# Aufbau interaktiver 3D-Engines

Universität Osnabrück
Fachbereich Mathematik / Informatik

## 6. Übung

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

14.05.2013

# Übersicht

1. Einführung in JavaScript
2. Integration in die Engine

# JavaScript

1. Warum Scripting ?
   - Designer benötigen Abstraktion vom Code
   - Einfachere Programmiersprache
   - Schnellerer Entwicklungszyklus
2. Eigenschaften von JavaScript
   - Dynamisch typisiert
   - Klassenlos
   - Objektorientiert auf Basis von Prototypen
   - Funktionen sind vollwertige Objekte
   - Wird hauptsächlich in der Sandbox eines Browsers ausgeführt (Interessant ist WebGL Programmierung)

# JavaScript: Typen

- Variablentypen

```
var testBool = true; // Anlegen einer lokalen Variable
print(typeof testBool); // Ausgabe: boolean

var testString = "Hallo";
print(typeof testString); // Ausgabe: string

var testNumber = 42;
print(typeof testNumber); // Ausgabe: number

var testFunction = function() {};
print(typeof testFunction); // Ausgabe: function

var testObject = {};
print(typeof testObject); // Ausgabe: object

var testUndefined;
print(typeof testUndefined); // Ausgabe: undefined

var testArray = [];
print(typeof testArray); // Ausgabe: object
```

**Merke:**

- Alle Typen sind vordefinierte Objekte

# JavaScript: Kontrollstrukturen

- Kontrollstrukturen

```
if (90 > 50) {
    print("Die Bedingung ist erfuellt!");
} else {
    print("Die Bedingung ist nicht erfuellt!");
}
// Ausgabe: Die Bedingung ist erfuellt!

print("Die Bedingung ist " + (90 > 50 ? "" : "nicht ") + "erfuellt!");
// Ausgabe: Die Bedingung ist erfuellt!

var testVar = 40;
switch (testVar) {
    case 30:
        print("TestVar ist 30");
        break;
    case 40:
        print("TestVar ist 40");
        break;
    default:
        break;
}
// Ausgabe: TestVar ist 40
```

# JavaScript: Kontrollstrukturen

- Kontrollstrukturen (Besonderheiten)

```
if (true) {
    print("Test 1");
}
// Ausgabe: Test 1
if ('true') {
    print("Test 2");
}
// Ausgabe: Test 2
if (false) {
    print("Test 3");
}
// keine Ausgabe
if ('false') {
    print("Test 4");
}
// Ausgabe: Test 4
var varUndefined; // Variable ist undefined
if (varUndefined == null) {
    print("Test 5");
}
// Ausgabe: Test 5
if (varUndefined === null) {
    print("Test 6");
}
// keine Ausgabe
```

**Merke:**

- *false*-Werte sind:
  - *false*, *null*, *undefined*, '', 0, *NaN*
- == und *!*= konvertieren einige Typen, lieber === und *!*== benutzen

# JavaScript: Schleifen

- Schleifen

```
for (var i = 0; i < 5; i += 1) {
    print("Schleife");
}
// Ausgabe: 5 mal ...

var Zaehler = 0;
while (Zaehler < 3) {
    print("Schleife 2");
    Zaehler += 1;
}
// Ausgabe: 3 mal ...

Zaehler = 0;
do {
    print("Schleife 3");
    Zaehler += 1;
} while (Zaehler < 3);
// Ausgabe: 3 mal ...
```

# JavaScript: Object

- Das Object
  - Key - Value Datenstruktur

```
// Anlegen eines neuen Objects
var newObject = new Object();
newObject = {};

// Definieren von Eigenschaften
newObject.name = "Fritz";
newObject['name'] = "Berthold";

newObject.age = 42;
print(newObject['age']); // Ausgabe: 42

// object literal syntax
newObject = { name : "Fritz", age : 42 };

print('NewObject Name: ' + newObject.name); // Ausgabe: Fritz
delete newObject.name; // delete-Operator kann eine Objekt-Eigenschaft entfernen
print('NewObject Name: ' + newObject.name); // Ausgabe: undefined
```

# JavaScript: Funktionen

- Funktionen

```
// Objekt-Eigenschaft ist eine Funktion
newObject.getAge = function() {
    return this.age; // this ist das Object dieser Property
}
newObject.setAge = function(age) {
    if (age == undefined) { // Default-Parameter festlegen
        age = 42;
    }
    for (var i = 0; i < arguments.length; i += 1) {
        print(arguments[i]); // Alle angegebenen Parameter durchiterieren
    }
    this.age = age;
}
newObject['getClone'] = function() {
    var clone = { name : this.name, age : this.age, getAge : this.getAge, setAge : this.setAge };
    // gibt eine Objekt-Kopie zurueck
    // Moeglichkeit fuer mehrere Rueckgabewerte
    return clone;
}

var clone = newObject.getClone();
newObject.setAge(50);
print('NewObject Age: ' + newObject.getAge()); // Ausgabe: 50
print("Clone Age: " + clone.getAge()); // Ausgabe: 42
clone.setAge(50);
print("Clone Age: " + clone.getAge()); // Ausgabe: 50
```

# "Vererbung"

- "Vererbung" durch Prototypen

```
// Konstruktor-Funktion für ein neues Objekt
var Tier = function (name, laut) {
    this.name = name;
    this.laut = laut;
}

var dieKatze = new Tier("Die Katze", "miau");
// new erstellt ein neues Objekt und fuehrt die Funktion
// mit diesem Object als "this" aus (Dazu gleich mehr)
var derHund = new Tier("Der Hund", "wau");

print(dieKatze.macht());
// Type-Error, da die Funktion nicht gefunden werden kann

// Definition einer Funktion beim Prototypen
Tier.prototype.macht = function () {
    return this.name + " macht " + this.laut;
}
// Wenn im Objekt eine Eigenschaft nicht gefunden wird, wird im
// Prototypen nachgeschlagen
print(dieKatze.macht());
// Ausgabe: Die Katze mache miau
```

# "Vererbung"

- "Vererbung" durch Prototypen (2)

```
var Raubtier = function (name, laut, waffe) {
    this.waffe = waffe;
    this.name = name;
    this.laut = laut;
}

// Der Prototyp des Raubtiers wird durch ein Tier-Objekt ersetzt
Raubtier.prototype = new Tier();
// Der Konstruktor wird manuell korrigiert
Raubtier.prototype.constructor = Raubtier;

Raubtier.prototype.attacke = function () {
    return this.name + " greift mit " + this.waffe + " an!";
}

var RaubKatze = new Raubtier("Die Raubkatze", "miau", "Krallen");
print(RaubKatze.macht());
// Die Raubkatze macht miau
print(RaubKatze.attacke());
// Die Raubkatze greift mit Krallen an
```

# Built-In Objects

- Built-In Objects (Funktionen der Prototypen)
  - Object

```
// Prueft, ob die Eigenschaft von Objekt (true), oder vom Prototypen (false) kommt
Object.prototype.hasOwnProperty(propname);

for (var prop in RaubKatze) {
    if (RaubKatze.hasOwnProperty(prop)) {
        print("eine eigene Eigenschaft is: " + prop);
    }
}
// Ausgabe: name, laut, waffe

// Gibt eine String-Repräsentation des Objekts zurück
Object.prototype.toString();
print("toString: " + RaubKatze.toString());
// Ausgabe: [object Object]
```

# Built-In Objects

- Built-In Objects (Funktionen der Prototypen)
  - Function

```
// Ruft eine Funktion mit einem bestimmten "this" auf
Function.prototype.call(thisObject, arg1, arg2, ...);
```

  - Array

```
var testArray = ["Apfel", "Birne", "Kirsche"];

Array.prototype.length // Laenge des Arrays

testArray[5] = "Ananas";
print("Laenge: " + testArray.length);
// Ausgabe: 6

print("Nummer 3: " + testArray[3]);
// Ausgabe: undefined

Array.prototype.splice(index, count); // Loescht count Elemente vom index beginnend
Array.prototype.sort(); // Sortiert Elemente alphabetisch
Array.prototype.join(','); // Erstellt beim TestArray den String "Apfel, Birne, Kirsche"
```

# Built-In Objects

- Built-In Objects (Funktionen der Prototypen)
  - Math
    - Hat Funktionen wie sin(Number), cos(Number), … ähnlich wie in Java
  - Number
    - Verschiedene Ausgabeformate
  - String
    - Ähnlich wie in Java
- Weitere Hinweise
  - Closures

# Integration in die Engine

- *JavaScriptManager.java*
  - Nutzt die Rhino Bibliothek -

https://developer.mozilla.org/en-US/docs/Rhino

  - Erstellt JavaScript-Context und Scope
  - *executeFile(String resource)* - führt eine JavaScript-Datei aus
  - *invokeFunction(String name, Object ...)* - führt eine JavaScript-Funktion mit bestimmten Argumenten aus

# Integration in die Engine

- Rhino-Funktionalitäten
  - Zugriff auf Java-Klassen und Objekte

```
var ProcessManager = Packages.main.GameApp.getGlobalProcessManager();
var Vector3f = Packages.org.lwjgl.util.vector.Vector3f;
var MathUtil = Packages.util.MathUtil;
```

  - Instanziierung von Interfaces und abstrakten Klassen (nächste Folie)

# Integration in die Engine

- Ein Event erstellen
  - Bestehendes Event

```
var actorMoveEvent = new Packages.events.LogicEvents.ActorMoveEvent(actorId);
actorMoveEvent.setPosition(xPos, yPos, zPos);
actorMoveEvent.setRotation(yaw, pitch, roll);
EventManager.queueEvent(actorMoveEvent);
```

  - ScriptEvent (1)

```
public abstract class ScriptEventData extends EventData {

    public String getStringAttribute(String name) {
        return this.getScriptData().get(name, this.getScriptData()).toString();
    }
    @Override
    public int getId() {
        return 0;
    }
    public Scriptable getScriptData() {
        return null;
    }
    public Vector3f getPosition() {
        return null;
}}
```

# Integration in die Engine

- Ein Event erstellen
  - ScriptEvent (2)

```
var rotation = new Vector3f(0.0, 0.0, 0.0);
var testEvent = { ID: 5,
                  getId : function () { return this.ID; },
                  getScriptData : function () { return { name : "testEvent"} },
                  getPosition : function () { return rotation; },
                };
var realEvent = new Packages.event.ScriptEventData(testEvent);

EventManager.queueEvent(realEvent);
```

  - Der passende Script-EventListener

```
var testEventListener = { trigger : function (data) {
                            print("ScriptEvent angekommen :)");
                        }
};
var realEvtListener = new Packages.event.EventListener(testEventListener);
EventManager.register(realEvtListener, 5);
```

# Integration in die Engine

- Script-Prozesse erstellen

```
var testProcess = {
                maxTime : 1000,
                currTime : 0,
                counter : 0,
                onUpdate : function (dMillis) {
                                this.currTime += dMillis;
                                if (this.currTime > this.maxTime) {
                                    print("Test");
                                    this.counter += 1;
                                    this.currTime = 0;
                                }
                                if (this.counter == 4) {
                                    this.succeed();
                                }
                            }
                };

var realProcess = new Packages.process.EngineProcess(testProcess);
ProcessManager.attachProcess(realProcess);
```

# Integration in die Engine

- Weitere Hinweise
  - Ein Prozess kann prüfen, ob sich an einer Skript-Datei etwas verändert (per lastModified-Timestamp) hat und die Datei neu laden
  - Hilfsfunktionen zur ActorErstellung und Events anlegen
  - **Achtung**
    - Scripting dient zur schnellen Entwicklung und Überprüfung von Spiel-Elementen
    - Scripts sollten nie Standard-Funktionen der Engine ersetzen
    - Nie die Spieldaten direkt verändern, nur über **Events**, die vor der Ausführung geprüft werden

# Weitere Resourcen

- Douglas Crockford: JavaScript: The Good Parts
- Aaftab Munshi: OpenGL ES 2.0 Programming Guide - WebGL Programmierung
- JavaScript-Engines
  - http://www.cocos2d-x.org/ - Cross-Platform Engine
  - http://threejs.org/ - vereinfachte WebGL-Entwicklung

Nächste Woche vorraussichtlich:

- To be announced

# Vielen Dank für die Aufmerksamkeit ☺