TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Mathias Weber

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Spieleprogrammierung unter Android

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Spieleprogrammierung unter Android

- Unterschied Anwendungs-/Spieleprogrammierung
- Main-Loop
- Repräsentation der Welt und Zeichnen
- Frames per Second und Einfluss schwacher Hardware
- Reaktion auf Benutzereingaben
- Kollisionserkennung

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Unterschied Anwendungs-/Spieleprogrammierung

Die Spieleprogrammierung unterscheidet sich von der Anwendungsentwicklung in folgenden Punkten:

- Spiele berechnen kein Ergebnis
- Spiele sind *reaktiv*, sie reagieren auf Eingaben des Benutzers in Echtzeit
- Spiele ändern den Zustand in Abhängigkeit von der Zeit
  - Nahe Verwandschaft mit den Simulationen (z. B. Physik-Simulation von Übungsblatt 6)

Main-Loop

Kernkomponente eines Spiels ist der Main-Loop.
Der typische Aufbau eines Main-Loops sieht wie folgt aus:

```
while(running) {
    <Reaktion auf Benutzereingaben>
    <Aktualisierung der Welt>
    <Zeichnen>
    <Kontrolle der Animationsgeschwindigkeit>
}
```

Die Main-Loop wird beim Starten des Spiels gestartet.
Alle Aufgaben werden von dieser Schleife gesteuert.

Repräsentation der Welt

Vorüberlegung: Spieleidee

- Hintergrundgeschichte?
- Wer ist der Held, was versucht er zu erreichen?
- Spielemechanik; was muss der Spieler tun?
- Wie sehen die Regeln aus? Sieg? Niederlage? Fortschritt?

Tappy Defender I

- Die Pilotin Valerie hat ihren Dienst auf der Raumstation beendet und fliegt heim
- Ihr Raumschiff ist beschädigt und sie muss ihren Booster verwendet, um vorwärts und gleichzeitig hoch zu fliegen
- Ein Spieler muss den Bildschirm antippen und halten, um das Raumschiff zu boosten
- Beim Boosten fliegt das Schiff hoch, wird aber auch schneller
- Inspiriert von Flappy Bird

Tappy Defender II

- Dem Raumschiff kommen gegnerische Schiffe entgegen, die es rammen wollen
- Das Schiff des Spielers hat drei Schutzschilde; bei Kollision wird das gegnerische Schiff und ein Schild des Spielers zerstört
- Der Spieler muss eine gewisse Distanz zurücklegen, ohne zerstört zu werden
- Der Spieler bleibt immer am linken Rand des Bildschirms; durch das Boosten kommen die Gegner schneller auf das Schiff zu

Repräsentation der Spielewelt

- Die Spielewelt muss in unserem Programm repräsentiert werden
- Dazu brauchen wir
  - entsprechende Klassen für die Objekte
  - entsprechende Methoden für das Verhalten
  - Bilder zur Anzeige des Zustands der Objekte

Raumschiff des Spielers

Benötigte Attribute:

- Koordinaten (auf dem Bildschirm) mit X- und Y-Koordinate
- Geschwindigkeit des Raumschiffs
- Anzahl der verbleidenden Schilde
- Grafik zur Darstellung des Schiffs auf dem Bildschirm (Bitmap)

Raumschiffe der Gegner

Benötigte Attribute:

- X- und Y-Koordinaten
- Geschwindigkeit des Raumschiffs
- Grafik zur Darstellung des Schiffs

Abstraktion über gemeinsame Attribute möglich.

Interface GameObject

Alle Objekte, die im Spiel auftauchen, brauchen die Koordinaten auf dem Bildschirm und ein Bitmap, um das Objekt anzeigen zu können:

```
public interface GameObject {
    int getX();
    int getY();
    Bitmap getBitmap();
}
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Der gemeinsame Zustand für Objekte im Spiel kann in der Klasse `AbstractGameObject` abstrahiert werden.

```
public abstract class AbstractGameObject implements GameObject{
    protected Bitmap bitmap;
    protected int x, y;

    @Override
    public Bitmap getBitmap() {
        return bitmap;
    }
    @Override
    public int getX() {
        return x;
    }
    @Override
    public int getY() {
        return y;
    }
}
```


TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Der Spieler: Klasse PlayerShip I

Auf Basis der Klasse `AbstractGameObject` können wir die Klasse für das Spieler-Schiff erstellen.

- Die Koordinaten, das Bitmap und die Geschwindigkeit müssen entsprechend initialisiert werden (Zugriff über `protected` Attribute möglich)
- Die Geschwindigkeit soll nach oben (nicht unendlich beschleunigen) und unten (Spieler soll nicht stehen bleiben) begrenzt sein.
- Der Spieler soll sich weder oben noch unten aus dem Bild bewegen können.
- Der Zustand der Schilde muss gespeichert werden.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

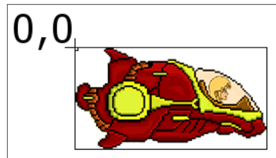
Der Spieler: Klasse PlayerShip II

```
public class PlayerShip extends AbstractGameObject {
    private int speed;
    private int shieldStrength;
    private boolean boosting;

    public static final int MAX_SPEED = 15;
    public static final int MIN_SPEED = 2;
    public static final int GRAVITY = -7;
    private final int maxY;
    private final int minY;

    public PlayerShip(Context context, int screenX, int screenY) {
        x = 50;
        y = 50;
        speed = MIN_SPEED;
        shieldStrength = 3;
        bitmap = BitmapFactory.decodeResource(context.getResources(),
                                              R.drawable.ship);

        maxY = screenY - bitmap.getHeight();
        minY = 0;
    }
...}
```



TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Der Spieler: Klasse PlayerShip III

Der Zustand des Spielers muss periodisch aktualisiert werden, abhängig von der vergangenen Zeit. Dabei beachten wir folgende Regeln:

- Wenn die Booster-Triebwerke laufen, erhöht sich die Geschwindigkeit, sonst verringert sie sich.
- Die Geschwindigkeit wird durch `MIN_SPEED` und `MAX_SPEED` begrenzt.
- Der Spieler soll immer am linken Rand des Spielfelds sein, somit ändert sich die X-Koordinate nicht.
- Die Y-Koordinate muss entsprechend der Geschwindigkeit und "Gravitation" angepasst werden.

Diese Berechnungen entsprechen nicht der Realität (siehe Gravitation), sondern sind der Spieledynamik geschuldet.

Der Spieler: Klasse PlayerShip IV

```
public void update(double deltaT) {
    if (boosting) {
        speed += 2;
    } else {
        speed -= 3;
    }

    if (speed > MAX_SPEED) {
        speed = MAX_SPEED;
    }

    if (speed < MIN_SPEED) {
        speed = MIN_SPEED;
    }

    y -= (speed + GRAVITY) * deltaT;

    if (y < minY) {
        y = minY;
    } else if (y > maxY) {
        y = maxY;
    }
}
```


Die Gegner: Klasse EnemyShip I

Die Klasse für die generischen Raumschiffe kann ebenfalls auf Basis der Klasse `AbstractGameObject` definiert werden. Hier gilt:

- Das Bitmap der Gegner muss initialisiert werden.
- Die Gegner sollen am rechten Rand in zufälligen Höhen auftauchen.
- Die Geschwindigkeit der Gegner soll zufällig sein.
- Die Gegner fliegen von rechts nach links auf den Spieler zu.

Die Gegner: Klasse EnemyShip II

```
public class EnemyShip extends AbstractGameObject {
    private int speed;

    private int maxX;
    private int minX;
    private int maxY;
    private int minY;

    public EnemyShip(Context context, int screenX, int screenY) {
        bitmap = BitmapFactory.decodeResource(context.getResources(),
                                               R.drawable.enemy);
        maxX = screenX;
        maxY = screenY;
        minX = 0 - bitmap.getWidth();
        minY = 0 - bitmap.getHeight();

        reset();
    }
    private void reset() {
        Random generator = new Random();
        speed = generator.nextInt(10)+3;
        x = maxX;
        y = generator.nextInt(maxY - minY) + minY;
    }
```


Die Gegner: Klasse EnemyShip III

Da der Spieler sich nicht vom linken Rand weg bewegt, muss dessen Geschwindigkeit beim Aktualisieren des Zustands der Gegner mit einberechnet werden.

Um die Anzahl der Gegner, die sich aktuell auf dem Bildschirm befinden, gleich zu halten, werden Gegner, die den Bildschirm verlassen, auf der anderen Seite neu positioniert.

Die Gegner: Klasse EnemyShip IV

```
    public void update(double deltaT, int playerSpeed) {
        x -= playerSpeed * deltaT;
        x -= speed * deltaT;

        if (x < minX) {
            reset();
        }
    }
...}
```


Spielezustand

Der Zustand unseres Spiels besteht aus folgenden Elementen:

- Ein Spielerschiff (`PlayerShip`)
- Mehrere feindliche Schiffe (`List<EnemyShip>`)
- Zusammen bilden sie die Spieleobjekte, die gezeichnet werden müssen (`List<GameObject>`)

Android Anwendung I

Bis hier ist alles, mit Ausnahme der Behandlung der Bitmap, sehr generisch. Wir wollen nun eine Android-Anwendung erstellen, die unser Modell der Welt zeichnet und Eingaben erlaubt.

Dazu erzeugen wir

- eine Hauptaktivität, die beim Starten des Spiels aufgerufen wird,
- eine Spielaktivität, die das eigentliche Spiel darstellt und
- die Zeichenfläche, die die Spielinhalte darstellt.

Android Anwendung II

Aktivitäten und Intents zum Wechseln zwischen Aktivitäten wurden bereits in Kapitel 25 behandelt.

Der View dient zur Darstellung des aktuellen Zustands des Spiels. Unterstützung der Zeichenmethoden bietet die Klasse `SurfaceView`.

```
public class TDView extends SurfaceView {
    private final int screenX;
    private final int screenY;

    public int getScreenX() {
        return screenX;
    }
    public int getScreenY() {
        return screenY;
    }
    public TDView(Context context, int x, int y) {
        super(context);
        this.screenX = x;
        this.screenY = y;
    }
}
```

Android Anwendung III

Die aktuelle Größe des Bildschirms kann durch Zugriff auf das Android-Framework ermittelt werden:

```
Display display = getWindowManager().getDefaultDisplay();
Point size = new Point();
display.getSize(size);
gameView = new TDView(this, size.x, size.y);
```

Der Main-Loop muss in möglichst regelmäßigen Abständen ein Bild des aktuellen Zustandes (Frame) auf den `TDView` zeichnen.

Zeichenroutine I

Da sowohl das Android-Framework als auch unser Main-Loop auf den View zugreifen, müssen wir Android mitteilen, wann wir einen neuen Frame zeichnen; wir müssen den View sperren (engl. *lock*).

Nach dem Zeichnen des Frames müssen wir die Zeichenoberfläche wieder freigeben, damit Android den Frame auf dem Bildschirm anzeigen kann.

In Android verwendet man dazu einen `SurfaceHolder`, welchen wir mit `view.getHolder()` bekommen können und in der Variablen `holder` speichern.

Zeichenroutine II

```
private void draw() {
    if (holder.getSurface().isValid()) {
        Canvas canvas = holder.lockCanvas();

        // clear the old frame
        canvas.drawColor(Color.argb(255, 0, 0, 0));


        // Frame zeichnen

        holder.unlockCanvasAndPost(canvas);
    }
}
```

Beim Zeichnen des Frames ist die Reihenfolge der Aufrufe wichtig, da sie festlegt, welche Elemente von anderen überdeckt werden.

Zeichenroutine III

Zum Zeichnen des Frames müssen wir bis jetzt nur alle `GameObject`-Instanzen zeichnen:

```
// draw all game objects
for (GameObject o : gameObjects) {
    canvas.drawBitmap(o.getBitmap(), o.getX(), o.getY(),
        paint);
}
```

Die Methode `drawBitmap` zeichnet das Bitmap der einzelnen Objekte und die angegebenen Koordinaten. Das Paint-Objekt gibt weitere Konfigurationsparameter an, jedoch benötigen wir in diesem Beispiel noch keine solchen Parameter.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

FPS und schwache Hardware I

Jetzt, da wir die Zeichenroutine haben, können wir in unserem Main-Loop die Frames des Spiels zeichnen. Abhängig von der Geschwindigkeit der Hardware dauert dieser Vorgang jedoch unterschiedlich lang.

Diese Dauer muss bei der Berechnung des aktualisierten Zustands pro Frame mit einberechnet werden. Dazu muss diese Zeit im Main-Loop gemessen werden.

Zeichnen wir so oft wie möglich, führt dies zu hoher Prozessorlast und damit auch zu hohem Akku-Verbrauch.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

FPS und schwache Hardware II

```
long lastLoopTime = System.nanoTime();
while(playing) {
    long now = System.nanoTime();
    long updateLength = now - lastLoopTime;
    lastLoopTime = now;

    // updateLength in nanoseconds
    update(updateLength);
    draw();
}
```


TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

FPS und schwache Hardware III

```
private void update(long deltaT) {
    double time = deltaT / 1000000.0;
    time = time / 15.0;

    // update player
    player.update(time);

    // update all enemies
    for (EnemyShip enemy : enemies) {
        enemy.update(time, player.getSpeed());
    }
}
```


TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Beschränkung der Framerate I

Um den Prozessor und den Akku zu schonen, wollen wir die Framerate (Anzahl der Frames pro Sekunde, die gezeichnet werden) begrenzen. Dazu müssen wir zwischen dem Zeichnen der einzelnen Frames eine gewisse Zeit warten.

Framerate von 30 Frames pro Sekunde bedeutet, ich muss 1/30 Sekunden warten, bis ich den nächsten Frame zeichne.

Aber: Ich brauche auch Zeit, um den Frame zu zeichnen. Also:

$$t_{zeichnen} + t_{warten} = 1/30\text{s}$$

und somit

$$t_{warten} = 1/30\text{s} - t_{zeichnen}$$

Beschränkung der Framerate II

In der MainLoop-Klasse:

```
public static final int TARGET_FPS = 30;
public static final long FRAME_TIME = 1000000000 / TARGET_FPS;
```

Anpassung des Main-Loops:

```
long lastLoopTime = System.nanoTime();
while(playing) {
    long now = System.nanoTime();
    long updateLength = now - lastLoopTime;
    lastLoopTime = now;

    handleInput();
    // updateLength in nanoseconds (around 33 milliseconds)
    update(updateLength);
    draw();
    control(System.nanoTime() - lastLoopTime);
}
```

Die Methode `control` wartet
`FRAME_TIME - (System.nanoTime() - lastLoopTime)` lange.

Bemerkungen

- Bei schwacher Hardware wird die gewünschte Framerate nicht erreicht.
- Durch Einberechnung der tatsächlichen Zeit zwischen zwei Schleifendurchläufen wird die Geschwindigkeit durch die Framerate nicht beeinflusst.
- `System.nanoTime()` gibt einen Zeitstempel in Nanosekunden an (typischer Weise Nanosekunden seit letztem Systemstart)
- Die Methoden zum Warten sind unter Java nicht genau genug, um die Framerate exakt zu halten.

Reaktion auf Benutzereingaben I

Bis jetzt haben wir nur eine Simulation. Ein Spiel muss jedoch auf Eingaben des Benutzers reagieren. In diesem Fall wollen wir unser Player-Schiff steuern.

Beim Berühren des Bildschirms soll das Schiff den Booster anschalten, beim Loslassen soll der Booster ausgeschaltet werden.

Diese Aktionen modellieren wir als Aufzählungstyp (*Enum*):

```
public enum GameAction {
    TOUCH_DOWN, TOUCH_UP
}
```


Reaktion auf Benutzereingaben II

Beim Auftreten der Aktionen (in Form von Events) werden sie in einer Queue gesammelt und durch die Methode `handleInput` verarbeitet.

```
private void handleInput() {
        Iterator<GameAction> iterator = actions.iterator();
        GameAction action;
        while (iterator.hasNext()) {
            action = iterator.next();
            switch (action) {
                case TOUCH_DOWN:
                    player.startBoosting();
                    break;
                case TOUCH_UP:
                    player.stopBoosting();
                    break;
            }
            iterator.remove();
        }
    }
```

Illusion der Bewegung I

Zu diesem Zeitpunkt haben wir folgendes:

- Wir können das Spiel ausführen
- Die Gegener kommen auf uns zu
- Wir können das Spielerschiff hoch und runter bewegen

Es sieht jedoch so aus, als ob der Spieler sich nicht bewegt. Um die Bewegung des Spielers anzudeuten, verwenden wir die Illusion, dass sich die Sterne im Hintergrund bewegen.

Illusion der Bewegung II

```
public class SpaceDust {
    private int x,y;
    private int size;
    private int maxX;
    private int maxY;
    private int minX;
    private int minY;

    public SpaceDust(int screenX, int screenY) {
        maxX = screenX;
        maxY = screenY;
        minX = 0;
        minY = 0;
        Random generator = new Random();
        size = generator.nextInt(5)+1;
        // set starting coordinates
        x = generator.nextInt(maxX);
        y = generator.nextInt(maxY);
    }
```


Illusion der Bewegung III

```
public void update(double deltaT, int playerSpeed) {
    x -= playerSpeed * deltaT;

    // respawn
    if (x < 0) {
        x = maxX + x; // distribute the dust evenly
        Random generator = new Random();
        y = generator.nextInt(maxY);
        speed = generator.nextInt(10);
    }
}
```


Kollisionserkennung I

Die Darstellung des Spiels sieht schon recht gut aus, jedoch fliegt unser Player durch gegnerische Schiffe einfach durch, ohne dass etwas passiert. Wir haben jedoch definiert, dass der Spieler mit den Gegner kollidieren sollen.

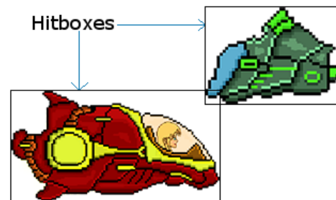
Wie berechnen wir, dass unser Spieler mit einem Gegner *kollidiert* ist?

Die Berechnung auf Basis der vollständigen Figur wäre zu aufwändig und somit die Berechnung zu langsam (wir wollen die Framerate nicht zu sehr verringern).

Kollisionserkennung II

Verwende einfachere Figuren, die das Spielobjekt annähernd beschreiben und berechne deren Überlappung.

Häufig werden Rechtecke verwendet, welche auf englisch dann auch *bounding boxes* genannt werden. Bei Verwendung zur Kollisionserkennung werden die bounding boxes auch oft *hitboxes* genannt.



Kollisionserkennung III

Wir fügen hitboxes zu unseren `GameObject`s hinzu und berechnen während der Aktualisierung des Zustands, ob unser Spieler kollidiert ist.

```
public Rect getHitbox() {
    return hitBox;
}
```

In der `update` Methode:

```
// collision detection with enemy ships
boolean collision = false;
for (EnemyShip enemy : enemies) {
  if (Rect.intersects(player.getHitbox(), enemy.getHitbox())) {
      enemy.onCollision();
      collision = true;
  }
}
```


Kollisionserkennung IV

Gegner, die mit unserem Spiele kollidieren tauchen am rechten Rand wieder auf, beim Spieler geht gleichzeitig ein Schild verloren. Hat der Spieler kein Schild und kollidiert, ist das Spiel vorbei.

```
if (collision) {
    player.onCollision();
    if (player.getShieldStrength() < 0) {
        // game over
        gameOver = true;
    }
}
```

Der Rest des Spiels muss entsprechend angepasst werden, um die Aktualisierung nur zu machen, wenn das Spiel noch nicht vorbei ist (also `gameOver == false`).

Komplettierung

Unserem Spiel fehlen noch ein paar wichtige Komponenten, welche hier jedoch nicht im Detail besprochen werden:

- Anzeige des Zustands des Spielers (z.B. Anzahl der verbleibenden Schilde, zurückgelegte Distanz)
- Andere Bedingungen, wann das Spiel beendet wird (z.B. beim Erreichen einer vorgegebenen Distanz)
- Bestenlisten
- Töne, die Aktionen im Spiel signalisieren
- Feineinstellung der Schwierigkeit und verschiedene Schwierigkeitsstufen

Weiterführende Literatur

John Horton: Android Game Programming by Example. 2015, Packt Publishing

Android Dokumentation: Tutorials für Entwicklung von Android Anwendungen, Dokumentation der Android Bibliotheken, Beispiele. https://developer.android.com/develop/index.html

libGDX: Cross Platform Game Development Library. https://libgdx.badlogicgames.com/