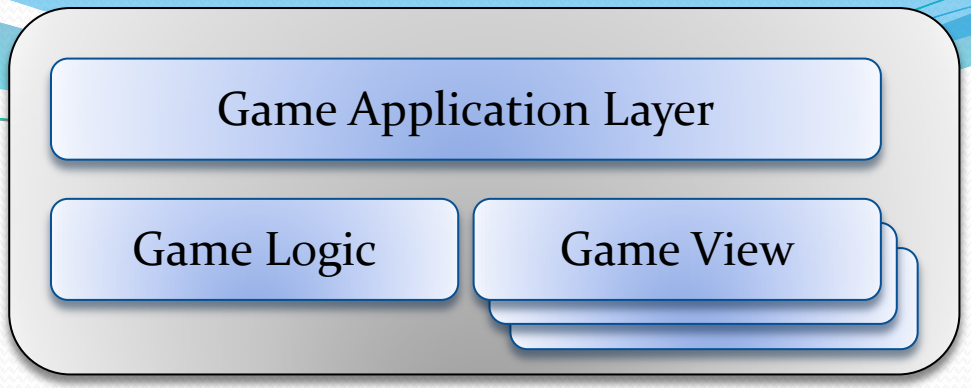
Aufbau interaktiver 3D-Engines

Universität Osnabrück
Fachbereich Mathematik / Informatik

3. Vorlesung (22.04.2013)

Prof. Dr. rer. nat. Oliver Vornberger
Nico Marniok, B. Sc.
Erik Wittkorn, B. Sc.

Rückblick 1



- Aufbau einer Engine muss klar strukturiert sein und vom Entwickler durchschaut werden
- Game Application Layer
  - Initialisierungen
  - Hardwarenahe Implementationen
  - Interface zur Systemresourcen
- Game Logic
  - Simulation der Spielwelt
  - Verwaltung von Actors
- Game View
  - Sicht auf Spielwelt
  - Interface zwischen Spieler und Logik
  - Nächste Woche mehr…

Rückblick 2

- Events als primäres Kommunikationsmittel zwischen Komponenten
  - Events werfen (**`triggerEvent()`** / **`queueEvent()`**)
  - Auf Events hören (`EventListener.java`)
- Objekte der Spielwelt heißen *Actors*
  - Ein Actor besteht aus beliebig vielen Komponenten
    - Jeder Typ maximal ein mal!
  - Eine Komponente beschreibt eine Eigenschaft des Actors
    - Visuelle Darstellung
    - Physikalische Eigenschaften
    - Einsammelbar
    - …

Heute

- Ergänzung zu ActorComponents
- Einschub: `config.ini`
- Events vs. Prozesse
- Timer
- Input
- Resource Cache

Ergänzung: ActorComponents

- Ergänzung zur Initialisierung (Code wird noch aktualisiert)
- Problem
  - Abhängigkeiten zwischen ActorComponents
  - Undefinierte Reihenfolge der Knoten einer xml-Datei
- Lösungsvorschlag
  - Erstellen der Actor Instanz
  - Erzeugen und Anhängen aller ActorComponents
  - Aufrufen aller **`init()`** Methoden
    - Bei Fehlschlag: Actor zerstören
  - Aufrufen aller **`postInit()`** Methoden
- Vorteil: Während **`init()`** bereits bekannt, welche ActorComponents es geben wird

Einschub: `config.ini`

- Benutzer haben unterschiedliche Vorlieben oder Einschränkungen für die Interaktionen mit dem Spiel
  - Vollbild oder Fenster?
  - 800 x 600 oder 7680 x 4800?
  - Steuerung mit WASD oder Pfeiltasten?
- Festlegung im Code offensichtlich keine gute Idee
- Lösung: Mitliefern einer den eigenen Wünschen anpassbaren Datei `config.ini`

Einschub: config.ini

- Implementation relativ langweilig
- Inhalt von der Art „key=value“
- Kommentare starten mit #
- Optional: Bereiche definieren mit [BEREICHNAME]
- Beispiel

```
[DISPLAY]
iResolutionX=1920
iResolutionY=1080
iFrequency=120
bWindowed=true
bVerticalSync=false

[RESOURCE]
sResourcePath=D:\Dropbox\Aufbau interaktiver 3D-Engines\RubiksCubeData
sResourceFile=./resources.zip
```

Einschub: config.ini

- Konventionen
  - Bereichsnamen in Großbuchstaben
  - Strings referenzierende Schlüssel starten mit kleinem s
  - Integer referenzierende Schlüssel starten mit kleinem i
  - Float referenzierende Schlüssel starten mit kleinem f
  - Boolean referenzierende Schlüssel starten mit kleinem b
  - Farben referenzierende Schlüssel starten mit kleinem c
- Werte besorgen
  - `GameApp.getConfig().getAsInt(„iResolutionX“);`
  - `GameApp.getConfig().getAsBoolean(„bWindowed“);`
  - …

Einschub: config.ini

- Benutzungshinweise
  - Mitgeben einer default_config.ini in jar Datei
  - ```
    GameApp.setConfigFile(
      Main.class.getResourceAsStream(
            "/config/default_config.ini"));
    ```
  - Die default_config.ini wird nicht verändert
  - Schlüssel, die in der default_config.ini enthalten sind, aber nicht in der config.ini, werden ergänzt
  - Unterschiedliche Werte in der default_config.ini und der config.ini bleiben unverändert
  - Die Struktur der config.ini wird automatisch an die der default_config.ini angepasst
    ```

Beschränkte Events

- Events beschreiben *einen einzelnen* Zeitpunkt
  - Actor wurde zerstört
  - Explosion fand statt
  - …
- Wie würden wir eine Aktion beschreiben, die eine zeitliche Ausdehnung hat?
  - Spieler bereitet Bombe vor
  - Bombe tickt
  - Spieler lädt nach
  - Kaufphase läuft
- In Events nicht vorgesehen

Lösung: Prozesse

- Besitzt zeitliche Ausdehnung
  - Muss vorher nicht festgelegt werden
- Besitzt Methode `void update(long dMillis);`
- Kann erfolgreich sein oder fehlschlagen
- Kann pausiert oder abgebrochen werden
  - Kann durch Implementation aber auch umgangen werden
- Kann einen Kindprozess besitzen, der bei erfolgreicher Terminierung gestartet wird
  - Wenn „Spieler legt Bombe“ erfolgreich, dann starte „Bombe tickt“

Prozesse

- Einzelne Prozesse erben von `EngineProcess.java`
  - Muss implementiert werden
    - `void update(long dMillis);`
  - Weitere nützliche (bereits implementierte) Methoden
    - `void succeed()` – Beendet den Prozess erfolgreich
    - `void fail()` – Lässt den Prozess fehlschlagen
    - `void pause()` – Pausiert den Prozess
    - `void resume()` – Führt den Prozess nach einer Pause fort
    - `void attachChild()` – Setzt **den** Kindprozess

Prozesse

- Nützliche Methoden zum Überschreiben
  - `void` **`onInit()`** – Aufruf vor dem ersten **`update()`**
  - `void` **`onSuccess()`** – Aufruf nach **`succeed()`**
  - `void` **`onFail()`** – Aufruf nach **`fail()`**
  - `void` **`onAbort()`** – Aufruf nach externem Abbruch
- Ob8 beim Überschreiben!

```
public void onInit() {
    super.onInit();
    // now my stuff
    // ...
}
```

Prozessverwaltung

- Verwaltung in zentraler Klasse `ProcessManager.java`
- `GameApp.getGlobalProcessManager();`
- Eigene Prozessmanager können ebenfalls erstellt und benutzt werden
  - Beispielsweise für GUI Animationen
- Nützliche Methoden
  - `void attachProcess(EngineProcess process);`
    - Fügt einen neuen Prozess hinzu
  - `void abortAllProcesses(boolean immediate);`
    - Bricht alle Prozesse ab
  - `int update(long deltaMillis);`
    - Führt eine Reihe von Operationen mit den Prozessen aus

Prozessverwaltung

- Prozesse führen über einen oder nach einem Zeitraum Operationen durch
- Im Gegensatz zu Events keine Benachrichtigung über neue oder beendete Prozesse
  - Mögliche Lösung: Events in **`onInit()`** bzw. **`onSucceed()`** werfen
- Typische Anwendungen für Prozesse
  - Animationen von GUI Elementen
  - Allgemein: Zeitgesteuerte Spielelemente
  - Generische Animationen mit Objekten, für die keine Animation vorgesehen ist
    - Beispiel: Drehung einer Ebene des RubiksCube

Parallelisierte Prozesse

- Prozesse laufen standardmäßig singlethreaded
- Gewisse Prozesse sind theoretisch parallelisierbar
  - Solange sie *keine* LJWGL Aufrufe durchführen!
- Synchronisierung mittels **`update()`** Methode denkbar
- Nur zu empfehlen, falls man sich sehr gut mit Threads in Java auskennt!
- Tipps
  - Auf keinen Fall alle Methoden `synchronized` machen!
  - Overhead der Threaderzeugung beachten!
  - Thread zum Daemon machen
  - Thread sinnvollen Namen geben (Debugging)
  - Logger und Resource Cache sind bereits threadsafe implementiert

Timer

- Viele Elemente einer Engine zeitkritisch
  - Simulation
  - Animationen
  - Prozesse
  - …
- `void update()`
  - Wird genau einmal jeden Frame aufgerufen
  - Wie viel Zeit ist in der Zwischenzeit vergangen?
  - Aufruf von bspw. `QueryPerformanceCounter()` oder `System.nanoTime()` systemabhängig
  - Hat daher nichts in Logic oder View zu suchen

GameApp: Timer

- **`Timer`**`(`*`Precision`* `precision)`
  - Konstruktor mit Präzision (meistens Millisekunden, 10^{-3}s)
- `boolean` **`init()`** / `void` **`destroy()`**
- `void` **`next()`**
  - Berechnet den aktuellen absoluten Zeitstempel
- `long` **`getDelta()`**
  - Liefert die Zeit zwischen dem aktuellen und dem letzten Zeitstempel
- `long` **`getTimeStamp()`**
  - Liefert den aktuellen Zeitstempel
- `void` **`setFactor`**`(long numerator, long denominator)`
  - Setzt den Faktor dieses Timers gegenüber der Echtzeit
  - Parameter 1 und 2 resultieren darin, dass der Timer die Hälfte der real vergangenen Zeit zurückliefert (Zeitlupe)

GameApp: Timer

- Ob8! Auflösung des Systemtimers muss nicht Auflösung der Engine entsprechen!
- Naive Implementation

```
private long timestamp;
private long delta;

private boolean init() {
    this.timestamp = System.nanoTime();
    this.delta = 0;
    return true;
}

public void next() {
    long now = System.nanoTime();
    this.delta = (now - this.timestamp) / 1000000L;
    this.timestamp = now;
}
```

GameApp: Timer

- Was ist das Problem? Achtung: Mathematik!

```
public void next() {
    long now = System.nanoTime();
    this.delta = (now - this.timestamp) / 1000000L;
    this.timestamp = now;
}
```

- Bei mehr als 1000 FPS dauert ein Frame weniger als 1 Millisekunde
- Bei Genauigkeit des Systems von Nanosekunden ist $d = now - this.timestamp < 1000000 = 10^6$
- Damit ist $\delta = 10^{-6} \cdot d < 10^{-6} \cdot 10^6 = 1$
- Und da δ vom Typ `long` ist, gilt in jedem Frame $\delta = 0$
- Entwarnung: Mathematik Ende (fürs Erste)

GameApp: Timer

- Deutliche Probleme treten in der Praxis bereits bei mehr als 500 FPS auf (Mikroruckler / „Unnatürliche“ Animationen)
- Lösung
  - FPS begrenzen
    - schlecht für spätere Leistungsoptimierung
    - Auch hier bereits leicht spürbare Probleme
  - Timer auf `double` basieren lassen
    - Führt dazu, dass in einzelnen Komponenten dasselbe passiert
  - Akkumulator
    - Idee: zusätzliche Variable, die „verschluckte“ Zeitspannen aufsummiert

GameApp: Timer

```
private long timestamp;
private long delta;
private long accu;

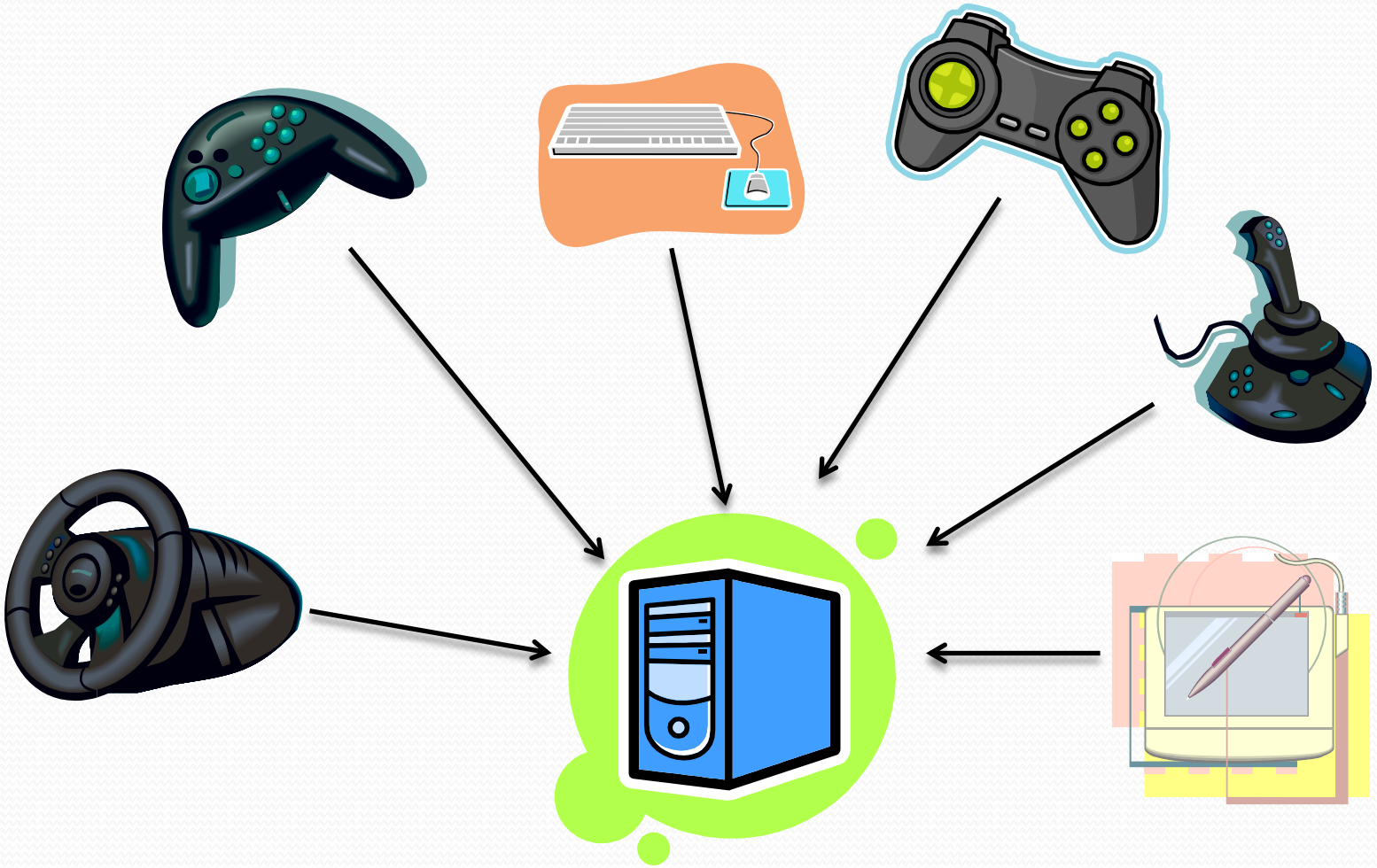
private boolean init() {
    this.timestamp = System.nanoTime();
    this.delta = 0;
    this.accu = 0;
    return true;
}

public void next() {
    long now = System.nanoTime();
    long realDelta = now - this.timestamp + this.accu;
    this.delta = realDelta / 1000000L;
    this.accu = realDelta - this.delta * 1000000L;
    this.timestamp = now;
}
```

GameApp: Timer

- Timer ist Teil der GameApp
- Nützlich: 2 Timer
  - RealTimer, der immer die real vergangene Zeit liefert
  - GameTimer, der die vergangene Zeit im Spiel liefert
    - Auf diese Weise „schneller Zeitvorlauf“ oder Zeitlupe leicht zu realisieren, ohne Auswirkung auf GUI, Bewegungen der Kamera, …
- `GameApp.getRealTimer();`
- `GameApp.getGameTimer();`
- `void update(long gameMillis);`

Spielerinput



Input

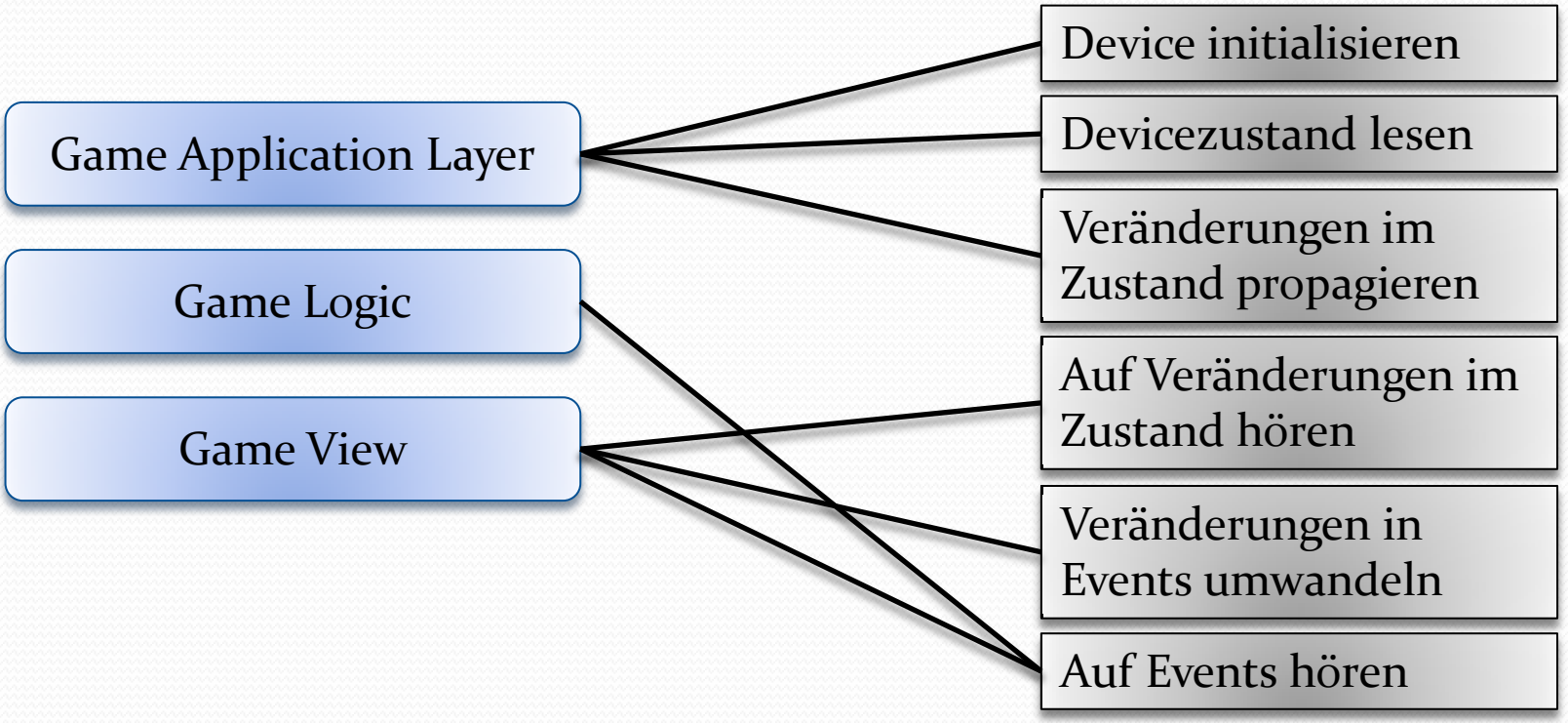
- Extrem naive Implementation

```
void update(long deltaMillis) {
    if(Keyboard.isKeyDown('W') || Gamepad.getAnalog(0).getY() > 0.1) {
        this.movePlayerForward();
    }
    if(Keyboard.isKeyDown('S') || Gamepad.getAnalog(0).getY() < -0.1) {
        this.movePlayerBackward();
    }
    // nervt jetzt schon...
}
```

- Offensichtliche Nachteile
  - Unübersichtliche **`update()`** Methode
  - Keine Anpassungsmöglichkeiten für den Benutzer
  - Code kaum übertragbar auf beliebige Spiele
- Versteckter Nachteil
  - **`isKeyDown()`** liefert *momentanen* Zustand

Input

- Input richtig verarbeiten
  - Aufgabenverteilung



Input: Device initialisieren

- Hardwarenah, daher GameApp
- Wir beschränken uns auf Maus / Tastatur
- In LWJGL Initialisierung sehr einfach
  - `Mouse.create();`
  - `Keyboard.create();`
- Allgemein leider nicht
  - Rawinput in C/C++ mit Windows
  - DirectInput in DirectX API

Input: Devicezustand lesen

- Unterschiedliche Ansätze in Betriebssystemen / APIs
- Eine Möglichkeit
  - Zustand des Devices wird in Bytearray gespeichert
  - Inhalt abhängig vom Device
  - Beispielsweise repräsentieren einzelne Bytes den Zustand einer Taste auf der Tastatur oder den Ausschlag des Joysticks
  - Häufig einzige Möglichkeit, exotische Devices anzusprechen
  - Nachteil auch hier
    - Inputs zwischen zwei Zuständen gehen verloren

Input: Devicezustand lesen

- Andere Möglichkeit
  - Sammeln von Inputs als Events
  - API (hier: LWJGL, Keyboard) stellt Funktionen bereit, wie
    - `int` **`getEventKey`**`()` – Liefert den Code einer Taste, die dem Event zugeordnet ist
    - `boolean` **`getEventKeyState`**`()` – Liefert `true`, falls die Taste gedrückt wurde; `false`, falls sie losgelassen wurde
    - `void` **`next`**`()` – Geht weiter zum nächsten Event
  - Vorteil
    - Inputs gehen nicht verloren
  - Nachteil
    - Nicht auf beliebige Devices übertragbar

Input: Gamepad / Joystick

- Mit LWJGL mittels `jinput.jar` möglich
- Werden wir nicht behandeln
- Darf aber in Projekten benutzt werden

Input: Input propagieren

- Geschieht mittels spezieller Interfaces
- Swing sehr ähnlich
- Zwei Interfaces in der Engine verfügbar
  - `KeyboardHandler.java`
    - Wird mit Input auf der Tastatur aufgerufen
  - `PointerHandler.java`
    - Wird mit Input der Maus aufgerufen

Input: KeyboardHandler.java

- Interface mit drei Methoden
  - `boolean onKeyDown(int key, char c);`
    - Wird mit dem Drücken einer Taste aufgerufen
  - `boolean onKeyUp(int key);`
    - Wird mit dem Loslassen einer Taste aufgerufen
  - `boolean isProcessingRepeatEvents();`
    - Liefert diese Methode `true`, werden Repeatevents mittels `onKeyDown()` propagiert
- Rückgabewerte geben an, ob das Event konsumiert wurde

Input: PointerHandler.java

- Interface mit vier Methoden
  - `boolean pointerMoved(int x, int y, int dX, int dY);`
    - Wird bei einer Bewegung der Maus aufgerufen
  - `boolean onButtonDown(int button);`
    - Wird aufgerufen, wenn ein Button gedrückt wurde
  - `boolean onButtonUp(int button);`
    - Wird aufgerufen, wenn ein Button losgelassen wurde
  - `boolean onWheelMoved(int delta);`
    - Wird aufgerufen, wenn das Mausrad bewegt wurde
- Rückgabewerte geben auch hier an, ob das Event konsumiert wurde
- Drags und Klicks / Doppelklicks müssen in der Implementation manuell ermittelt werden

Input: Handler

- Handler müssen registriert werden
  - `GameApp.getInputHandler().registerKeyboardHandler(handler);`
    - Registriert einen Keyboardhandler
  - `GameApp.getInputHandler().registerPointerHandler(handler);`
    - Registriert einen Pointerhandler
- Reihenfolge wichtig!
  - Handler werden bei Events in umgekehrter Reihenfolge durchlaufen
  - Liefert ein Handler `true` zurück, wird das Event nicht an die restlichen Handler propagiert

Input: Weiter verarbeiten

- Handler hören auf Eingaben vom Benutzer, daher Game View
- Eingaben können hier direkt verarbeitet werden, *falls* sie lediglich die Ansicht des Spielers beeinflussen, wie bspw.
  - Spieler öffnet Menü
  - Spieler öffnet Inventar
- Eingaben sollten in Events umgewandelt werden, falls sie die Spielwelt beeinflussen, wie bspw.
  - Spieler bewegt sich nach vorne
  - Spieler lädt nach
  - Spieler wechselt zu Außenperspektive (Manipulation des Kameraactors)

Input: Unsere Engine

- Events können mit Tastatur- / Mauseingaben verknüpft werden
- Konfiguration in `config.ini`
- Vorteil
  - Nach Definition der Events keinerlei Änderungen am Code mehr nötig
  - Sehr nützlich zum Debuggen
    - Wird ein Event korrekt verarbeitet?
  - Komplett konfigurierbar durch Benutzer
- Wichtig: Events komplett mit Package referenzieren
  - `event.events.QuitEvent=ESCAPE`

Input: Spezielle Events

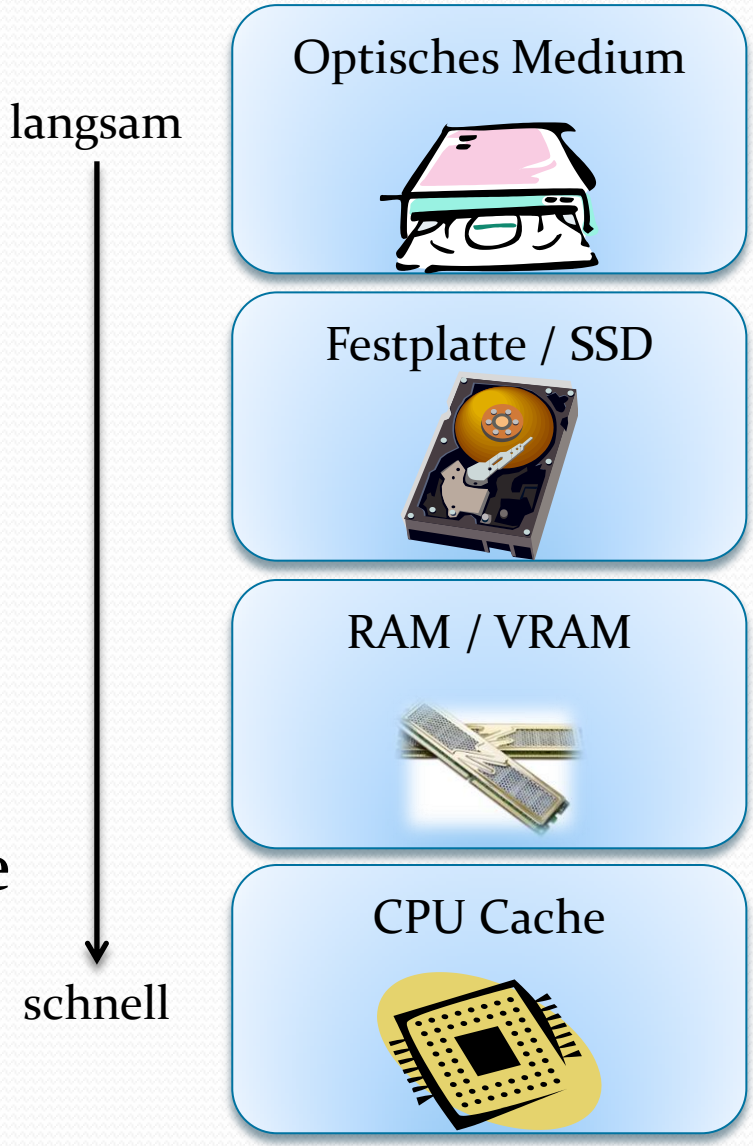
- `ToggleEvent.java`
  - Spezielles Event für Tastenanschläge
  - `boolean` **`isActivated`**`()` – Liefert true, falls die Taste gedrückt wurde
- `PointerMovedEvent.java`
  - Spezielles Event für Mausbewegungen
  - `int` **`getDeltaX`**`()` – Liefert die Mausbewegung
  - `int` **`getDeltaY`**`()` – Liefert die Mausbewegung
- `PointerWheelEvent.java`
  - Spezielles Event für die Bewegung des Mausrads
  - `int` **`getDelta`**`()` – Liefert die Bewegung des Mausrads

Resourcen

- Der Code eines Spiels nimmt nur einen Bruchteil der gesamten Datengröße ein
- Ein 3D-Spiel besteht aus
  - Models
  - Texturen
  - Audio
  - Konfigurationsdateien
    - Actors
    - Levels
  - …

Resourcen

- Speicherhierarchie
- In der Regel zu viele Daten für den RAM
  - Alles laden nicht möglich
- Gute Organisation der Daten von großer Wichtigkeit
- Lösung: Der Resource Cache



Resourcen: Metapher aus GCC

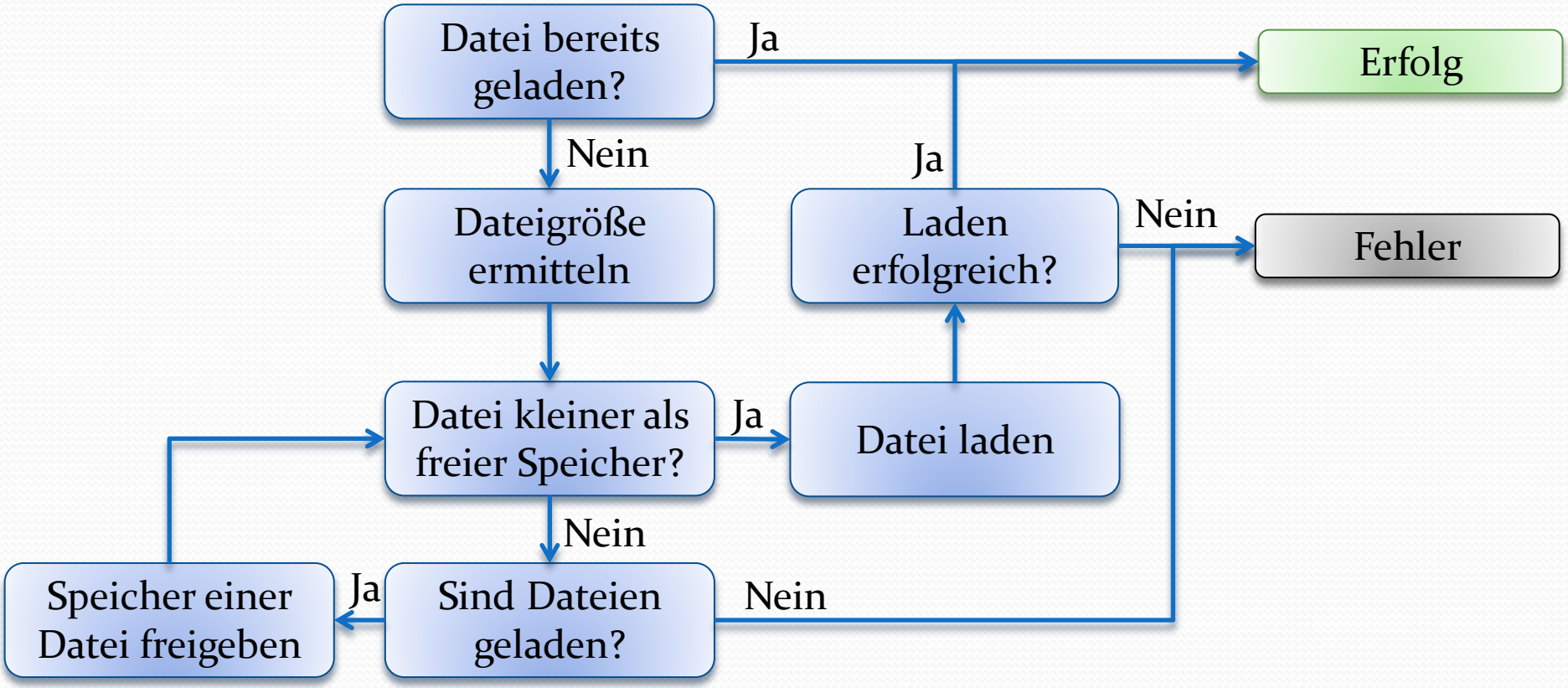
- Betrachte ein brennendes Gebäude
- Eine Menschenmenge muss aus wenigen Ausgängen evakuiert werden
- Unsystematische Evakuierung
  - Menschen geraten in Panik
  - Chaos bricht aus und nur ein Bruchteil der Menge schafft es
- Organisierte Evakuierung
  - Menge teilt sich in einzelne Gruppen auf
  - Jede Gruppe läuft geordnet zum nächsten Ausgang
- Bei uns
  - Das brennende Gebäude entspricht der Festplatte
  - Die Menschen sind die einzelnen Dateien

Resource Cache

- Lädt angeforderte Daten aus dem Sekundärspeicher (bspw. Festplatte / DVD-Laufwerk) in den Primärspeicher (Arbeitsspeicher)
- Verwaltet die Daten des Primärspeichers
  - Lädt Daten nach
  - Gibt Daten wieder frei
  - Lädt Daten vor
- Hat eine bestimmte Größe an Speicher zur Verfügung
- Von jeder geladenen Resource ist bekannt
  - Wird sie zurzeit benutzt?
  - Wenn nein, wann wurde sie zuletzt benutzt?

Resource Cache

- Szenario: Anforderung einer Datei / Resource



Resource Cache

- Bei unserer Engine Dateien anfordern
  - `GameApp.getResCache().getHandle(new Resource(„pathToFile“));`
  - Wartet, bis die Datei geladen wurde
- Daten einer Datei werden in einem `ResHandle` gekapselt
  - `byte[] getBuffer()` – Liefert die Daten als Bytes
  - `InputStream getBufferAsInputStream()` – Liefert die Daten als Stream
  - `void unlock()` – Gibt die Daten zum Löschen frei

Unser Resource Cache

- Resource Cache parallelisiert
- Dateien werden im Hintergrund im eigenen Thread geladen
- Dateien vorladen lassen
  - `GameApp.getResCache().addRequest(new Resource(„pathToFile“));`
  - Wartet nicht, bis die Datei geladen wurde

Nächste Woche voraussichtlich:

- Game View
- GUI
- Mathematische Grundlagen

Vielen Dank für die Aufmerksamkeit ☺