Aufbau interaktiver 3D-Engines

Universität Osnabrück
Fachbereich Mathematik / Informatik

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

Organisatorisches

- 1 Vorlesung + 1 Übung, 6 ECTS, zählt als *Praktikum*!
- Vorlesung: Montags, 12:00 – 14:00, regelmäßige Teilnahme erwünscht!
  - Gehalten von Nico Marniok
- Übung: Dienstags, 12:00 – 14:00, regelmäßige Teilnahme erwünscht!
  - Gehalten von Erik Wittkorn
  - Erste Übung am 09.04.2013!
- Testate alle 2 Wochen in 2er Teams
  - Ausgabe Übungsaufgaben: Alle 2 Wochen dienstags, ab 09.04.2013 (1. Übung)
  - Abgabe Lösungen: Alle 2 Wochen freitags, ab 19.04.2013, 23:59
  - Vorstellung einer Lösung alle 2 Wochen dienstags in der Übung ab 23.04.2013
- Größeres Projekt am Ende der Vorlesungszeit über 2 Wochen in 2er Teams
  - Zwischenpräsentation nach einer Woche vor Erik und Nico
  - Endpräsentation nach zwei Wochen vor Herrn Vornberger
  - Tipp: Bereitet die Endpräsentation *sehr gut* vor und haltet die zur Verfügung stehende Zeit *unbedingt* ein!

Organisatorisches

- Voraussetzungen
  - Informatik A
  - Computergrafik
    - Wird erst in der zweiten Hälfte relevant
  - Empfohlen: Informatik B, Grundwissen Lineare Algebra (Matrizen- und Vektorrechnung im R^4)
- Literatur
  - Game Coding Complete, Fourth Edition
    - Mike McShaffry and David "Rez" Graham
    - Course Technology, 2012
- Software
  - NetBeans http://www.netbeans.org/
  - eclipse http://www.eclipse.org
  - LWJGL http://www.lwjgl.org/
  - Luaj http://luaj.org/luaj/README.html
  - jBullet http://jbullet.advel.cz/

Game-Engines / 3D-Engines

- „A game engine is a system designed for the creation and development of video games.”
  - Wikipedia, Game engine, 23.03.2013

Beispiel: Gamebryo / Creation Engine

The Elder Scrolls V: Skyrim
(Bethesda Softworks)

Fallout New Vegas
(Bethesda Softwors / Obsidian)

Beispiel: CryEngine 1 / 2 / 3

Crysis (Crytek)

Crysis 3 (Crytek)

Beispiel: Blender

Blick auf eine Engine: Endnutzer

- Der Benutzer stellt an interaktive Programme mit Grafikausgabe „sehr hohe“ Erwartungen, z.B.
  - Die Ausgabe soll in Echtzeit und mit einer stabilen Bildwiederholrate geschehen
  - Das Programm soll stabil gegenüber chaotischen oder sehr schnellen Eingaben sein
    - Reihenfolge der Eingaben muss beachtet werden
    - Es sollen keine Eingaben verloren gehen
  - Die Ausgabe soll den Eingaben entsprechen

Blick auf eine Engine: Endnutzer

- Warum sind diese Erwartungen „sehr hoch“?
  - Eine Vielzahl von Systemen müssen kooperativ arbeiten
    - Input lesen
    - Grafik- / Soundausgabe
    - Spielwelt simulieren
    - Physik berechnen
    - Daten laden
    - …
  - Berechnungen während der Ausführung dürfen nicht „ewig“ dauern
    - Anmerkung: Ewig = „Zeitspanne > 50ms“

Anforderungen vom Entwickler

- Engine soll übersichtlich sein
- Komponenten sollen möglichst voneinander getrennt sein
  - Stichwort: Arbeiten im Team
- Leicht zu erweitern
- Engine soll für viele Projekte wiederverwendbar sein
- Plattformübergreifende Entwicklung
- Kompilieren nach kleinen Änderungen soll nicht „ewig“ dauern
  - Anmerkung: Ewig = „Zeitspanne > 30sec“

Implementation eines Spielelements

- Was soll passieren? (Kurze Beschreibung des Features)
- Wann soll es passieren? (Unmittelbarer Auslöser)
- Was sind die Voraussetzungen? (Mittelbare Auslöser)
- Was sind die sicheren Folgen?
- Was sind die möglichen Folgen?

Beispiel: Minecraft

- Was
  - Block wird angegraben
- Wann
  - Spieler benutzt Werkzeug (oder leere Hände)
- Voraussetzungen
  - Spieler blickt auf Block
  - Block ist abbaubar
- Direkte Folgen
  - Sound abspielen
  - Animation abspielen
  - Schaden des Blocks neu berechnen und setzen
- Mögliche Folgen
  - Schadenstextur des Blocks aktualisieren
  - Block zerstören

Beispiel: Minecraft

- Genauer: *Block zerstören* bedeutet
  - Block aus Grafiksystem entfernen
  - Block aus Spielelogik entfernen
  - Block aus Physiksystem entfernen
  - Evtl. Überreste generieren
- Spielelemente, mit ähnlichen Folgen
  - TNT zünden
  - Block per Skript entfernen (Adventuremodus / Mods)

Beispiel: Minecraft

- Merke
  - Spielelemente neigen zu
    - wenigen direkten Folgen
    - vielen möglichen Folgen
  - Verschiedene Spielelemente haben gleiche Folgen
- Resultierende direkte Implementation
  - Umfangreiche Methode, die alle Spielelemente durchgeht und Ursachen / Voraussetzungen prüft
  - Viele bedingte Ausdrücke
  - Doppelter Code oder viele kleine Methoden mit „Folgenclustern“

Beispiel: Minecraft

- Direkte Implementation

```
private void update() {
    Block b = player.getAimedBlock();
    if(Mouse.isButtonDown(0) && b != null) {
        Item item = player.getItem();
        float progress = item.getAnimationProgress(HIT_ID);
        if(progress == 0.0) {
            GraphicsSystem.startAnimation(item, HIT_ID);
            SoundSystem.startSound(item, HIT_ID);
        }
        int damage = item.getDamage();
        int deltaDamage = 1;
        switch(item.getId()) {
            case PICKAXE_DIAMOND_ID: deltaDamage = 20; break;
            case PICKAXE_WOOD_ID: deltaDamage = 2; break;
            // ...
        }
        damage += deltaDamage;
        if(damage > 100) {
            Scene.remove(b);
            Scene.create(b.getRemainingsForTool(item));
        } else {
            b.setDamage(damage);
            GraphicsSystem.setTexture(item, ResourceSystem.getDamageTexture(damage));
        }
    }
}
```

Beispiel: Minecraft

- Vorteile der Methode `update()`
  - Man sieht auf einen Blick, was alles passiert
- Nachteile
  - Wird extrem lang
    - Übersicht beginnt schnell zu leiden
    - Viele Entwickler arbeiten an derselben Datei
  - Referenziert viele andere Systeme (Grafik, Sound, Ressourcen, …)
    - Dadurch muss die Datei selbst neu kompiliert werden, wenn sich ein einziges dieser Systeme ändert oder erweitert wird
  - Alle Aktionen werden auf einmal durchgeführt
    - Dadurch kann es passieren, dass das Spiel kurz stockt
  - Mehr Nachteile werden schnell ersichtlich, wenn man das ganze Spiel auf diese Weise programmiert

Besser: Eventgesteuerte Engine

- Aktionen des Spielers erzeugen Events
  - Beispiel: `StartHittingEvent`
- Spielwelt erzeugt Events
  - Beispiel: `WeatherChangedEvent`
- KI erzeugt Events
  - Beispiel: `CreeperExplodedEvent`
- Kurz: Alles erzeugt Events

Events

- Komponenten der Engine können auf verschiedenste Events hören
  - Beispiel: Grafiksystem hört auf
    - `BlockDamageChangedEvent`
    - `BlockDestroyedEvent`
    - `WeatherChangedEvent`
    - `ExplosionEvent`
    - …
- Trennung von Ursache und Folge
- Wichtig ist nur, *dass* etwas passiert ist; nicht *wie*

Beispiel: BlockDestroyedEvent

```
public void init() {
  EventManager.registerListener(this, BlockDestroyedEvent.ID);
  // ...
}

@Override
public void trigger(EventData data) {
  switch(data.getId()) {
    case BlockDestroyedEvent.ID:
      BlockDestroyedEvent evt = (BlockDestroyedEvent)data;
      this.removeBlock(evt.getBlockId());
      break;
    // ...
  }
}
```

Eventgesteuerte Engine

- Neue Spielelemente hinzufügen
  - Was soll passieren? (Kurze Beschreibung des Features)
  - Wann soll es passieren? (Unmittelbarer Auslöser)
  - Was sind die Voraussetzungen? (Mittelbare Auslöser)
  - Welche Events werden geworfen?
- Neue Komponenten hinzufügen
  - Wie wird die Komponente beeinflusst (Auf welche Events hört sie?)
- Mehr dazu: nächste Woche

Wichtige Grundlagen: XML

- XML = e**X**tensible **M**arkup **L**anguage
- Geschichtlicher Überblick
  - Siehe Wikipedia
- Mit XML lässt sich alles Mögliche beschreiben
- Lässt sich sehr gut parsen *und* ist gut lesbar für den Menschen
- Eignet sich besonders gut, um Levels, GUIs, Actors usw. zu beschreiben

XML Aufbau: Markup vs. Content

- Inhalt als Klartext (Strings)
- Zwei Grundtypen
  - Markup
    - Strings, die mit < beginnen und mit > enden
    - Strings, die mit & beginnen und mit ; enden (unwichtig)
  - Content: alles, was kein Markup ist
- Beispiel

```
<RectShape>
  <Position x=„4“ y=„3“ />
  <Size width=„2“ height=„7“ />
  <Color>Yellow</Color>
</RectShape>
```

XML Aufbau: Tags

- Tags
  - Texteinheit, die mit < beginnt und mit > endet, ist ein Tag
  - Man unterscheidet 3 Typen
    - Start-tag: Inhalt beginnt und endet mit einem Buchstaben
    - End-tag: Inhalt beginnt mit einem Slash /
    - Empty-element-tag: Inhalt beginnt mit einem Buchstaben und endet mit einem Slash /
- Beispiel

```
<RectShape>
  <Position x=„4“ y=„3“ />
  <Size width=„2“ height=„7“ />
  <Color>Yellow</Color>
</RectShape>
```

XML Aufbau: Attribute

- Attribute
  - Ein Konstrukt der Form `name=„value“` innerhalb eines Tags
    - `name` darf nur *anständige* Zeichen enthalten
    - `value` darf beliebige Zeichen außer doppelte Anführungszeichen enthalten
      - Maskieren mittels Backslash (`\“`) funktioniert *nicht*!
- Beispiel

```
<RectShape>
  <Position x=„4“ y=„3“ />
  <Size width=„2“ height=„7“ />
  <Color>Yellow</Color>
</RectShape>
```

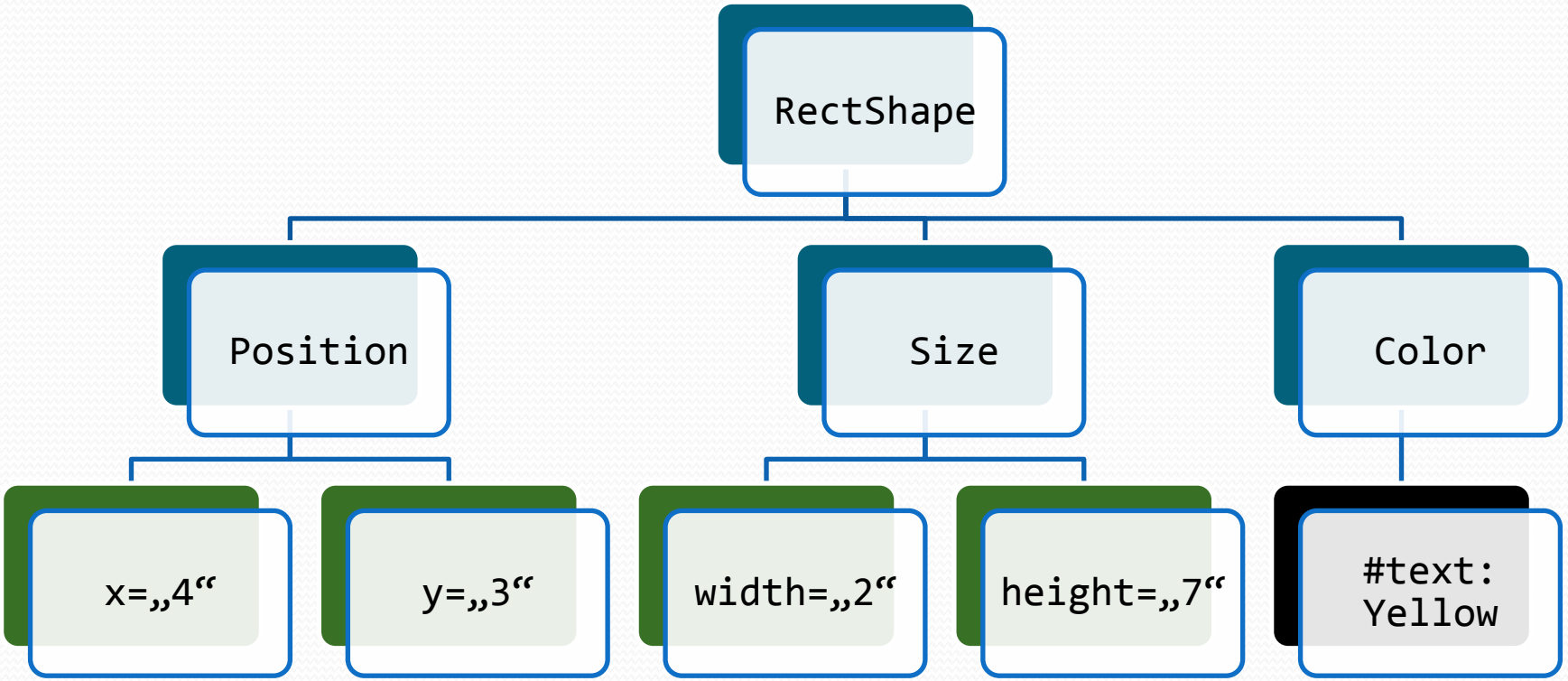
XML Aufbau: Elemente

- Elemente, 2 verschiedene Typen
  - Texteinheit, beginnend mit Start-tag und dazu passendem End-tag
    - Die Texteinheit zwischen den beiden korrespondierenden Tags ist der *Content des Elements*
    - Kann wieder Elemente enthalten, die dann ***Kindelemente des Elements*** genannt werden (Baumstruktur)
  - Ein einzelnes Empty-element-tag
  - Beispiel

```
<RectShape>
  <Position x=„4“ y=„3“ />
  <Size width=„2“ height=„7“ />
  <Color>Yellow</Color>
</RectShape>
```

XML als Baum

```
<RectShape>
  <Position x=„4“ y=„3“ />
  <Size width=„2“ height=„7“ />
  <Color>Yellow</Color>
</RectShape>
```



XML in Java

- Wir werden DOM benutzen
- XML wird komplett eingelesen und in einen Baum überführt
- Einfaches Traversieren des Baums (Java 1.7):

```
Node node = root.getFirstChild();
while(node != null) {
  switch(node.getNodeName()) {
    case "CubeShape":
      this.initCube(node);
      break;
  }
  node = node.getNextSibling();
}
```

XML, Beispiel GUI

```
<Gui name="overlay">
    <Button text="New Game"
        x="608" y="-64" width="64" height="64"
        action="events.NewGameEvent" />

    <Label text="pre-alpha"
         x="16" y="-16" width="256" height="32" />

    <Button text="res:Textures/Quit.png"
        x="-8" y="-8" width="48" height="48"
        action="events.QuitEvent" />

    <Button text="res:Textures/Save.png"
            x="256" y="256" width="32" height="32"
            action="events.SaveEvent" />
</Gui>
```

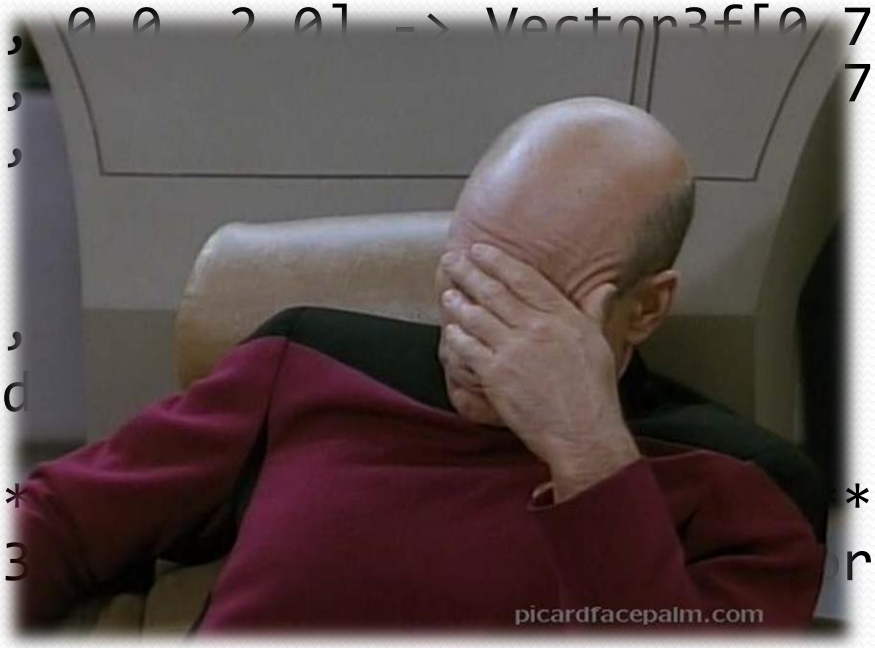
Wichtige Grundlagen: Logging

- Szenario: KI-Programmierer braucht Ausgaben, um einen Fehler zu finden…

```
Actor badguy = this.getActor("BadGuy");
Route route = badguy.findRoute(this.player);
if(route != null) {
  for(Vector3f waypoint : route.getWayPoints()) {
    System.out.print(waypoint + " -> ");
  }
  System.out.println("\n" + route.getWayPoints().size());
} else {
  System.err.println("No way!"); // Bill and Ted: Yes way!!
}
```

Ausgabe des Codes

```
17
stop
Vector3f[1.0, 0.0, 2.0] -> Vector3f[0.7, 0.0, 2.3] ->
Vector3f[1.3, 0.0, 1.7, 0.0, 1.6] ->
Vector3f[0.5,
5
done
Vector3f[1.0,
Actor created
No way!
**********                                **********
******Vector3                              r3f[0.7, 0.0,
0.1] ->
2
```



Probleme

- Verschiedene Programmierer erzeugen unterschiedliche Debuggingausgaben
- Niemand wird vor jede seiner Ausgaben schreiben, dass er / sie dafür verantwortlich ist oder wo das zugehörige `println` im Programm zu finden ist
- Projekt nach `System.out.println` durchsuchen
  - `Found 127 matches of System.out.println in 41 files`
- Besser
  - automatische Angabe der Datei und Zeile einer Ausgabe
  - Filtern der Ausgabe

Ansatz Logger

- Programm umschreiben und dabei statt `System.out` die Klasse `Logger` benutzen

```
Actor badguy = this.getActor("BadGuy");
Route route = badguy.findRoute(this.player);
if(route != null) {
  String out = "";
  for(Vector3f waypoint : route.getWayPoints()) {
    out += waypoint + " -> ";
  }
  Logger.INSTANCE.info(out);
  Logger.INSTANCE.info(route.getWayPoints().size());
} else {
  Logger.INSTANCE.warning("No way!"); // Bill and Ted: Yes way!!
}
```

Ausgabe des Codes

```
[INFO] VertexBuffer.java, 23: 17
[ERROR] GameLogic.java, 123: stop
[INFO] AiHandler.java, 37: Vector3f[1.0, 0.0, 2.0] ->
Vector3f[0.7, 0.0, 2.3] -> Vector3f[1.3, 0.0, 1.4] ->
Vector3f[0.7, 0.0, 1.6] -> Vector3f[0.5, 0.0, 1.5] ->
[INFO] AiHandler.java, 38: 5
[INFO] MeshLoader.java, 100: done
[INFO] GameLogic.java, 55: Vector3f[1.0, 1.0, 1.0]
[INFO] GameLogic.java, 21: Actor created: 178
[WARNING] AiHandler.java, 40: No way!
[INFO] GameLogic.java, 67: ****************************
[INFO] AiHandler.java, 37: Vector3f[0.2, 0.0, 0.3] ->
Vector3f[0.7, 0.0, 0.1] ->
[INFO] AiHandler.java, 38: 2
```

Ausgabe des Codes

- Nach eingestelltem Filter

```
[INFO] AiHandler.jav                                .0, 2.0] ->
Vector3f[0.7, 0.0, 2                                .0, 1.4] ->
Vector3f[0.7, 0.0, 1                                .0, 1.5] ->
[INFO] AiHandler.jav
[WARNING] AiHandler.
[INFO] AiHandler.jav                                .0, 0.3] ->
Vector3f[0.7, 0.0, 0
[INFO] AiHandler.jav
```

Nächste Woche:

- Übersicht Aufbau einer Engine
  - GameApp
  - GameLogic
  - GameView
- Detailliert: GameApp
- Detailliert: EventManager

Vielen Dank für die Aufmerksamkeit