolean pressedKeys[] = new boolean[256];

    public void onCreate(Bundle instanceBundle) {
        ...
        surfaceView.setFocusableInTouchMode(true);
        surfaceView.requestFocus();
        surfaceView.setOnKeyListener(this);
    }

    public boolean onKey(View v, int keyCode, KeyEvent event) {
        if(event.getAction() == KeyEvent.ACTION_DOWN)
            pressedKeys[keyCode] = true;

        if(event.getAction() == KeyEvent.ACTION_UP)
            pressedKeys[keyCode] = false;
        return false;
    }

    public boolean isKeyPressed(int keyCode) {
        return pressedKeys[keyCode];
    }
...
}
```

4 – Implementing the Input Methods

```
public class PollingKeyExample extends Game {
    public Screen createStartScreen() {
        return new PollingKeyScreen(this);
    }

    class PollingKeyScreen extends Screen {
        int clearColor = Color.BLUE;

        public PollingKeyScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) {
            game.clearFramebuffer(clearColor);

            if(game.isKeyPressed(KeyEvent.KEYCODE_MENU)) {
                clearColor = (int)(Math.random() * Integer.MAX_VALUE);
            }
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Implementing the Input Methods

- **Touch Events are a bit of a problem:**
  - No multi-touch API on Android 1.5/1.6
- **How can we support all Android versions?**
  - We create a touch event handler for Android 1.5/1.6 that can only process single-touch events.
  - We create another touch event handler for Android > 1.6 that can also process multi-touch events.
- **We also need to adjust the raw touch coordinates for the offscreen surface dimensions so we work in the same coordinate system!**

4 – Implementing the Input Methods

```
public interface TouchHandler {
    public boolean isTouchDown(int pointer);
    public int getTouchX(int pointer);
    public int getTouchY(int pointer);
}
```

- **Implementations are to big for slides. See:**
  - src/com/badlogic/agd/SingleTouchHandler.java
  - src/com/badlogic/agd/MultiTouchHandler.java

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...
    private TouchHandler touchHandler;

    public void onCreate(Bundle instanceBundle) {
        ...
        if(Integer.parseInt(VERSION.SDK) < 5)
            touchHandler = new SingleTouchHandler(surfaceView);
        else
            touchHandler = new MultiTouchHandler(surfaceView);
    }

    public boolean isTouchDown(int pointer) {
        return touchHandler.isTouchDown(pointer);
    }

    public int getTouchX(int pointer) {
        return (int)(touchHandler.getTouchX(pointer) /
                (float)surfaceView.getWidth() *
                offscreenSurface.getWidth());
    }

    public int getTouchY(int pointer) {
        return (int)(touchHandler.getTouchY(pointer) /
                (float)surfaceView.getHeight() *
                offscreenSurface.getHeight());
    }...
}
```

4 – Implementing the Input Methods

```
public class TouchExample extends Game {
    public Screen createStartScreen() {
        return new TouchScreen(this);
    }

    class TouchScreen extends Screen {
        Bitmap bob;

        public TouchScreen(Game game) {
            super(game);
            bob = game.loadBitmap("bob.png");
        }

        public void update(float deltaTime) {
            game.clearFramebuffer(Color.BLUE);

            for(int pointer = 0; pointer < 5; pointer++) {
                if(game.isTouchDown(pointer)) {
                    game.drawBitmap(bob, game.getTouchX(pointer), game.getTouchY(pointer));
                }
            }
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

Badlogic Games

4 – Implementing the Input Methods

- The accelerometer gives us the acceleration in meters per second on 3 axes.
- They range from -10 to 10 (acceleration the earth‘s gravity field excerts on a body)

Badlogic Games

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener,
 SensorEventListener {
   ...
   private float[] accelerometer = new float[3];

   public void onCreate(Bundle instanceBundle) {
      ...
      SensorManager manager = (SensorManager).getSystemService(Context.SENSOR_SERVICE);
      if (manager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() != 0) {
         Sensor accelerometer = manager.getSensorList(Sensor.TYPE_ACCELEROMETER).get(0);
         manager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_GAME);
      }
   }

   public float[] getAccelerometer() { return accelerometer; }

   public void onAccuracyChanged(Sensor sensor, int accuracy) { }

   public void onSensorChanged(SensorEvent event) {
      System.arraycopy(event.values, 0, accelerometer, 0, 3);
   }

   public void onPause() {
      ...
      if(isFinishing()) ((SensorManager)getSystemService(Context.SENSOR_SERVICE)).unregisterListener(this);
   }
}
```

4 – Implementing the Input Methods

```
public class AccelerometerExample extends Game {
    public Screen createStartScreen() {
        return new AccelerometerScreen(this);
    }

    class AccelerometerScreen extends Screen {
        Bitmap bob;
        public AccelerometerScreen(Game game) {
            super(game);
            bob = game.loadBitmap("bob.png");
        }

        public void update(float deltaTime) {
            float x = -game.getAccelerometer()[0];
            float y = game.getAccelerometer()[1];
            x = (x / 10) * game.getFramebufferWidth() / 2 + game.getFramebufferWidth() / 2;
            y = (y / 10) * game.getFramebufferHeight() / 2 + game.getFramebufferHeight() / 2;
            game.clearFramebuffer(Color.BLUE);
            game.drawBitmap(bob, (int)x - 64, (int)y - 64);
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Implementing the Input Methods

- We have all input polling methods in place.
- Event-based input makes sense for key events and the touch screen.
- Problem: we need to pass on instances of KeyEvent and TouchEvent to our main loop thread from the UI thread.
- We could do this with a list for each event type.
- But we'd create new events all the time!
- That would make the GC angry.
- Solution: Pooling KeyEvent/TouchEvent instances!

4 – Implementing the Input Methods

```
public abstract class Pool<T> {
    private List<T> items = new ArrayList<T>();

    protected abstract T newItem();

    public T obtain() {
        if(items.size() == 0) return newItem();
        return items.remove(0);
    }

    public void free(T item) {
        items.add(item);
    }
}
```

4 – Implementing the Input Methods

- Instead of calling new KeyEvent() we call Pool<KeyEvent>.obtain().
- We need to implement the Pool.newItem() method of course.
- Pool for TouchEvent instances analogous.

```
public class KeyEventPool extends
Pool<KeyEvent> {
    protected KeyEvent newItem() {
        return new KeyEvent();
    }
}
```

4 – Implementing the Input Methods

- **Our strategy for each event type:**
  - We have a Pool to create and recycle event instances.
  - We have a buffer that gets filled with events on the UI thread.
  - We have a second buffer that we move the UI thread buffer contents to on the main loop thread.
  - The Game.getXXXEvents() methods return this second buffer to the current Screen.
  - Once the Screen is updated we can recycle all the events in the second buffer.
  - Goto 1 :)
- **Let‘s implement this for key events.**

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener,
 SensorEventListener {
   ...
   private KeyEventPool keyEventPool = new KeyEventPool();
   private List<com.badlogic.agd.KeyEvent> keyEvents = new ArrayList<com.badlogic.agd.KeyEvent>();
   private List<com.badlogic.agd.KeyEvent> keyEventBuffer = new ArrayList<com.badlogic.agd.KeyEvent>();

   private void fillEvents() {
      synchronized (keyEventBuffer) {
         for(int i = 0; i < keyEventBuffer.size(); i++) { keyEvents.add(keyEventBuffer.get(i)); }
         keyEventBuffer.clear();
      }
   }

   private void freeEvents() {
      synchronized(keyEventBuffer) {
         for(int i = 0; i < keyEvents.size(); i++) { keyEventPool.free(keyEvents.get(i));
         keyEvents.clear();
      }
   }
}
```

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener,
 SensorEventListener {
   ...
    public boolean onKey(View v, int keyCode, KeyEvent event) {
      if(event.getAction() == KeyEvent.ACTION_DOWN) {
         pressedKeys[keyCode] = true;
         synchronized(keyEventBuffer) {
            com.badlogic.agd.KeyEvent keyEvent = keyEventPool.obtain();
            keyEvent.type = KeyEventType.Down;
            keyEvent.keyCode = keyCode;
            keyEventBuffer.add(keyEvent);
         }
      }

      if(event.getAction() == KeyEvent.ACTION_UP) {
         pressedKeys[keyCode] = false;
         synchronized(keyEventBuffer) {
            com.badlogic.agd.KeyEvent keyEvent = keyEventPool.obtain();
            keyEvent.type = KeyEventType.Up;
            keyEvent.keyCode = keyCode;
            keyEvent.character = (char)event.getUnicodeChar();
            keyEventBuffer.add(keyEvent);
         }
      }
      return false;
   }
}
```

Badlogic Games

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener,
 SensorEventListener {
   ...
    public boolean onKey(View v, int keyCode, KeyEvent event) {
      if(event.getAction() == KeyEvent.ACTION_DOWN) {
         pressedKeys[keyCode] = true;
         synchronized(keyEventBuffer) {
            com.badlogic.agd.KeyEvent keyEvent = keyEventPool.obtain();
            keyEvent.type = KeyEventType.Down;
            keyEvent.keyCode = keyCode;
            keyEventBuffer.add(keyEvent);
         }
      }

      if(event.getAction() == KeyEvent.ACTION_UP) {
         pressedKeys[keyCode] = false;
         synchronized(keyEventBuffer) {
            com.badlogic.agd.KeyEvent keyEvent = keyEventPool.obtain();
            keyEvent.type = KeyEventType.Up;
            keyEvent.keyCode = keyCode;
            keyEvent.character = (char)event.getUnicodeChar();
            keyEventBuffer.add(keyEvent);
         }
      }
      return false;
   }
}
```

Badlogic Games

4 – Implementing the Input Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener,
 SensorEventListener {
    ...
    public void run() {
    ...
    while(true) {
        ...
        if(state == State.Running) {
            ...
            fillEvents();
            if(screen != null) screen.update(0);
            freeEvents();
            ...
        }
    }
}
```

4 – Implementing the Input Methods

```
public class KeyEventExample extends Game {
    public Screen createStartScreen() {
        return new KeyEventScreen(this);
    }

    class KeyEventScreen extends Screen {
        public KeyEventScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) {
            List<KeyEvent> keyEvents = game.getKeyEvents();
            for(int i = 0; i < keyEvents.size(); i++) {
                KeyEvent event = keyEvents.get(i);
                Log.d("KeyEventScreen", "key: " +
                                        event.type + ", " +
                                        event.keyCode + ", " +
                                        event.character);
            }
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Implementing the Input Methods

- We do exactly the same for touch events!
- The TouchEvent buffer is filled on the UI thread in the TouchHandler.
- On the main loop thread we copy the events from the UI buffer to the main loop buffer and provide it to the Screen.
- I spare you the detailed code.
- Here‘s an example

4 – Implementing the Input Methods

```
public class TouchEventExample extends Game {
    public Screen createStartScreen() {
        return new TouchEventScreen(this);
    }

    class TouchEventScreen extends Screen {
        public TouchEventScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) {
            List<TouchEvent> touchEvents = game.getTouchEvents();
            for(int i = 0; i < touchEvents.size(); i++) {
                TouchEvent event = touchEvents.get(i);
                Log.d("TouchEventScreen", "key: " +
                                event.type + ", " +
                                event.x + ", " +
                                event.y + ", " +
                                event.pointer);
            }
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

Badlogic Games

4 – Implementing the Audio Methods

- We have two methods to load Sound and Music instances from asset files.
- For Sounds we use the SoundPool class of Android.
- For Music we wrap the Android MediaPlayer class.
- Sound and Music instances need to be disposed if they are no longer used to free up resources.
- (The same is true for Bitmaps btw.)

4 – Implementing the Audio Methods

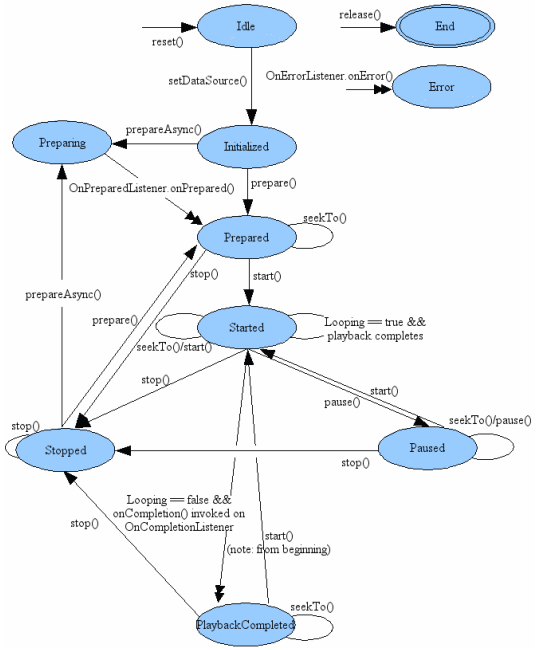
```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...
    private SoundPool soundPool;

    public void onCreate(Bundle instanceBundle) {
        ...
        setVolumeControlStream(AudioManager.STREAM_MUSIC);
        this.soundPool = new SoundPool(20, AudioManager.STREAM_MUSIC, 0);
    }

    public Sound loadSound(String fileName) {
        try {
            AssetFileDescriptor assetDescriptor = getAssets().openFd(fileName);
            int soundId = soundPool.load(assetDescriptor, 0);
            return new Sound(soundPool, soundId);
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load sound '" + fileName + "'");
        }
    }
    public void onPause() {
        ...
        if(isFinishing()) soundPool.release();
    }...
}
```

4 – Implementing the Audio Methods

```
public class SoundExample extends Game {

    public Screen createStartScreen() {
        return new SoundScreen(this);
    }

    class SoundScreen extends Screen {
        Sound sound;

        public SoundScreen(Game game) {
            super(game);
            sound = game.loadSound("explosion.ogg");
        }

        public void update(float deltaTime) {
            if(game.isTouchDown(0)) sound.play(1);
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { sound.dispose(); }
    }
}
```

4 – Implementing the Audio Methods

- For the Music class we use Android‘s MediaPlayer.
- Why wrap it? That‘s why:



Badlogic Games

4 – Implementing the Audio Methods

```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...
    public Music loadMusic(String fileName) {
        try {
            AssetFileDescriptor assetDescriptor = getAssets().openFd(fileName);
            return new Music(assetDescriptor);
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load music '" + fileName + "'");
        }
    }
    ...
}
```

4 – Implementing the Audio Methods

```
public class MusicExample extends Game {
    public Screen createStartScreen() {
        return new MusicScreen(this);
    }

    class MusicScreen extends Screen {
        Music music;
        boolean isPlaying = false;

        public MusicScreen(Game game) {
            super(game);
            music = game.loadMusic("music.ogg");
            music.setLooping(true); music.play();
            isPlaying = true;
        }

        public void update(float deltaTime) {
            if(game.isTouchDown(0)) {
                if(music.isPlaying()) {
                    music.pause(); isPlaying = false;
                }
                else {
                    music.play(); isPlaying = true;
                }
            }
        }

        public void pause() { music.pause(); }
        public void resume() { if(isPlaying) music.play(); }
        public void dispose() { music.dispose(); }
    }
}
```

4 – Implementing the Audio Methods

- We need to make sure to pause/dispose the music when the application is paused/disposed.
- Otherwise the MediaPlayer will happily continue to play.

4 - File I/O

- **We can access assets like this:**
  - InputStream in = Game.getAssets().open("filename");
- **Assets are read only!**

4 – File I/O

- **We can read & write files on the external storage.**
- **The storage must be mounted:**
  - Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED);
- **Path to the external storage:**
  - File dir = Environment.getExternalStorageDirectory();
- **From here on we can use standard Java I/O.**

4 – File I/O

- **For small data items like highscores we can use Android‘s SharedPreferences.**
- **It‘s basically a named HashMap you can put different primitive types in.**
- **Creation:**
  - Game.getSharedPreferences(„name“, 0);
- **Fetching values:**
  - SharedPreferences.getXXX(String key, XXX default);
- **Writing values is a little more involved.**

4 – File I/O

- **Need to get an Editor instance from the SharedPreferences:**
  - Editor editor = preferences.edit();
- **Put values in the Editor:**
  - editor.putString(„key“, „value“);
- **Commit the changes for future use:**
  - editor.commit();
- **We can also clear everything or a single item:**
  - editor.clear();
  - Editor.remove(„key“);

4 – Framerate & Delta Time

- The framerate is the number of frames we render/update per second.
- Let‘s add a new method to Game called Game.getFrameRate()
- It will return the (average) number of frames.
- We simply count how often the main loop iterates in a second.
- We can use this method to see how well our game performs.

4 – Framerate & Delta Time

```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...
    private int framesPerSecond = 0;

    public void run() {
        int frames = 0;
        long startTime = System.nanoTime();
        while(true) {
            ...
            frames++;
            if(System.nanoTime() – startTime > 1000000000) {
                framesPerSecond = frames;
                frames = 0;
                startTime = System.nanoTime();
            }
        }
    }

    public int getFramerate() {
        return framesPerSecond;
    }
}
```

4 – Framerate & Delta Time

```
public class FramerateExample extends Game {
    public Screen createStartScreen() {
        return new FramerateScreen(this);
    }

    class FramerateScreen extends Screen {
        public FramerateScreen(Game game) {
            super(game);
        }

        public void update(float deltaTime) {
            Log.d("FramerateScreen", "fps: " + game.getFramerate());
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

4 – Framerate & Delta Time

- Look at the code below.
- What would happen on a device that can do 30 FPS?
- What would happen on a device that can do 60 FPS?

```
public class MoverScreen extends Screen {
    Bitmap bitmap;
    float x = 0;

    public void update(float deltaTime) {
        x += 1;
        game.drawBitmap(bitmap, x, 0);
    }
}
```

The Image would move faster on the 60 FPS device!

4 – Framerate & Delta Time

- **Solution: Time-based Movement**
- **A.k.a. Framerate-Independent Movement**
- **Objects have**
  - Position expressed as 2D coordinates (x,y)
  - Velocity expressed as 2D vector (vx,vy)
- **Unit can be meter/pixel/etc. for position.**
- **Unit can be meter/pixel/etc. per second for velocity.**
- **Update of position:**
  - newx = oldx + vx * deltaTime;
  - newx = oldy + vy * deltaTime;

4 – Framerate & Delta Time

- Delta time is the timespan passed since the last update.
- We just need to keep track how much time has passed between the current frame and the last frame.
- At 60 FPS we have a delta time of 1 / 60 ~ 0.016 seconds.
- At 30 FPS we have a delta time of 1 / 30 ~ 0.032 seconds.

4 – Framerate & Delta Time

```
public abstract class Game extends Activity implements Runnable, OnKeyListener {
    ...

    public void run() {
        ...
        long lastTime = System.nanoTime();
        while(true) {
            ...
            long currTime = System.nanoTime();
             if(screen != null) screen.update((currTime - lastTime) / 1000000000.0f);
             lastTime = currTime;
            ...
        }
    }

    public int getFramerate() {
        return framesPerSecond;
    }
}
```

4 – Framerate & Delta Time

```
public class DeltaTimeExample extends Game {
    public Screen createStartScreen() {
        return new DeltaTimeScreen(this);
    }

    class DeltaTimeScreen extends Screen {
        Bitmap bob;
        float x = 0;

        public DeltaTimeScreen(Game game) {
            super(game);
            bob = game.loadBitmap("bob32.png");
        }

        public void update(float deltaTime) {
            x = x + 10 * deltaTime;
            game.clearFramebuffer(Color.BLUE);
            game.drawBitmap(bob, (int)x, 10);
        }

        public void pause() { }
        public void resume() { }
        public void dispose() { }
    }
}
```

Agenda

1. ~~Android – A Gaming Platform~~
2. ~~Game Design 101~~
3. ~~Game Programming 101~~
4. ~~Android for Game Developers~~
5. ~~Lunch Break (Ohm Nom Nom Nom)~~
6. Droidanoid

5 - Droidanoid

A Breakout Clone!

Badlogic Games

5 – Core Game Mechanics

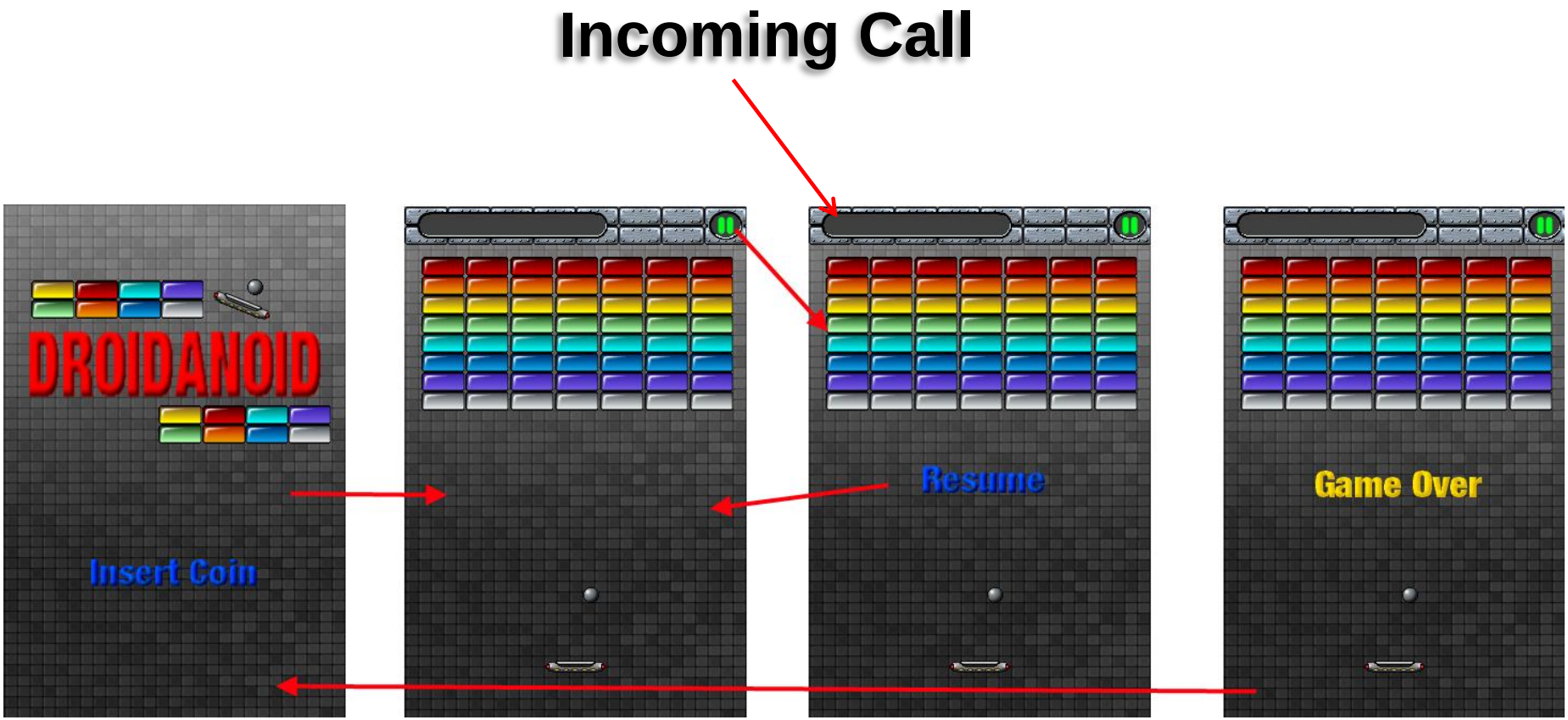
- Paddle on the bottom can move left/right based on accelerometer.
- Ball has a direction in which it moves.
- Ball bounces of walls, left, right and top.
- If ball hits a block, the block is destroyed and the ball changes direction
- Player is awarded points based on color of block (10, 15, 20, etc.)
- If no blocks left, level is filled with new blocks, ball is reset.
- If ball gets to bottom of screen, the game is over.

5 – Story & Art Style

- Rescue the Princess… wut?
- Evil forces invade the land of… nope, that doesn‘t work either.
- Who gives a damn!
- You have a paddle and a ball, hit the darn blocks to releave anger!
- Art style is 8-bit retro, cause we like it (and we can find assets on the web).
- Note: do not use stuff you find on the web before clearing permissions!

Badlogic Games

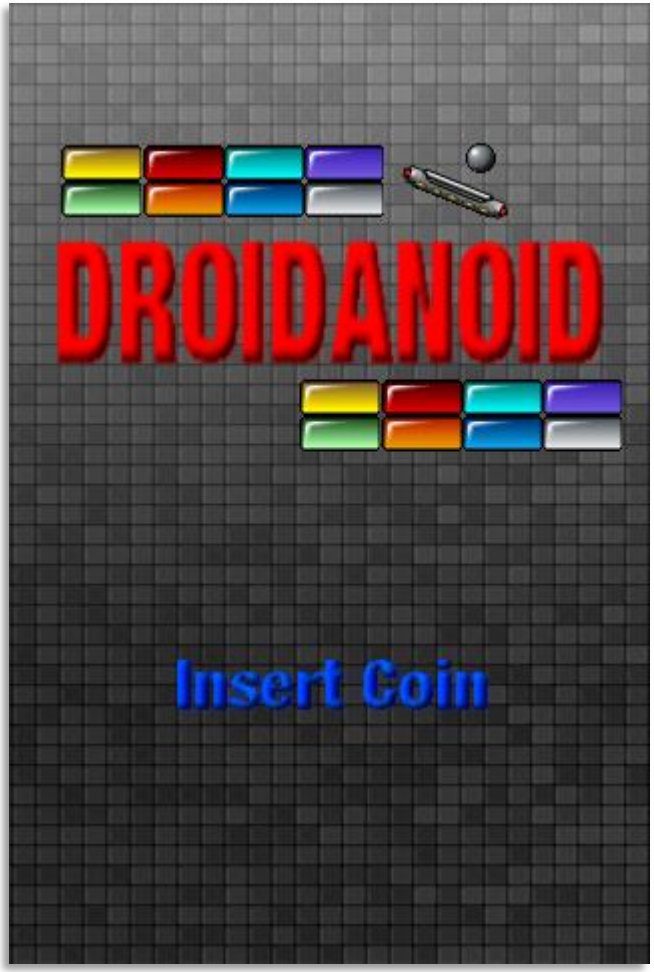
5 – Screens & Transitions



Badlogic Games

5 – Main Menu Screen

- Shows a static background image with the logo.
- „Insert Coin“ label is blinking with a 0.5 seconds interval.
- A touch of the screen will trigger a transition to the game screen.



Badlogic Games

5 – Game Screen

- **Shows a static background image with the top bar for score display and pause button.**
- **The blocks are individual objects.**
- **Same for ball and paddle.**
- **Has state:**
  - Running
  - Paused
  - GameOver

5 – Game Screen

- In the running state the action happens.
- Game world is simulated & rendered.
- Changes state to game over if ball leaves screen on the bottom edge.
- Changes state to paused if home or pause button is pressed, or call is incoming.

Badlogic Games

5 – Game Screen

- In the game over state we display an additional game over label.
- The game world is still rendered but not simulated
- Waiting for user touching the screen which will trigger transition to main menu.
- Note that incoming call or home button can put the game to sleep!
- When player returns the screen is still in the game over state.

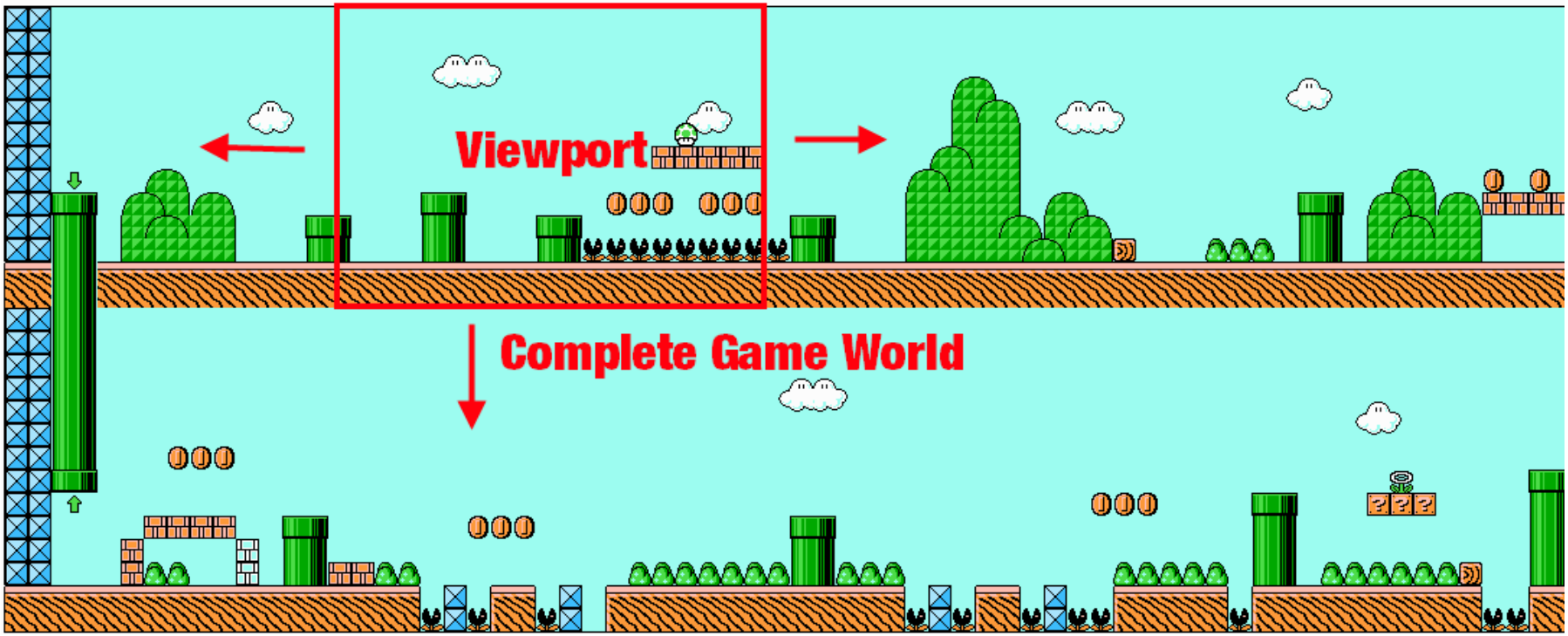
5 – Defining the Game World

- Dimensions & Sizes of game world should be designed before creating assets!
- We already know our assets so it‘s a bit „hen & egg“.
- Let‘s pretend we don‘t have the mock-ups yet.

5 – Defining the Game World

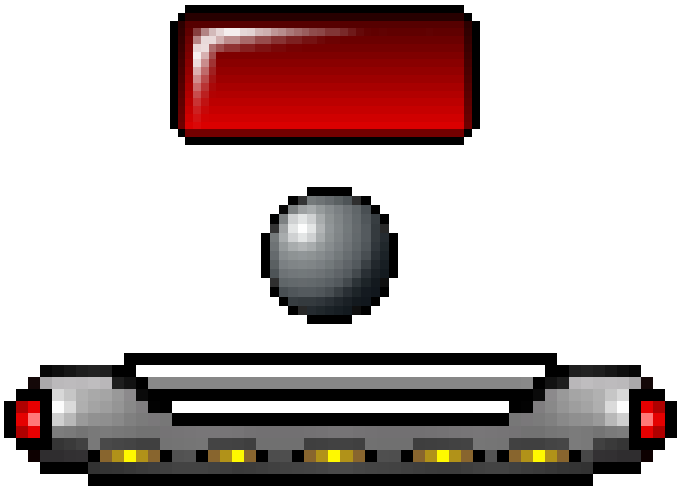
- Define target resolution (320x480).
- Define the portion available for displaying the game world. (0,36) to (319,479) pixels == viewport.
- In Droidanoid we see all of the world at once.
- In other games game world portion on screen acts as viewport into our game world that we can move around.

5 – Defining the Game World



5 – Defining the Game World

- **We have 3 types of objects:**
  - Blocks
  - Ball
  - Paddle
- **How big should they be?**
- **What units do we use?**



5 – Defining the Game World

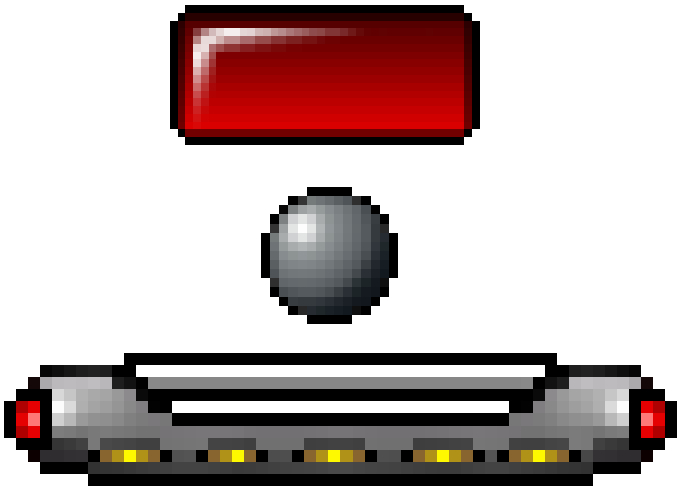
- We use pixels for convenience.
- Note: we don‘t have to!
- We could also use meters to specify the sizes, positions and velocities.
- Decouple graphical representation from logical representation.

5 – Defining the Game World

- **Scale factor to translate from logical units (e.g. meters) to pixels.**
- **In our hypothecal case that factor is 10.**
- **Logical -> Pixel**
  - (23, 4.78) m * 10 = (230, 47.8) px
- **Pixel -> Logical**
  - (230, 47.8) px / 10 = (23, 4.78) m
- **We model & simulate in logical units, render in pixel units!**

5 – Defining the Game World

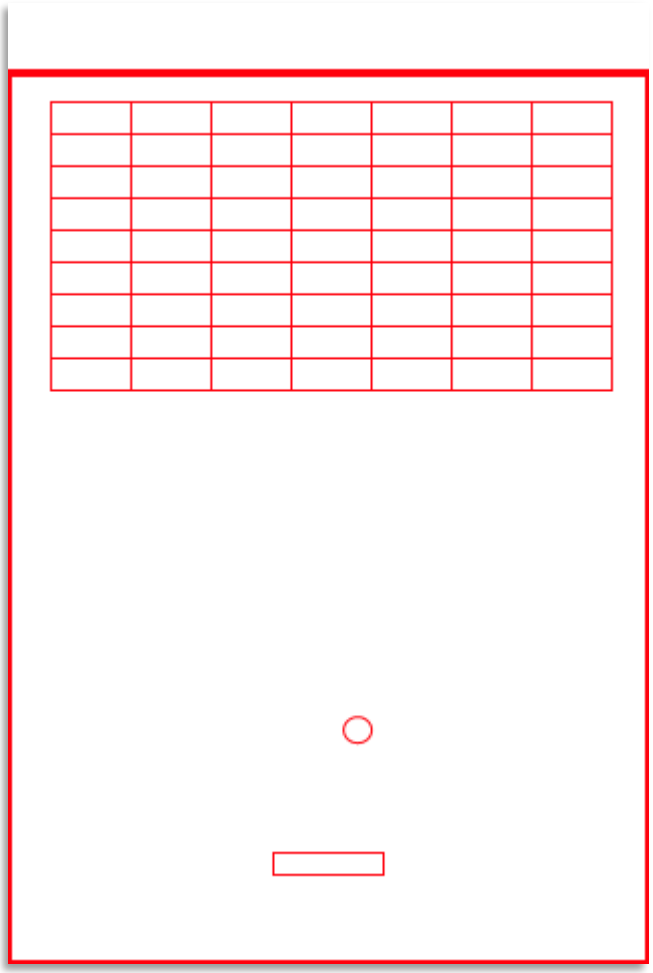
- How big should the objects be then!?
- Trial & Error + Gut Feeling.
- Sizes relative to each other must make sense (logical units).
- Sizes on screen must make sense (pixel units).
- „Super Mario should take up 1/5th of the viewport“



5 – Defining the Game World

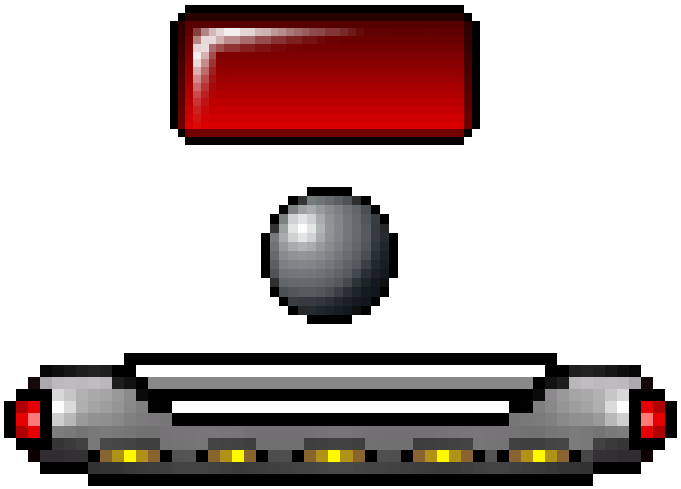
1. Define target resolution (320x480).
2. Define viewport in pixels (320x444).
3. Define logical to pixel scale factor(1 meter -> 10 pixels).
4. Calculate viewport size in logical units (32x44.4 meters).
5. Define object sizes given logical viewport (block: 4x1.8m, ball: 1.5x1.5m, paddle: 5.6x1.1m)
6. Calculate pixels sizes from logical sizes (times 10, block: 40x18px, ball: 15x15px, paddle: 56x11px).
7. Create Assets!

5 - Defining the Game World



5 – Defining the Game World

- **Surprise: we use pixels as logical units! _/_**
- **Why do i put you through this?**
- **Seperation of graphics and logical view of game world is important!**
  - Physics libraries force you to work in meters.
  - Scaling graphics doesn‘t change logical representation!
- **Similar to DIP in UI programming.**

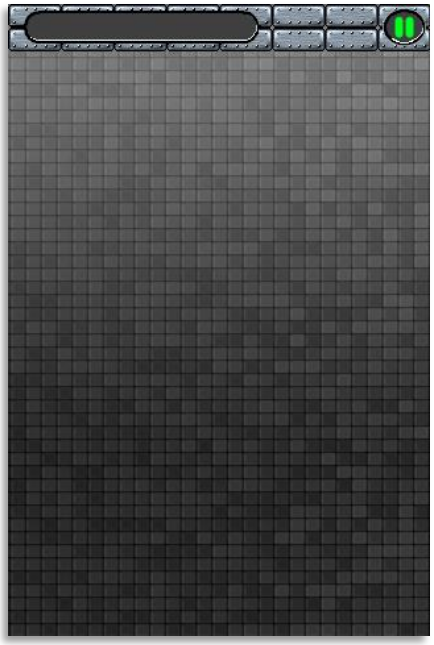


5 – Defining the Game World

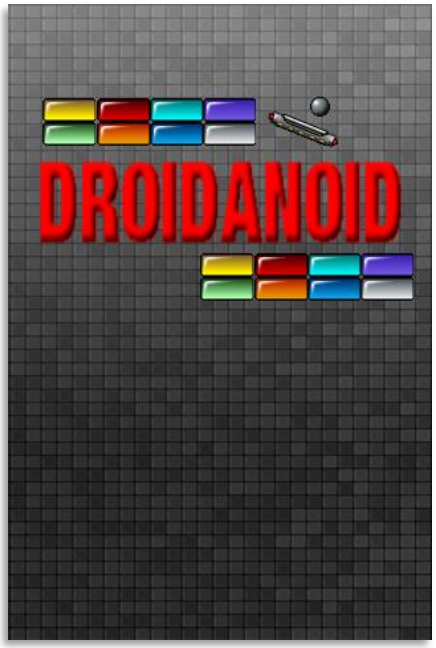
- **Decoupling lends itself to (modified) Model-View-Controller pattern.**
- **Model: dumb classes for**
  - Ball.
  - Block.
  - Paddle.
- **Controller:**
  - simulates world, using & updating model.
  - Processes and applies user input.
- **View: renders the current model state!**

5 – Creating the Assets

blocks.png



background.png



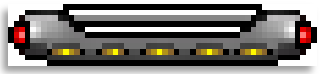
mainmenu.png

ball.png

insertcoin.png

resume.png

gameover.png



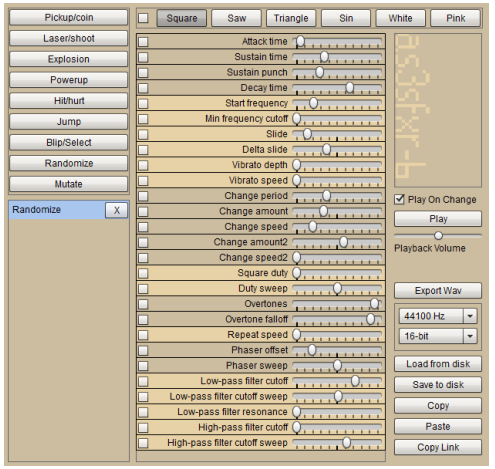
paddle.png

Sprites by David Gervais, free for educational/non-commercial use
http://forum.thegamecreators.com/?m=forum_view&t=67876&b=1
Gray Background from (fair use).
http://www.consolespot.net/graphics-design/15371-homebrew-icon-thread.html

Badlogic Games

5 – Creating the Assets

- For sound effects we use AS3SFXR
  http://ded.increpare.com/~locus/as3sfxr-b/
- 8-bit synth.
- Fitting music from http://www.freemusicarchive.org
- Obey the licenses!



Badlogic Games

5 – Implementing Droidanoid

- Let‘s start simple by implementing the Droidanoid class.
- It‘s our main entry point.
- Sets the first screen shown to the player (MainMenuScreen).

```
public class Droidanoid extends Game {
    public Screen createStartScreen() {
        return new MainMenuScreen(this);
    }
}
```

5 – Implementing MainMenuScreen

- **Screen implementation showing**
- **The main menu background image**
- **A blinking „Insert Coin“ label.**
  - Label should be shown for 0.5 seconds, then be invisble for 0.5 seconds etc.
- **We use the delta time to figure out how much time passed for blinking.**
- **In case screen is touched we trigger transition to GameScreen.**

5 – Implementing the MainMenuScreen

```
public class MainMenuScreen extends Screen {
    Bitmap mainmenu, insertCoin, ball;
    float passedTime = 0;

    public MainMenuScreen(Game game) {
        super(game);
        mainmenu = game.loadBitmap("mainmenu.png");
        insertCoin = game.loadBitmap("insertcoin.png");
    }

    long startTime = System.nanoTime();
    public void update(float deltaTime) {
        if(game.isTouchDown(0)) { game.setScreen(new GameScreen(game)); return; }

        passedTime += deltaTime;
        game.drawBitmap(mainmenu, 0, 0);
        if((passedTime - (int)passedTime) > 0.5f)
            game.drawBitmap(insertCoin, 160 - insertCoin.getWidth() / 2, 360);
    }

    public void dispose() {}
    public void pause() { }
    public void resume() { }
}
```

5 – Implementing GameScreen.

- **Has multiple tasks to handle**
  - Proper pause/resume.
  - Render UI (score, background, pause button etc.).
  - Simulate the game world.
  - Render the game world.
- **Let‘s implement the first two items.**

5 – Implementing GameScreen

- **We have 3 states:**
  - Running, Paused, Game Over.
- **We‘ll encode those with an enum.**
- **We need to display the background and the apropriate label („Resume“, „Game Over“) depending on state.**

5 – Implementing GameScreen

```
public class GameScreen extends Screen {
    enum State { Paused, Running, GameOver }

    Bitmap background, resume, gameOver;
    State state = State.Running;

    public GameScreen(Game game) {
        super(game);
        background = game.loadBitmap("background.png");
        resume = game.loadBitmap("resume.png");
        gameOver = game.loadBitmap("gameover.png");
    }
   …
```

5 – Implementing GameScreen

```
public class GameScreen extends Screen {
    ...
    public void update(float deltaTime) {
        if(state == State.Paused && game.getTouchEvents().size() > 0) {
            state = State.Running;
        }
        if(state == State.GameOver && game.getTouchEvents().size() > 0) {
            game.setScreen(new MainMenuScreen(game));
            return;
        }
        if(state == State.Running && game.getTouchY(0) < 32 && game.getTouchX(0) > 320 - 32) {
            state = State.Paused;
            return;
        }

        game.drawBitmap(background, 0, 0);
        if(state == State.Paused)
            game.drawBitmap(resume, 160 - resume.getWidth() / 2, 240 - resume.getHeight() / 2);
        if(state == State.GameOver)
            game.drawBitmap(gameOver, 160 - gameOver.getWidth() / 2, 240 - gameOver.getHeight() / 2);
    }

    public void pause() {
        if(state == State.Running) state = State.Paused;
    }
```

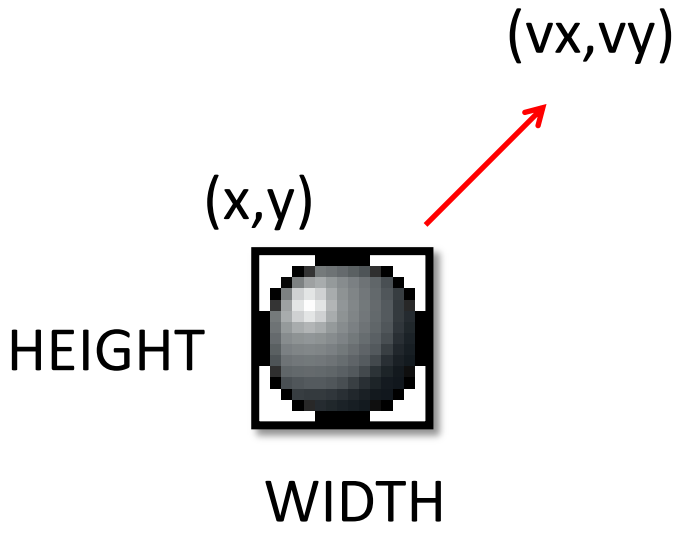
5 – Implementing GameScreen

- Time to implement the model, view and controller of our world.
- The model is composed of classes for each object (Ball, Paddle, Block).
- The controller keeps track of objects and simulates the world. Class name World.
- The view is a separate class we call WorldRenderer. It takes the model and renders its current state (via Bitmaps and Game).
- We‘ll develop those three things iteratively.

5 – Implementing GameScreen

- Ball has a position (x,y) and a velocity (vx, vy).
- Position equals upper left corner of ball, given in pixels
- Velocity is given in pixels per second (time-based movement!).
- Both attributes are actually 2D vectors!
- Velocity also encodes direction in x and y!
- Width and height are constants.

```
public class Ball {
    public static float WIDTH = 15;
    public static float HEIGHT = 15;
    float x = 160, y = 240;
    float vx = 170, vy = -170;
}
```



5 – Implementing GameScreen

- World class stores model (Ball only for now)
- Responsible for updating and colliding objects (delta time!)
- Let‘s make the Ball bounce of the walls.

```
public class World {
    public static float MIN_X = 0, MIN_Y = 32;
    public static float MAX_X = 319, MAX_Y = 479;
    Ball ball = new Ball();

    public void update(float deltaTime) {
        ball.x += ball.vx * deltaTime;
        ball.y += ball.vy * deltaTime;
        if(ball.x < MIN_X) { ball.vx = -ball.vx; ball.x = MIN_X; }
        if(ball.x - Ball.WIDTH > MAX_X) { ball.vx = -ball.vx; ball.x = MAX_X – Ball.WIDTH; }
        if(ball.y < MIN_Y) { ball.vy = -ball.vy; ball.y = MIN_Y; }
        if(ball.y - Ball.HEIGHT > MAX_Y) { ball.vy = -ball.vy; ball.y = MAX_Y – Ball.HEIGHT; }
    }
}
```

5 – Implementing GameScreen

- WorldRenderer has reference to World to get model state.
- Game reference to load Bitmaps and draw objects.

```
public class WorldRenderer {
    Game game;
    World world;
    Bitmap ballImage;

    public WorldRenderer(Game game, World world) {
        this.game = game;
        this.world = world;
        this.ballImage = game.loadBitmap("ball.png");
    }

    public void render() {
        game.drawBitmap(ballImage, (int)world.ball.x, (int)world.ball.y);
    }
}
```

5 – Implementing GameScreen

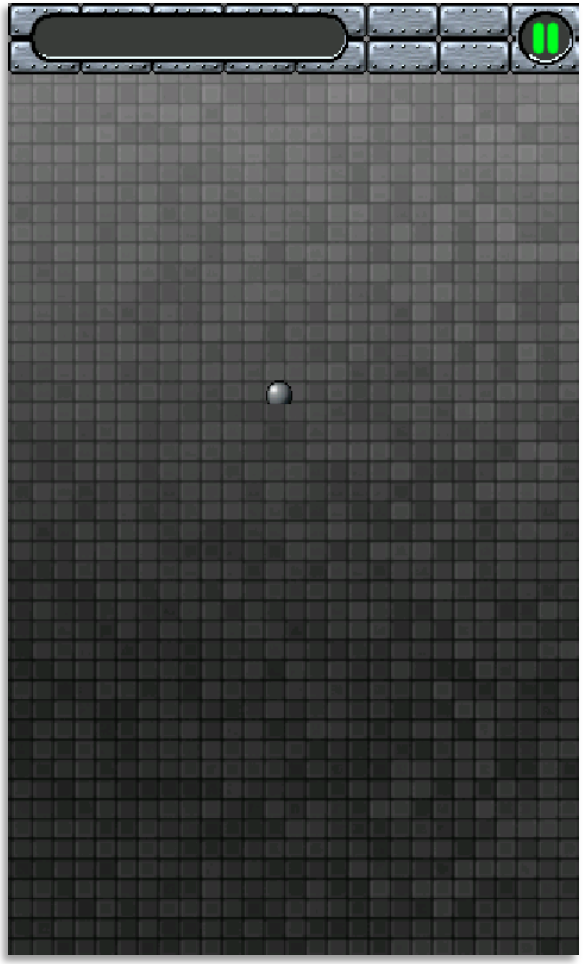
- We have to add the World & WorldRenderer to GameScreen.
- Call methods at appropriate place.

```
public class GameScreen extends Screen {
    …
    World world;
    WorldRenderer renderer;

    public GameScreen(Game game) {
        …
        world = new World();
        renderer = new WorldRenderer(game, world);
    }

    public void update(float deltaTime) {
        …
        if( (state == State.Running) world.update(deltaTime);

        game.drawBitmap(background, 0, 0);
        renderer.render();
        …
    }
```

Badlogic Games

5 – Implementing GameScreen

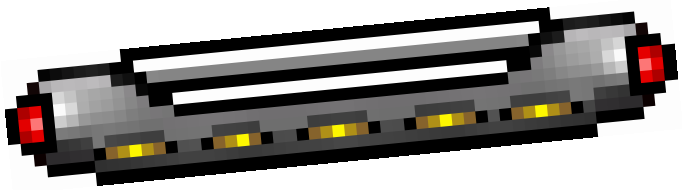
Yay!



5 – Implementing GameScreen

- Next up is the Paddle class.
- Pretty much like Ball, dimensions differ.
- Y-coordinate is fixed, we can only move in x!
- Velocity comes from outside -> Acceleromter!
- World is responsible for movement.

```
public class Paddle {
    public static float WIDTH = 56;
    public static float HEIGHT = 11;
    float x = 160 - WIDTH / 2, y = World.MAX_Y - 40;
}
```



Badlogic Games

5 – Implementing GameScreen

- Paddle is controlled by accelerometer.
- Reading of x-axis is velocity of paddle!
- We have to pipe accelerometer value to World so it can update Paddle.
- AccelX in range [-10, 10], have to negate and scale it a little -> tuning!

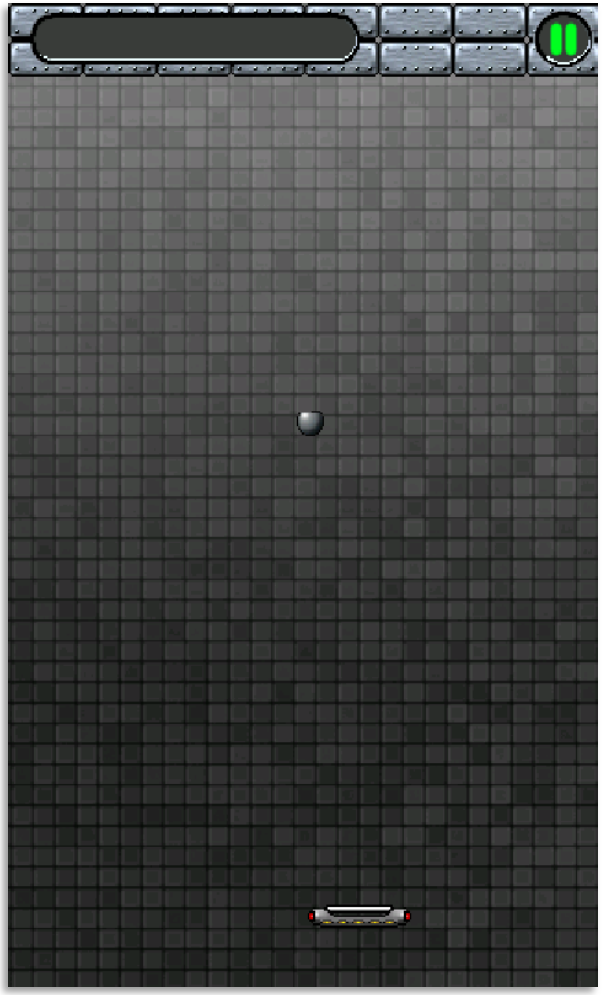
```
public class World {
    ...
    Paddle paddle = new Paddle();

    public void update(float deltaTime, float accelX) {
        ...
        paddle.x += -accelX * 50 * deltaTime;
        if(paddle.x < MIN_X) paddle.x = MIN_X;
        if(paddle.x + Paddle.WIDTH > MAX_X) paddle.x = MAX_X -Paddle.WIDTH;
    }
}
```

5 – Implementing GameScreen

```
public class WorldRenderer {
    …
    Bitmap paddleImage;

    public WorldRenderer(Game game, World world) {
        …
        this.paddleImage = game.loadBitmap("paddle.png");
    }

    public void render() {
        …
        game.drawBitmap(paddleImage, (int)world.paddle.x,(int)world.paddle.y);
    }
}
public class GameScreen {
    …
    public void update(float deltaTime) {
        …
 if(state == State.Running) world.update(deltaTime, game.getAccelerometer()[0]);
        …
    }
}
```

5 – Implementing GameScreen

We have paddle movement!



5 – Implementing GameScreen

- Let us collide the ball with the paddle.
- Only need to check top edge of paddle.
- In case of hit, ball changes velocity/position on y-axis.

```
public class World {
    …
    public void update(float deltaTime, float accelX) {
        …
        collideBallPaddle();
    }

    private void collideBallPaddle() {
        if(ball.y > paddle.y) return;
        if(ball.x >= paddle.x &&
           ball.x < paddle.x + Paddle.WIDTH &&
           ball.y + Ball.HEIGHT >= paddle.y) {
            ball.y = paddle.y - Ball.HEIGHT - 2;
            ball.vy = -ball.vy;
        }
    }
}
```

5 – Implementing GameScreen

- Time for the blocks.
- We have 8 rows and 7 columns.
- Each Block is its own object with a position, just like Ball or Paddle.
- Depending on the row a ball has a specific color and generates a specific number of points.
- We thus keep track of a blocks „type“ == row in a graphical sense.

5 – Implementing GameScreen

```
public class Block {
    public static float WIDTH = 40;
    public static float HEIGHT = 18;
    int type;
    float x, y;

    public Block(float x, float y, int type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }
}
```

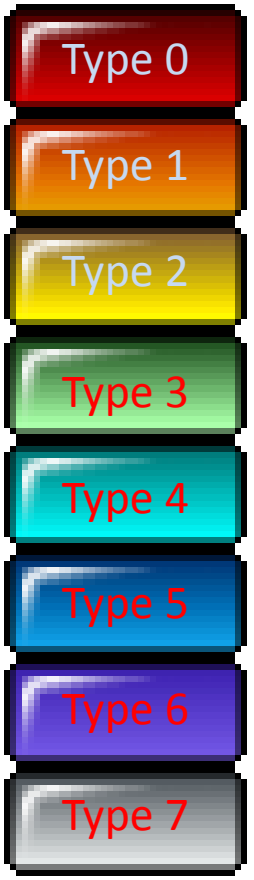
5 – Implementing GameScreen

- World keeps a list of Blocks.
- We have to generate them so they form a grid of 8x7 blocks. Easy!

```
public class World {
    ...
    List<Block> blocks = new ArrayList<Block>();

    public World() {
        generateBlocks();
    }

    private void generateBlocks() {
        blocks.clear();
        for(int y = 60, type = 0; y < 60 + 8 * Block.HEIGHT; y+= Block.HEIGHT, type++) {
            for(int x = 20; x < 320 - Block.WIDTH / 2; x+= Block.WIDTH) {
                blocks.add(new Block(x, y, type));
            }
        }
    }
}
```

5 – Implementing GameScreen

- In the WorldRenderer we do something called image atlasing.
- „blocks.png“ contains not only one block but multiple blocks.
- Depending on the type of block we have to render we chose a sub-image and only render a portion of „blocks.png“
- We use Block.type to select the sub-image.



5 – Implementing GameScreen

```
public class WorldRenderer {
    …
    Bitmap blocksImage;

    public WorldRenderer(Game game, World world ) {
        this.blocksImage = game.loadBitmap("blocks.png");
    }

    public void render() {
        …
        for(int i = 0; i < world.blocks.size(); i++) {
            Block block = world.blocks.get(i);
            game.drawBitmap(blocksImage, (int)block.x, (int)block.y,
                            0, (int)(block.type * Block.HEIGHT),
                            (int)Block.WIDTH, (int)Block.HEIGHT);
        }
    }
}
```

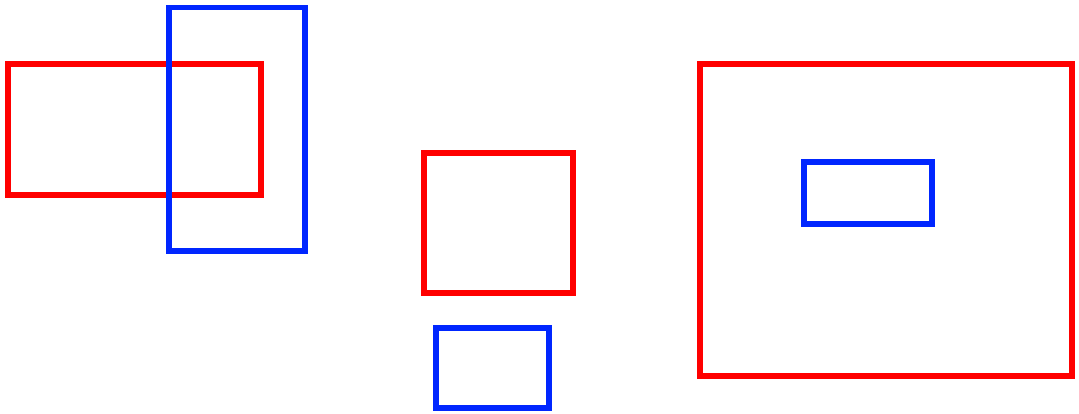
5 – Implementing GameScreen

5 – Implementing GameScreen

- **Plan of Attack for collision between ball & blocks:**
  - Figure out if the ball overlaps with a block.
  - In case it does change the velocity of the ball and remove the block from the world.
  - Corner cases (multiple blocks hit at a time).
- **Let‘s start by checking for overlaps and simply remove a hit block without changing ball velocity.**

5 – Implementing GameScreen

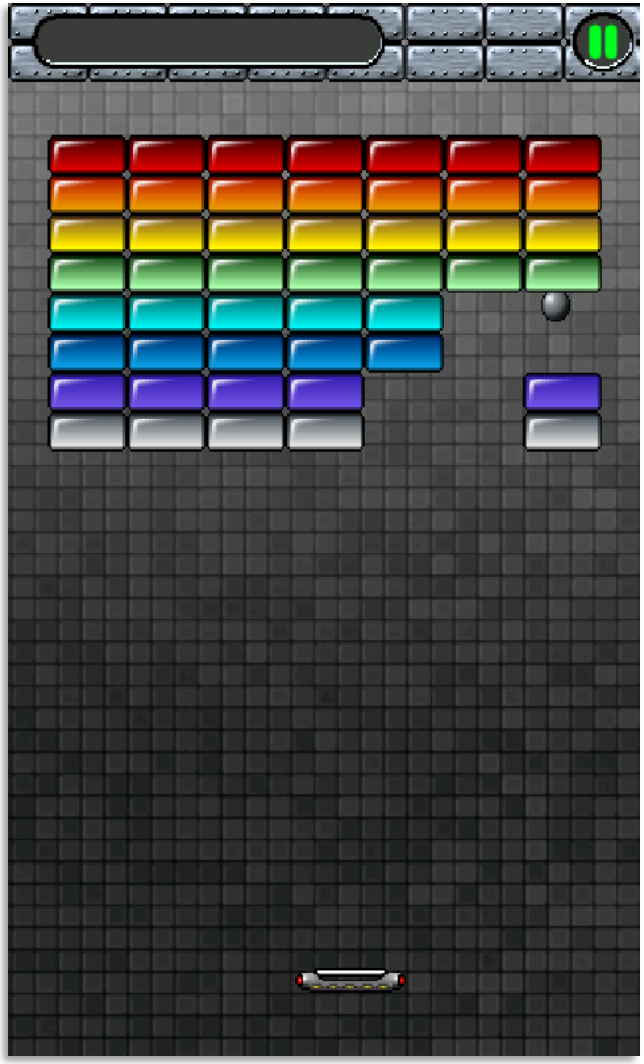
- Ball and blocks are rectangles for simplicity.
- How can rectangles overlap?



5 – Implementing GameScreen

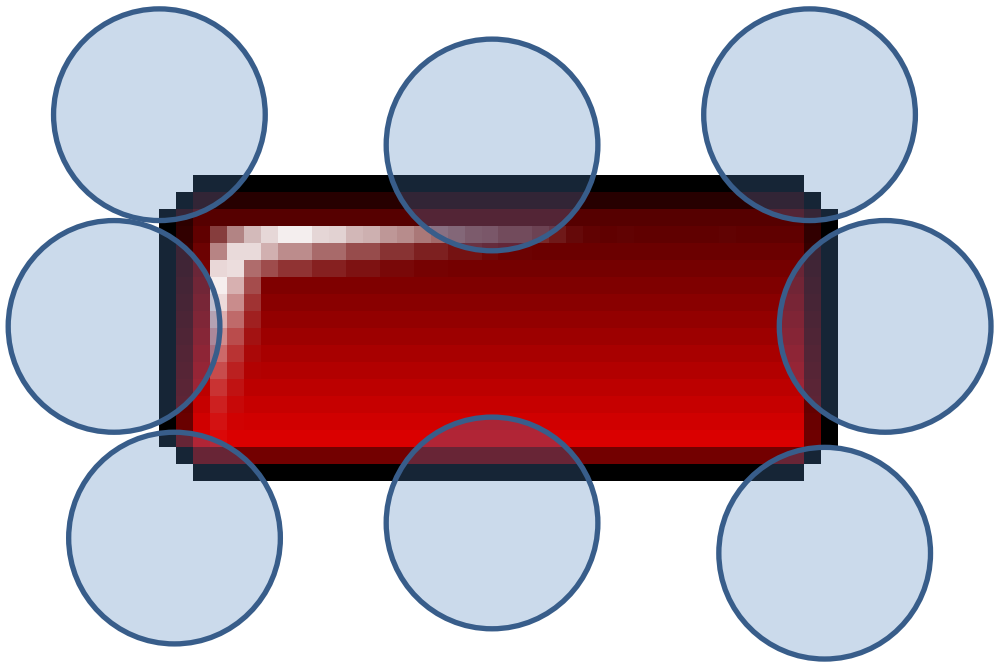
```
Public class World {
    public void update(float deltaTime, float accelX) {
        …
        collideBallBlocks();
    }

    private boolean collideRects(float x, float y, float width, float height,
                                 float x2, float y2, float width2, float height2) {
        if(x < x2 + width2 &&
           x + width  > x2 &&
           y < y2 + height2 &&
           y + width > y2)
            return true;
        return false;
    }

    private void collideBallBlocks() {
        for(int i = 0; i < blocks.size(); i++) {
            Block block = blocks.get(i);
            if(collideRects(ball.x, ball.y, Ball.WIDTH, Ball.HEIGHT,
                            block.x, block.y, Block.WIDTH, Block.HEIGHT)) {
                blocks.remove(i);
                i--;
            }
        }
    }
```

5 – Implementing GameScreen



5 – Implementing GameScreen

- Now we need to let the ball react to a collision.
- We‘ll simply reflect its velocity based on which side of a block it hit.
- Not so clear cut though.



5 – Implementing GameScreen

- We need to check all 4 corners first.
- If that fails we check the four edges.
- We use World.collideRects() using very thin rectangles for edges and corner points.

5 – Implementing GameScreen

```
Public class World {
    ...
    private void collideBallBlocks() {
        for(int i = 0; i < blocks.size(); i++) {
            Block block = blocks.get(i);
            if(collideRects(ball.x, ball.y, Ball.WIDTH, Ball.HEIGHT,
                            block.x, block.y, Block.WIDTH, Block.HEIGHT)) {
                float oldvx = ball.vx;
                float oldvy = ball.vy;
                reflectBall(ball, block);
                ball.x = ball.x - oldvx * deltaTime * 1.01f;
                ball.y = ball.y - oldvy * deltaTime * 1.01f;
            }
        }
    }

    public void reflectBall(Ball ball, Block block) { … };
    ...
}
```

5 – Implementing GameScreen

- **Regenerate blocks if all blocks are destroyed.**

```
Public class World {
    ...
    private void update(float deltaTime, float accelX) {
        if(blocks.size() == 0) generateBlocks();
        ...
    }
...
}
```

5 – Implementing GameScreen

- **Game Over Check (with flag)!**

```
public class World {
    ...
    boolean gameOver = false;

    private void update(float deltaTime, float accelX) {
        if(ball.y > MAX_Y - Ball.HEIGHT) { gameOver = true; return; }
        ...
    }
...
}
 public class GameScreen {
     ...
     private void update(float deltaTime) {
        if(world.gameOver) state = State.GameOver;
        ...
     }
     ...
 }
```

5 – Implementing GameScreen

- There‘s a bug in GameScreen!
- We should really check for a touch up event before we change to the MainMenuScreen.

```
public class GameScreen {
    ...
    private void update(float deltaTime, float accelX) {
        ...
        if(state == State.GameOver) {
            List<TouchEvent> events = game.getTouchEvents();
            for(int i = 0; i < events.size(); i++) {
                if(events.get(i).type == TouchEvent.TouchEventType.Up) {
                    game.setScreen(new MainMenuScreen(game));
                    return;
                }
            }
        }
        ...
    }
...
}
```

Badlogic Games

5 – Implementing GameScreen

- We should also keep track of points!
- Depending on block type we give 10, 9, 8 etc. points.

```
public class World {
    ...
    int points = 0;

    private void collideBallBlocks(float deltaTime) {
        for(int i = 0; i < blocks.size(); i++) {
            Block block = blocks.get(i);
            if(collideRects(ball.x, ball.y, Ball.WIDTH, Ball.HEIGHT,
                            block.x, block.y, Block.WIDTH, Block.HEIGHT)) {
                ...
                points += 10 - block.type;
                break;
            }
        }
    }
}
```

5 – Implementing GameScreen

- **How do we render the score?**
- **Could use image atlas for glyphs (~ characters).**
- **Instead we use Android fonts with Canvas!**
  - Typeface Game.loadFont(String fileName);
  - Game.drawText(Typeface font, String text, int x, int y, int col, int size);

5 – Implementing GameScreen

```
public abstract class Game extends Activity implements Runnable,
OnKeyListener, SensorEventListener {
    Paint paint = new Paint();

    public Typeface loadFont(String fileName) {
        Typeface font = Typeface.createFromAsset(getAssets(), fileName);
        if(font == null)
            throw new RuntimeException("Couldn't load font from asset '" + fileName
+ "'");
        return font;
    }

    public void drawText(Typeface font, String text, int x, int y, int col, int size) {
        paint.setTypeface(font);
        paint.setTextSize(size);
        paint.
        canvas.drawText(text, x, y + size, paint);
    }
}
```

5 – Implementing GameScreen

```
public class GameScreen extends Screen {
    …
    Typeface font;

    public GameScreen(Game game) {
        font = game.loadFont(„font.ttf“);
    }

    public void update(float deltaTime) {
        …
        game.drawText(font, Integer.toString(world.points), 24, 12,
Color.GREEN, 12);
    }
}
```

5 – Implementing Audio

- Sound is missing!
- Let‘s add looping music.

```
public class Droidanoid extends Game {
    Music music;

    public Screen createStartScreen() {
        music = this.loadMusic("music.ogg“);
        return new MainMenuScreen(this);
    }

    public void onPause() {
        super.onPause();
        music.pause();
    }

    public void onResume() {
        super.onResume();
        music.play();
    }
}
```

Badlogic Games

5 – Implementing Audio

- Who‘s responsible for playing back soundeffects?
- World knows when collisions happen but shouldn‘t play sounds!
- Solution: interface CollisionListener

```
public interface CollisionListener {
    public void collisionWall();
    public void collisionPaddle();
    public void collisionBlock();
}
```

5 – Implementing Audio

- **CollisionHandler is called whenever ball collides with something in World.**

```
public class World {
   CollisionListener listener;

    public World(CollisionListener listener) {
        ...
        this.listener = listener;
    }
    ...
}
```

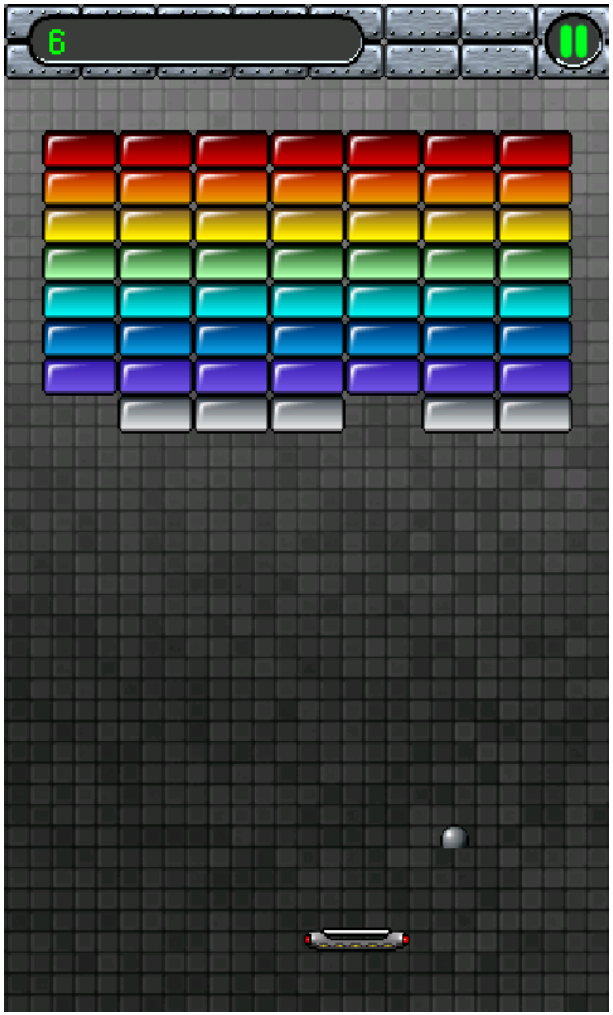
5 – Implementing Audio

- **Load the sound effects in GameScreen.**
- **Register CollisionListener with World**
  - Plays back sound effects!

```
Public class GameScreen extends Screen {
    ...
    Sound bounceSound, blockSound;

    public GameScreen(Game game) {
        ...
        bounceSound = game.loadSound("bounce.wav");
        blockSound = game.loadSound("blocksplosion.wav");
        world = new World(new CollisionListener() {
            public void collisionWall() { bounceSound.play(1); }
            public void collisionPaddle() { bounceSound.play(1); }
            public void collisionBlock() { blockSound.play(1); }
        });
        ...
    }
```

Fin

- **Polish!**
- **Make a real game out of it!**
  - Different block layouts.
  - Power ups.
  - More control schemes.
- **Optimizations**
  - Memory allocation? (Score string)
  - Profiling with DDMS.



Badlogic Games

Engines & Framworks

- Unreal Development Kit (http://www.udk.com/): A commercial game engine running on a multitude of platforms, developed by Epic Games. Those guys made games like Unreal Tournament so it's quality stuff. Uses its own scripting language.
- Unity3D (http://unity3d.com/): Another commercial game engine with great tools and functionalty. It too works on a multitude of platforms, like IOS, Android or in the browser and is easy to pick up. Allows a couple of languages for coding the game logic, Java is not among them.
- jPCT-AE (http://www.jpct.net/jpct-ae/): a port of the Java-based jPCT engine for Android. Has some great features with regards to 3D programming. Works on the desktop and on Android. Closed-source.
- Ardor3D (http://www.jpct.net/jpct-ae/): a very powerful Java-based 3D engine. Works on Android and on the desktop, open-source with great documentation.
- libgdx (http://code.google.com/p/libgdx/): an open-source Java-based game development framework by yours truly, for 2D and 3D games. Works on the Windows, Linux, OSX and of course Android without any code modifications. Develop and test on the desktop without the need for an attached device or the slow emulator.
- Slick-AE (http://slick.cokeandcode.com/): a port of the Java-based Slick framework to Android, build on top of libgdx. Tons of functionality and easy to use API for 2D game development. Cross-platform and open-source of course.
- AndEngine (http://www.andengine.org/): a nice, Java-based Android-only 2D engine, partially based on libgdx code (open source for the win!). Similar in concept to the famous Cocos2D game development engine for IOS.

References

- http://gamedev.net
- http://wiki.gamedev.net
- http://www.flipcode.com/archives/
- http://www.java-gaming.org/
- http://www.gamasutra.com
- http://www.google.com