Aufbau interaktiver 3D-Engines

Universität Osnabrück
Fachbereich Mathematik / Informatik

2. Vorlesung (15.04.2013)

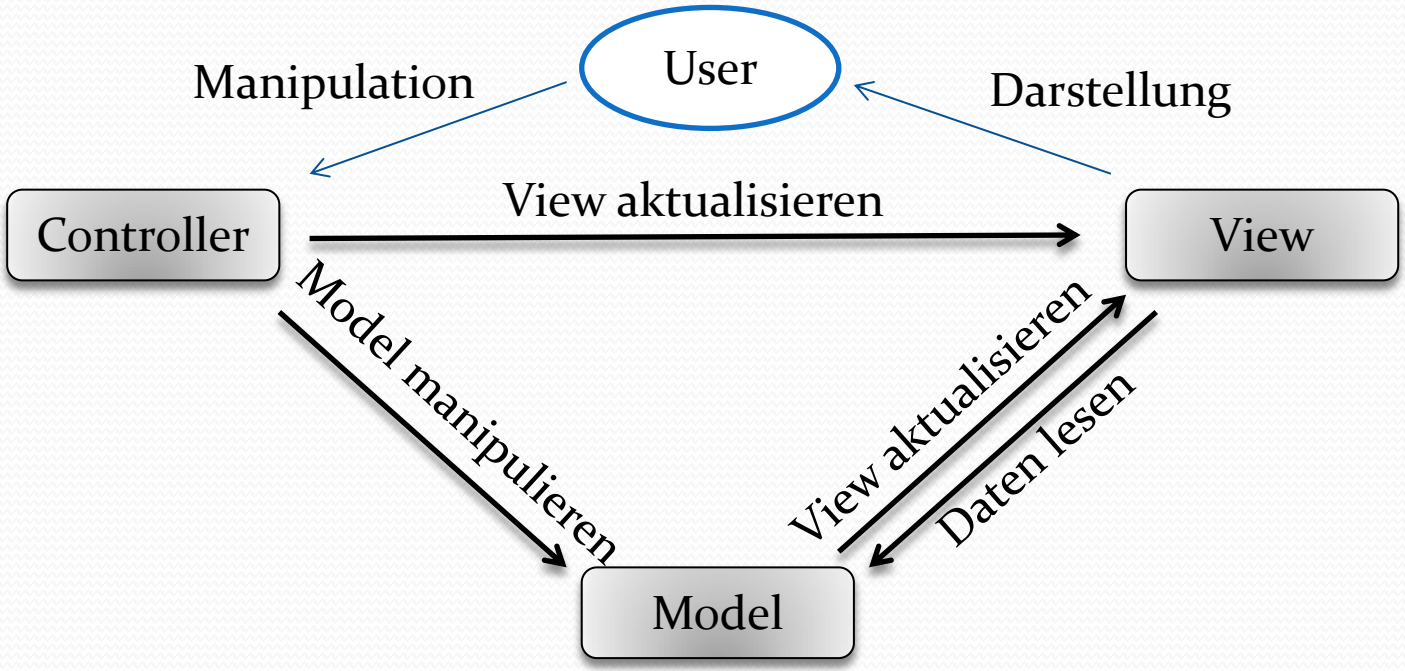
Prof. Dr. rer. nat. Oliver Vornberger
Nico Marniok, B. Sc.
Erik Wittkorn, B. Sc.

Aufbau einer Engine

- Viele Möglichkeiten, keine ist „die Richtige“
- Wahl aus Game Coding Complete zielt auf Verständlichkeit
- Verständnis *muss* vorhanden sein
  - Welche Komponenten gibt es?
  - Wo gehören neue Komponenten hin?
  - Wie fügt sich alles zusammen?
- Klare Struktur von immensem Wert
- Wie bei allen größeren Programmierprojekten gilt
  - Erst hinsetzen und nachdenken!

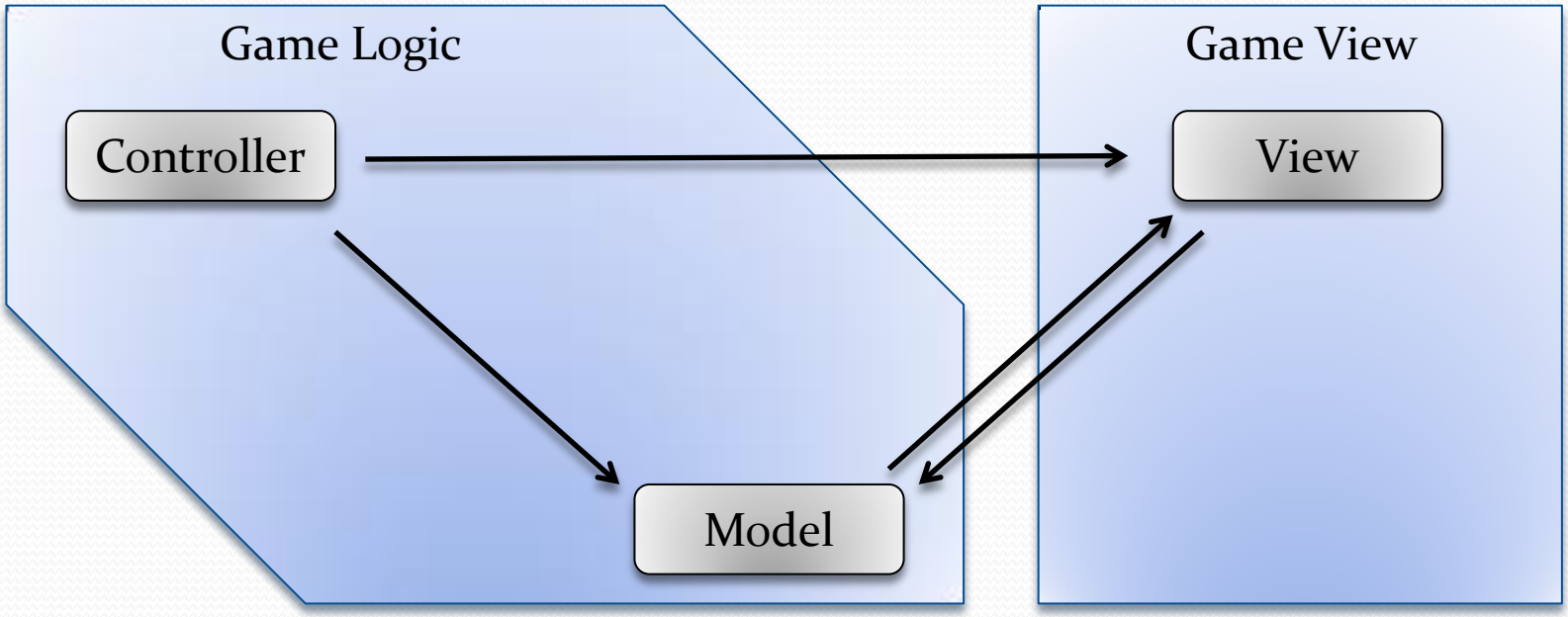
Model-View-Controller (MVC)

- Aus Informatik B bekanntes Designpattern
- Ursprüngliches MVC-Muster
  - G. E. Krasner, S. T. Pope [KP88]: “Model-View-Controller in Smalltalk-80



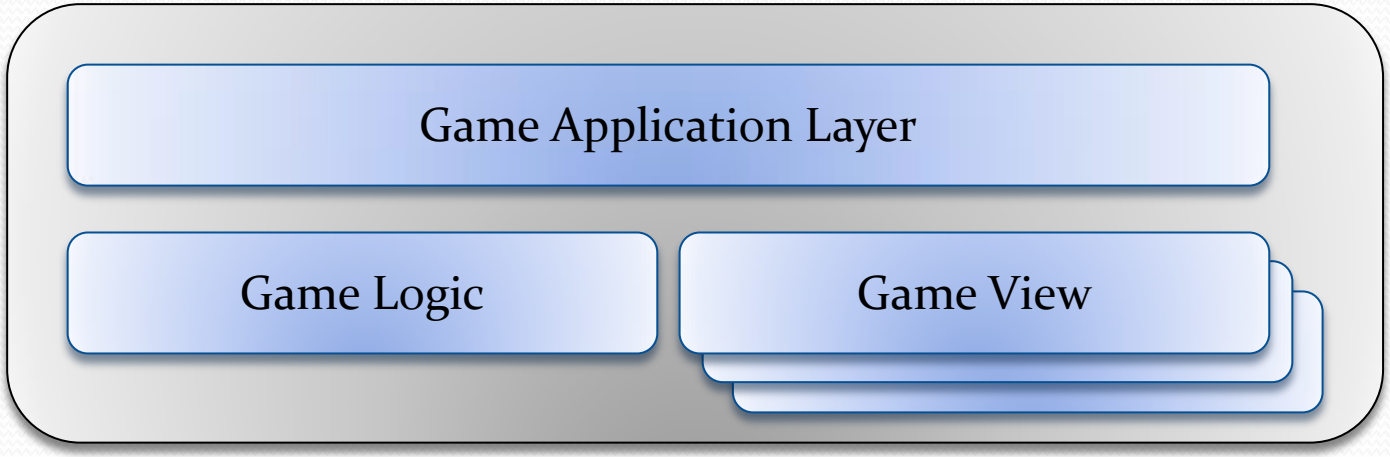
Aufbau einer Engine

- Controller und Model
  - Spiegelt die Logik hinter dem Spiel wieder
  - Was passiert wann in der Engine?
  - Zusammenfassen zu *Game Logic*



Grundsätzlicher Aufbau

- Einführung der zusätzlichen Komponente Game Application Layer
- Jeder Teil der Engine gehört zu einer von drei Komponenten



Kurz: Game Application Layer

- Initialisierungen und systemnahe Implementationen
  - Direct3D / OpenGL
  - Input lesen
  - Sound
  - Netzwerkkommunikation
  - Timer
  - Laden und Verwaltung von Resourcen
  - …
- Im Idealfall nur hier Veränderungen bei Portierungen

Kurz: Game Logic

- Logik und Simulation der Spielwelt
  - Actorverwaltung
  - Levelverwaltung
  - Physiksimulation
- Aufgaben des „Controllers“
  - Viewverwaltung
  - Umsetzung von Userinput
- Keine direkte Manipulation
- Kommunikation über Events

Kurz: Game View

- Interface zwischen Benutzer und Spielwelt
  - Grafikausgabe
  - Soundausgabe
  - Input in Spielbefehle übersetzen
- Interface zwischen KI und Spielwelt
  - KI hat Zugriff auf die gleichen Informationen, wie der Benutzer und sendet die gleichen Befehle

Game Application Layer

- Einstiegspunkt für unsere Projekte: `GameApp.java`
- Drei essentielle Methoden
  - `public static boolean init()`
  - `public static void mainloop()`
  - `public static void destroy()`
- Grundsätzlicher Aufbau der `main()` Methode:

```
if(GameApp.init()) {
    GameApp.mainLoop();
    GameApp.destroy();
} else {
    System.err.println("Something went horribly wrong!");
}
```

boolean init()

- Initialisierung der einzelnen hardwarenahen Komponenten

```
if(!component1.init()) {
    Logger.INSTANCE.error("Failed to initialize component 1!");
    return false;
}
if(!component2.init()) {
    Logger.INSTANCE.error("Failed to initialize component 2!");
    return false;
}
return true;
```

- Nur wenn alles geklappt hat

```
return true;
```

void destroy()

- Immer aufräumen!
- Zerstörung der einzelnen hardwarenahen Komponenten *in umgekehrter Reihenfolge*!

```
public static void destroy() {
    component2.destroy();
    component1.destroy();
}
```

void mainloop()

- Die meisten Programme
  - reagieren nur auf Input des Benutzers
  - machen nichts, solange kein Input da ist
- Komplexe Spiele befinden sich ständig in der Simulation
- Beispiele
  - Physik wird weiter berechnet
  - KI sucht sich einen Weg zum Spieler um ihn zu attackieren
  - GUI Elemente sind evtl. animiert

void mainloop()

- Endlosschleife
  - Aktualisiert laufend die Spielwelt
  - Liest Inputdevices
  - Gibt den Anstoß zum Rendern (Grafik / Sound)
- Anzahl der Schleifeniterationen pro Sekunde = FPS (Frames per Second / Bildwiederholrate)
- < 20 FPS (im laufenden Spiel) auf keinen Fall akzeptabel
- > 60 FPS (im laufenden Spiel) anzustreben
- Achtung: beschränkt durch Monitor
- Berechnungen innerhalb einer Iteration dürfen nicht „ewig“ dauern

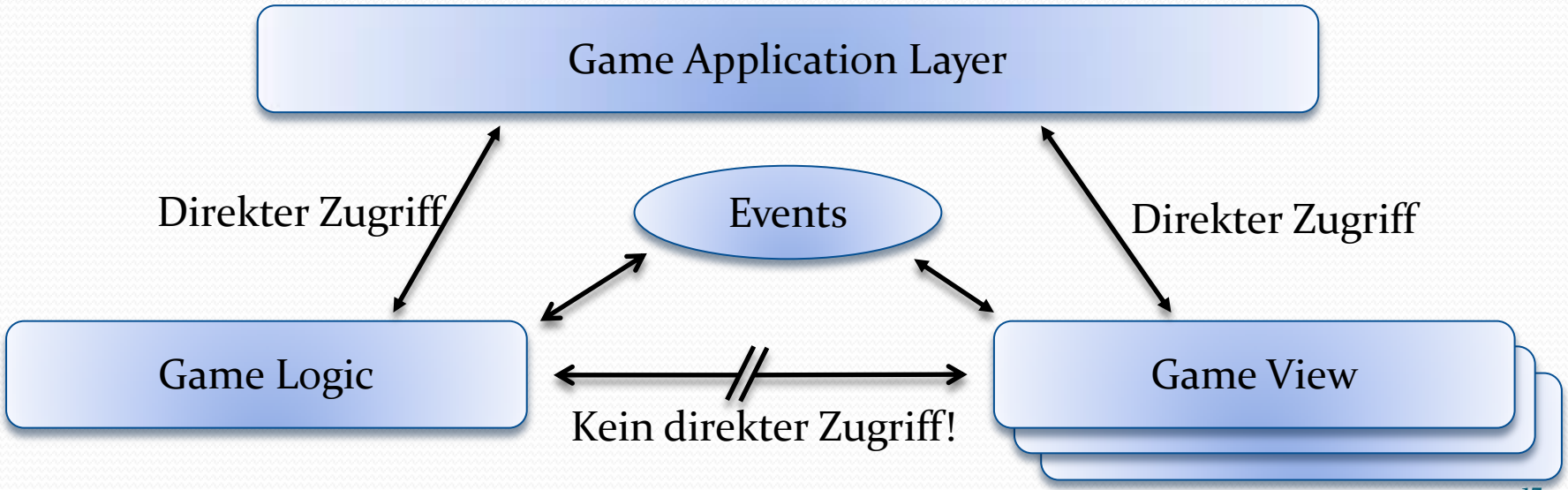
Dauer einer Iteration	Bildwiederholrate
50ms	20 FPS
17ms	60 FPS
8ms	120 FPS

void mainloop()

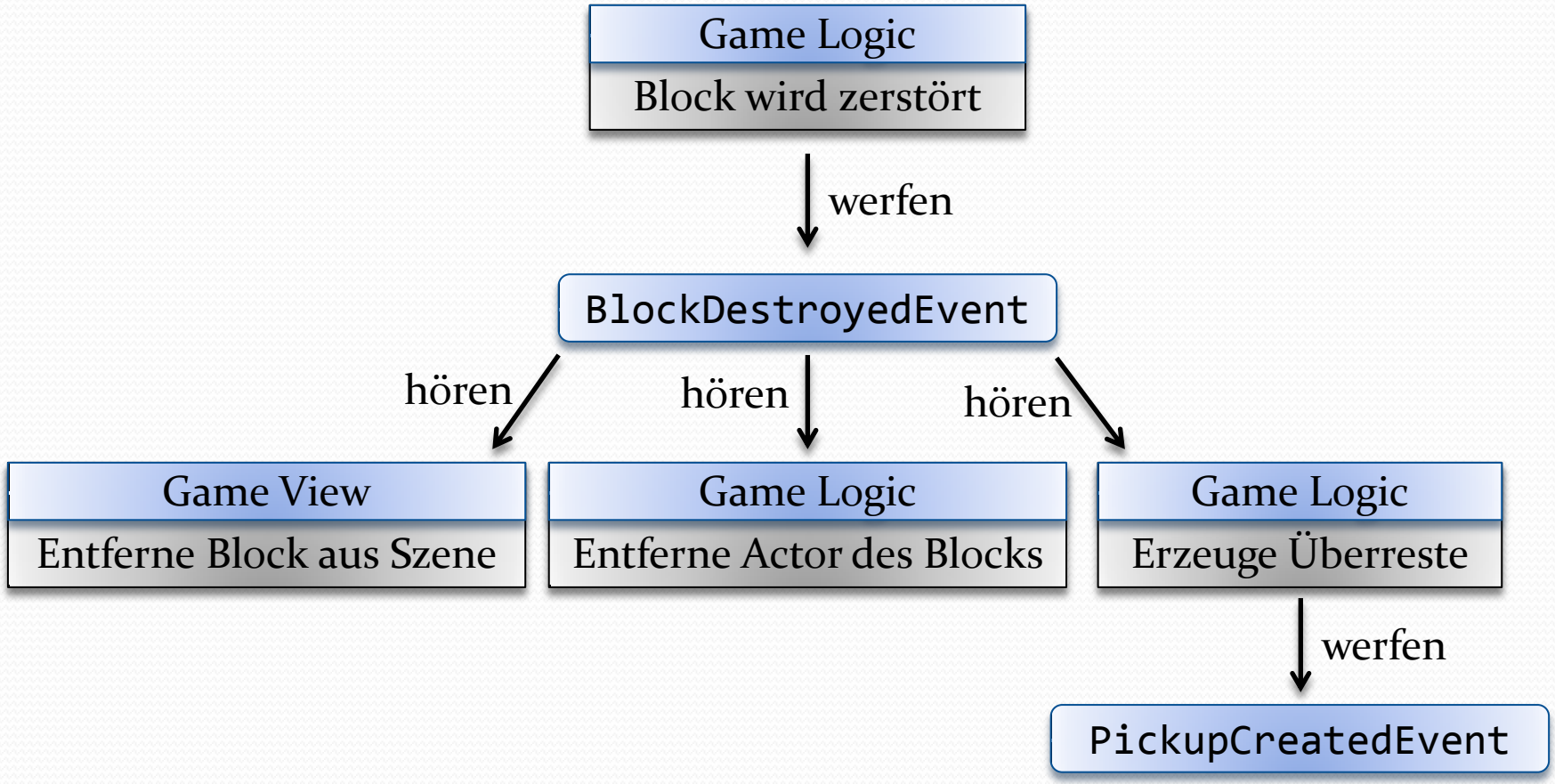
```
while(bContinue && !graphicsLayer.isWindowCloseRequested()) {
    graphicsLayer.clear();
    gameTimer.next();
    realTimer.next();
    inputHandler.processInput();
    eventManager.processEvents();
    physicsLayer.update(gameTimer.getDelta());
    globalProcessManager.update(gameTimer.getDelta());
    gameLogic.update(gameTimer.getDelta());
    gameLogic.render();
    graphicsLayer.update(realTimer.getDelta());
    graphicsLayer.setWindowTitle("FPS: " + graphicsLayer.getFps());
}
```

Kommunikation

- In der Regel direkter Zugriff auf Game Application Layer
- Kommunikation zwischen Logic und View über Events



Beispielszenario: Minecraft

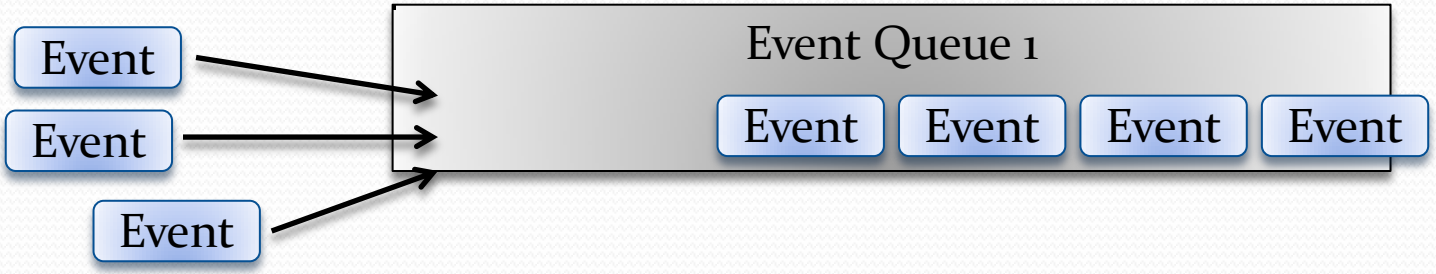


Events

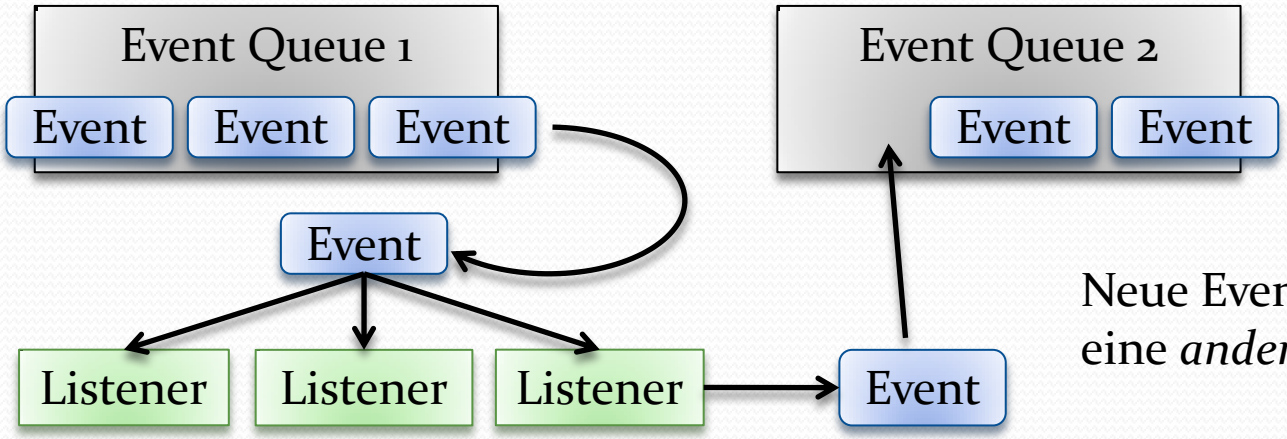
- Geschehnisse in der Spielwelt erzeugen Events
- Komponenten der Engine hören und reagieren auf Events (EventListeners)
- Verarbeiten von Events kann neue Events erzeugen
- Unendliche Eventkette? Möglich!
- Erzeugte Events werden gesammelt und erst *im nächsten Frame* verarbeitet

Event Queue

Während der Simulation erzeugte Events werden in eine Event Queue eingereiht:



Nach Abschluss der Simulation werden die Events sequentiell „abgearbeitet“:



Neue Events werden hierbei in eine *andere* Queue eingereiht

EventManager

- Verwaltung der Event Queues und der Listener
- Eine aktive und eine inaktive Queue (Double buffered)
- Events sammeln
  - Events werden immer in die aktive Queue eingereiht
- Events verarbeiten
  - Tauschen von aktiver und inaktiver Queue
  - Sequentiell die Elemente aus der inaktiven Queue entfernen und verarbeiten

EventData

- Abstrakte Basisklasse aller Events
- Eine einzige zu implementierende Methode
  - `public abstract int` **`getId()`**
  - Liefert eine ID, die diesen Event*typ* eindeutig identifiziert (GUID: Globally unique identifier)
- Beispiel: `QuitEvent.java`

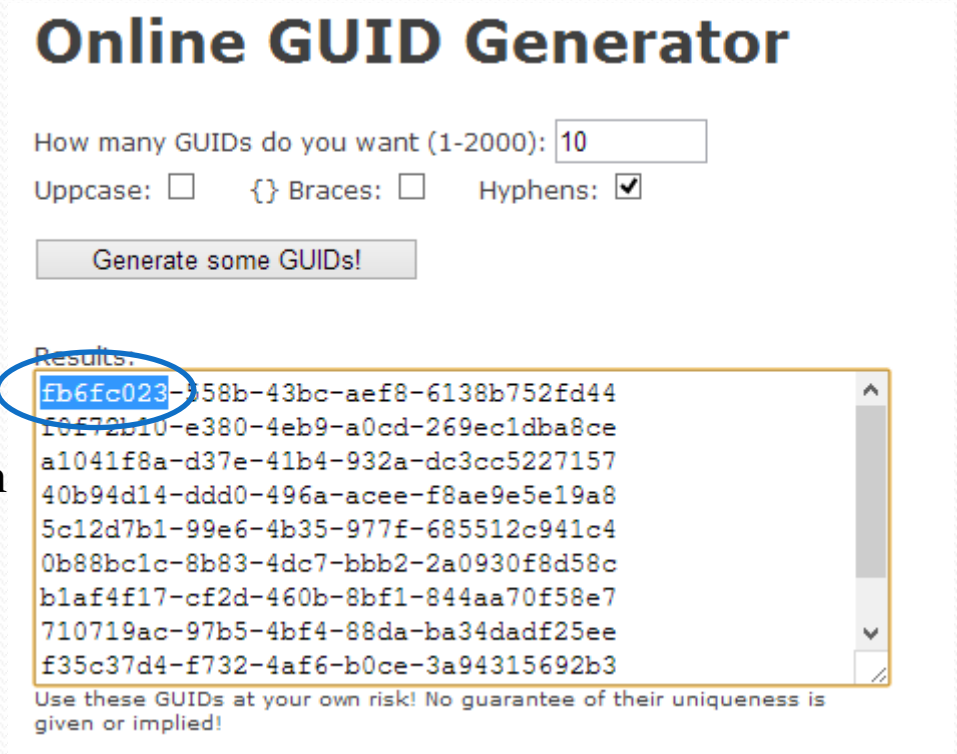
```
public class QuitEvent extends EventData {
    public static final int ID = 0x23a1ef58;

    @Override
    public int getId() {
        return ID;
    }
}
```

GUID

- `public static final int ID = 0x23a1ef58;`
- Jeder Event*typ* hat eine eigene ID
  - ***Nicht*** jedes Event!
- Woher nehmen?
  - Blind auf Tastatur einschlagen
  - Eleganter: http://www.guidgenerator.com/

GUID

Kopieren und in Javadatei mit `0x` beginnend einfügen



EventListener

- Interface mit einer einzigen zu implementierenden Methode
  - `public void trigger(EventData data);`
  - Wird vom EventManager aufgerufen, sobald ein Event verarbeitet wird
- Listener beim EventManager registrieren
  - `GameApp.getEventManager().register(this, QuitEvent.ID);`
  - Registrieren um auf Events mit einer bestimmten ID zu hören

Typische Implementation

```
@Override
public void trigger(EventData data) {
    switch(data.getId()) {
        case QuitEvent.ID:
            GameApp.bContinue = false;
            break;
        case ActorMovedEvent.ID:
            ActorMovedEvent evt = (ActorMovedEvent)data;
            this.actorMoved(evt.getActorId());
            break;
    }
}
```

Events erzeugen

- Instanz des Events ganz normal erzeugen
  - `EventData data = new QuitEvent();`
- Überlegung
  - Soll Event direkt verarbeitet werden? (Manchmal nötig)
    - `GameApp.getEventManager().triggerEvent(data);`
  - Soll das Event in die Queue eingereiht werden?
    - `GameApp.getEventManager().queueEvent(data);`

Typische Events

Event	In unserer Engine vorhanden
ActorMovedEvent	X
ActorDeletedEvent	X
QuitEvent	X
PlayerActorChangedEvent	X
LevelLoadedEvent	
PlayerDiedEvent	
ExplosionEvent	

Coole Features

- Spielgeschehen aufnehmen?
  - Auf „alle“ Events hören
  - Events in eigener Queue sammeln
  - Queue später „abspielen“, indem jedes Event daraus wieder geworfen wird
  - Mögliche Anwendung: Rennspiel Replayfunktion
- Netzwerkfunktionen?
  - Events in kleinen Bündeln sammeln und über Netzwerk schicken
  - GameLogic anpassen!

Game Logic

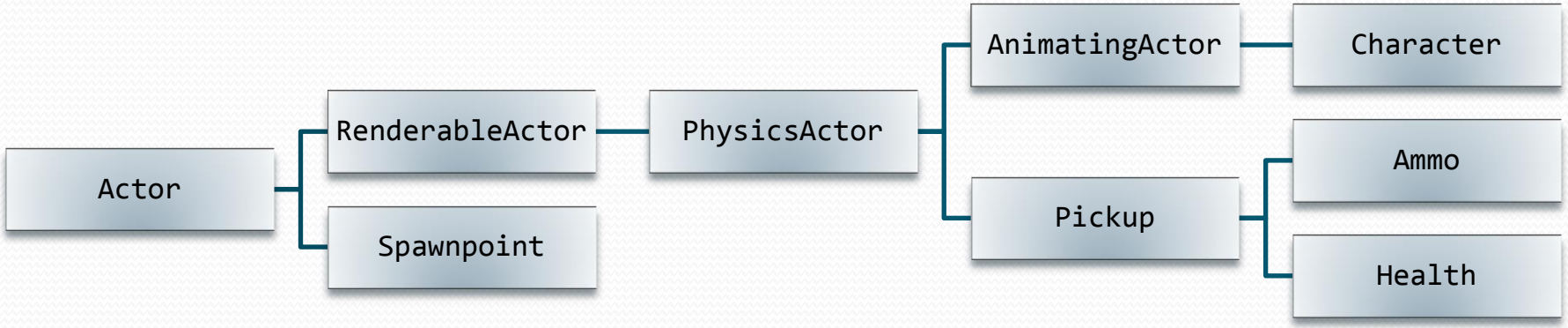
- „The Game Logic is the heart and soul of your game.“
  - Mike McShaffry, Game Coding Complete
- Definiert das Spieluniversum
- Reaktion auf externe Einwirkungen
  - Spieler macht Eingaben
  - KI wird aktiv
- Interaktionen zwischen *Dingen* der Spielwelt
- Actorverwaltung

Actors, Aktoren, Akteure

- Spiele sind voll von Objekten
  - Raumschiffe und Planeten in Weltraumsimulationen
  - Massenhaft Gegner und Pickups in Egoshootern
  - Autos und Straßenbestandteile in Rennspielen
  - 15 kleine Quader in „FastWorx TileGame“
  - 26 kleine Würfel in „Nicotopia RubiksCube“
- Wir nennen diese Objekte *Actors*

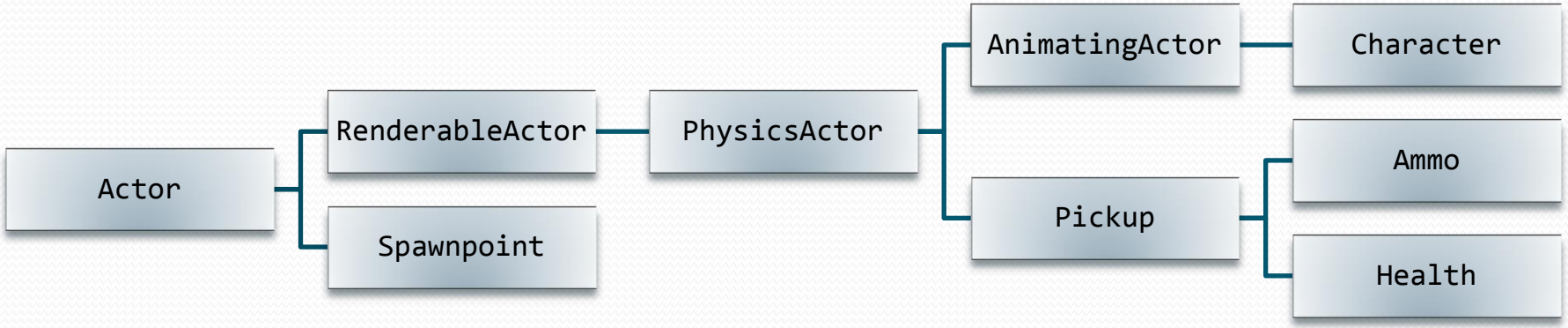
Actors und Objektorientierung

- Glücklicherweise haben wir Objektorientierung ☺
- Basisklasse Actor
- Besonderer Actor: graphisch darstellbarer
- Evtl. mit Physik
- Pickups sind in der Regel nicht animiert



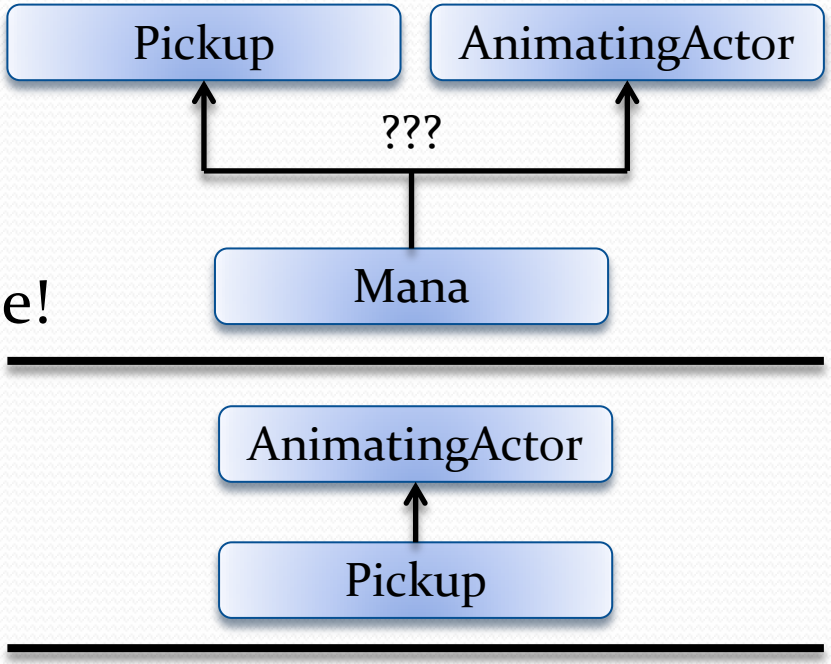
Actors und Objektorientierung

- Szenario
  - Halbes Spiel bereits auf diese Weise implementiert
  - Chef: „Wir brauchen noch eine wabernde blaue Wolke als Manapickup!“
  - Also: animiertes Pickup



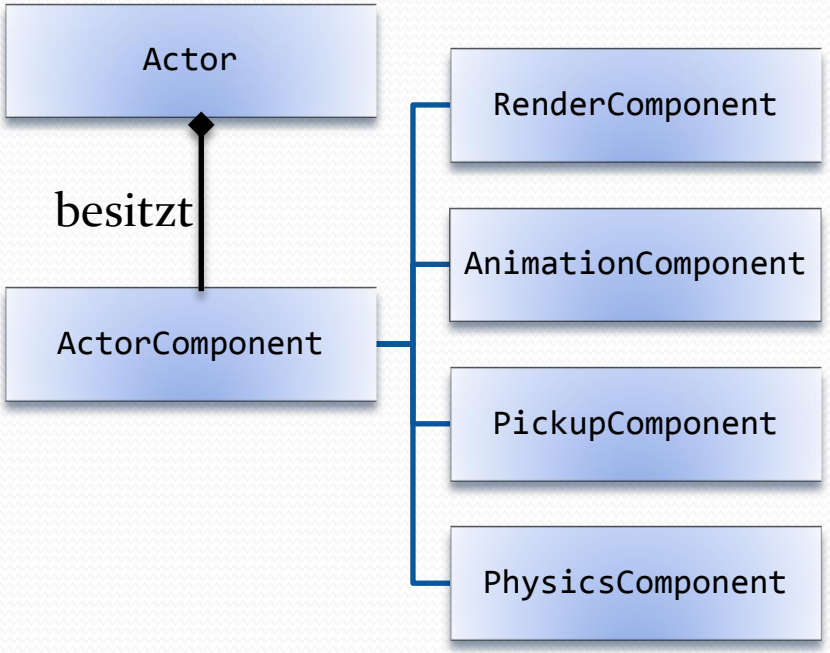
Animiertes Pickup

- Mehrfachvererbung?
  - In Java nicht möglich
  - Und auch sonst keine gute Idee!
- Umstrukturieren?
  - Kostet viel Zeit und Arbeit
  - Statische Pickups?
- Animationscode kopieren?
  - Code Kopieren ist niemals eine gute Idee!
- Animierter Actor über jedes Manapickup im Level?
  - Wird schnell vergessen



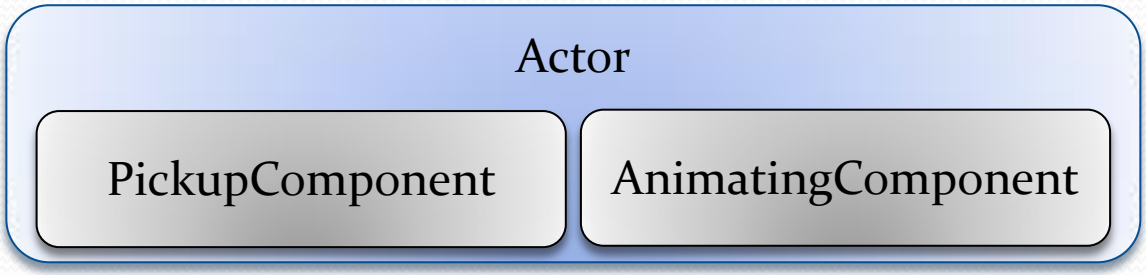
Actors und Objektorientierung

- Glücklicherweise haben wir Objektorientierung und wissen damit umzugehen ☺
- Eigenschaften eines Actors als Komponenten
- Actor darstellbar?
  - RenderComponent
- Actor einsammelbar?
  - PickupComponent
- Actor physikalisch?
  - PhysicsComponent



ActorComponent

- Jede Eigenschaft entspricht einer ActorComponent
- Animiertes Pickup jetzt sehr simpel:



- Weitere Vorteile
  - Neue Actortypen schnell zu implementieren
  - Actor leicht mit xml zu beschreiben

Cube.xml

```
<Actor>
    <TransformComponent />
    <PhysicsComponent>
        <Box>
            <Size x="2.0" y="2.0" z="2.0" />
            <Density value="10.0" />
        </Box>
    </PhysicsComponent>
    <RenderComponent>
        <ObjMesh resource="meshes/cube.obj" />
        <BoundingSphere>
            <Radius value="1.75" />
        </BoundingSphere>
    </RenderComponent>
</Actor>
```

Implementationsdetails

- Parsen der xml-Datei liefert String „`TransformComponent`“
- *Zur Laufzeit* zu tun
  - Finde Klasse anhand ihres Namens
  - Erzeuge Instanz
  - Füge Klasse dem Actor hinzu
- An anderer Stelle zu tun
  - Hole TransformComponent von einem Actor

Implementationsdetails

- Lösung: Java Reflections

```
String componentName = "components.TransformComponent";
Class compClass = null;
try {
    // get class object by name
    compClass = Class.forName(componentName);
} catch (ClassNotFoundException ex) {
    // invalid component
    Logger.INSTANCE.error("Invalid component: " + componentName);
}
if(compClass != null) {
    Object instance = null;
    try {
        // try to instanciate new instance
        instance = compClass.newInstance();
    } catch (InstantiationException | IllegalAccessException ex) {
        // some serious error
        Logger.INSTANCE.error(ex.getMessage());
    }
    // checking super class
    if(instance instanceof ActorComponent) {
        return (ActorComponent)instance;
    }
}
```

ActorComponent.java

- Abstrakte Klasse mit zu implementierenden Methoden
  - `protected abstract int getComponentId();`
    - Wie bei Events: GUID des Komponententyps
  - `protected abstract boolean init(Node data);`
    - Initialisierung mithilfe eines DOM Knotens

ActorComponent.java

- Weitere Methoden, die mit Inhalt gefüllt werden können
  - `protected void postInit() {}`
    - Wird nach Initialisierung aller Komponenten eines Actors aufgerufen
  - `protected void update(long deltaMillis) {}`
    - Wird jeden Frame aufgerufen
  - `protected void destroy() {}`
    - Wird bei Zerstörung des Actors aufgerufen

Anhang: packages

- Standardmäßige Suche nach ActorComponents im Package `actor.components`
- Andere Packages bei Initialisierung der Engine sichtbar machen
  - `ActorComponent.addComponentPackage(„other.package“);`

Actor.java

- Im Grunde nur noch Container für ActorComponents
- Wichtige Methoden
  - `public int getId()`
    - Liefert eindeutige ID des Actors
  - `public String getName()`
    - Liefert lesbaren Namen des Actors (nicht zwangsweise eindeutig!)
  - `public String getParentActorName()`
    - Liefert Namen des Vateractors (falls vorhanden)
  - `public ActorComponent getComponent(int id)`
    - Liefert die Komponente anhand der GUID

ActorComponents

ActorComponent	Funktion
`TransformComponent`	Actor besitzt Position und Ausrichtung im Raum; siehe Klasse `Pose.java`
`PhysicsComponent`	Actor ist dem Physiksystem bekannt und hat zu berücksichtigende Eigenschaften
`RenderComponent`	Actor kann von der HumanView grafisch dargestellt werden und ist Teil des Szenegraphen
`CameraComponent`	Actor kann als Kamera verwendet werden; es muss *mindestens einen* Actor mit dieser Komponente geben

Nochmal: Game Logic

- Jeder neue Actor wird automatisch zur Game Logic hinzugefügt
- Nützliche Methoden der Game Logic
  - `public Actor createActor(String resource)`
    - Erzeugt einen neuen Actor aus einer xml-Datei
  - `public void deleteActor(int id)`
    - Löscht einen Actor anhand seiner ID
  - `public Actor getActor(int id)`
    - Liefert einen bereits erzeugten Actor anhand seiner ID
  - `public Actor getActor(String name)`
    - Liefert einen bereits erzeugten Actor anhand seines Namens

Game Logic

- Für das eigene Projekt muss die Klasse `BaseGameLogic.java` erweitert werden
- Einzige zu implementierende Methode
  - `public abstract void loadLevel(String resource);`
    - Lädt den Level aus einer Datei
- Nochmal Aufbau der ***main***() Methode

```
GameApp.setGameLogic(new MyGameLogic());
if(GameApp.init()) {
    GameApp.mainLoop();
    GameApp.destroy();
} else {
    System.err.println("Something went horribly wrong!");
}
```

Nächste Woche voraussichtlich:

- Prozesse vs. Events
- GameApp: Timer
- GameApp: Input lesen
- GameView: Input verarbeiten
- GameApp: Resourcesystem
  - RAM / VRAM
  - Dateien einlesen
  - Dateien cachen

Vielen Dank für die Aufmerksamkeit ☺