Programm

- Ein paar Worte zu branches in SVN
- Die Objekthierarchie in Orxonox
- Styleguide
- Designgrundsätze
- Worauf man im Netzwerk achten muss
- Käferbekämpfung
- Epic fails aus der Geschichte des PPS
- strings und die STL
- Diskussion der Projekte in Gruppen

Zur Erinnerung

- Am Ende des Semesters gibt es eine öffentliche Präsentation
  - Orxonox wird dem begeisterten Publikum vorgeführt
  - Jeder präsentiert sein Ticket mit ein paar Worten und ein paar Slides
  - Ekstatische Fans werden ins Unispital überliefert
- Mailinglists: Wiki -> Development -> Gelber Block
  - announce für alle
  - dev für alle ausser die Desinteressierten
  - commit für die Programmierer (empfehlenswert: Filter einrichten)
- Alle Slides gibts auch online: Wiki -> Development -> Fetter Link im gelben Block -> unten in der Timetable

SVN Revisited

- SVN habt ihr bereits kennen gelernt:
  - checkout
  - add
  - commit
  - update

SVN Revisited

- Wir können nicht alle im selben repository arbeiten
- Jeder (sofern benötigt) bekommt einen branch
  - Ein branch ist eine Kopie des Hauptrepositorys (trunk)
  - Nach einer gewissen Zeit wird der branch zurück in den trunk gemerged
  - Dabei werden die Änderungen aus eurem branch mit den Änderungen im trunk vereint
  - Anschliessend wird (sofern benötigt) ein neuer branch erstellt

SVN Revisited

- Checkout eines branches:
  - svn co https://svn.orxonox.net/orxonox/branches/branchname branchname
  - Eine Liste aller branches gibt es auf der Wiki -> Development -> Browse Source -> branches

SVN Revisited

- Was man mit SVN bitte nicht machen sollte:
  - trunk auschecken
  - media auschecken
  - trunk in den media ordner kopieren
  - media commiten
- Man kann mit SVN zwar alles rückgängig machen aber es kostet Zeit. Daher gilt:
  - erst denken, dann commiten
- Ist übrigens nicht böse gemeint ;-)

Die Objekthierarchie

- In Orxonox gibt es das Framework und die Objekte
- Objekte beeinflussen das Spiel oder sind vom Spiel abhängig, direkt oder indirekt
- Beispiele:
  - SpaceShip: Ganz klar ein Objekt
  - Künstliche Intelligenz: Ist zwar nur abstrakt, trotzdem ein Objekt da sie das Spiel beeinflusst
  - Scoreboard: Beeinflusst das Spiel nicht, wird aber davon beeinflusst (Spielernamen, Punktestände), daher auch ein Objekt
  - Textur: Kein Objekt, gehört zum Framework

Die Objekthierarchie

- Die Objekthierarchie beschreibt die Klassenhierarchie der Objekte, die gegenseitigen Abhängigkeiten, Interfaces, Funktionen, Kontrollstrukturen, Manager-Klassen
- Warum ist die Objekthierarchie noch so unfertig?
  - Das Framework hatte in den letzten zwei Semestern Vorrang
  - Die Provisorischen Objekte (z.B. SpaceShip) basieren auf Hacks
  - Diese Hacks sind schwer zu ersetzen, daher macht man neue Hacks
  - Die Objekthierarchie benötigt daher einen Neubeginn

Die Objekthierarchie

- Das BaseObject
  - Die Mutter aller Objekte
  - Definiert Grundlegende Funktionen
  - Wer von BaseObjekt erbt, kann alles was ein Objekt können muss
    - Durch XML geladen werden
    - Einen Identifier haben
    - Config-values besitzen
    - Vieles mehr
  - Alle Objekte werden bei Beginn eines Levels geladen und am Ende zerstört

Die Objekthierarchie

- Das WorldEntity
  - Die Basisklasse aller Objekte mit einer Position im Raum
  - Beispiele:
    - SpaceShip: Benötigt eine Position, ist also ein WorldEntity
    - Ein Geschütz an einem SpaceShip: Hat eine Position relativ zur Position des SpaceShips, ist daher auch ein WorldEntity
    - Künstliche Intelligenz: Benötigt keine Position, ist also KEIN WorldEntity (aber immerhin ein BaseObject)
  - WorldEntities können zusammengehängt werden (attached)
    - Die Position eines angehängten WorldEntities ist relativ zur Position und Rotation des Trägers
    - Die Logik dieser Verknüpfung übernimmt die SceneNode
  - Jedes WorldEntity besitzt eine SceneNode

Die Objekthierarchie

- Die SceneNode
  - Eine Klasse aus Ogre (unsere Grafiklibrary), d.h. nicht von Orxonox selbst
  - Definiert Position und Rotation im Raum
  - Erlaubt Verknüpfungen
  - Ermöglicht das Anhängen einer grafischen Instanz:
    - Mesh (3D Modell)
    - Billboard
    - ParticleEffect
    - Light
    - usw...

Die Objekthierarchie

- Position im Raum ist kein Problem (wenngleich manchmal verwirrend auf Grund relativer Koordinaten und unterschiedlichen Skalierungen)
- Rotation im Raum ist jedoch ein grosses Problem
- Es existieren verschiedene Beschreibungsmodelle
- Im Folgenden soll die Problematik der Rotation in groben Zügen erläutert werden

Die Objekthierarchie

- Drei Winkel (eulersche Winkel), je einer pro Achse
  - Einfach und verständlich
  - Problem:
    - Reihenfolge der Winkel ist relevant
    - Negative Winkel bedeuten bei gleicher Reihenfolge keine Rück-Rotation
- Quaternion: Komplexe Zahl mit drei komplexen Werten und einem reellen
  - Beschreibt Rotation und Skalierung (daher vier Werte)
  - Einheitsquaternion beschreibt Rotation ohne Skalierung
  - Problem:
    - Keine Chance ein Quaternion im Kopf zu berechnen
    - Für Menschen unübersichtlich und fehleranfällig

Die Objekthierarchie

- Hybridmodelle, z.B. Blickrichtung + Rotation
  - Einfach zu beschreiben, einfach vorstellbar
  - Problem:
    - Mathematisch überbestimmt
    - Unflexibel für gewisse Rotationen
- Vereinfachungen: Fester Boden (z.B. in FPS), Blickrichtung alleine reicht
  - Sehr simpel
  - Problem:
    - Nicht alle Rotationen möglich
    - Virtuelle Ebene im Raum ist für ein Spaceshooter unpraktisch

Die Objekthierarchie

- Zusammenfassung:
  - Das Rotationsproblem ist noch ungelöst
  - Zur Zeit gibt es mehrere Möglichkeiten ein Objekt zu rotieren, das wird vermutlich auch so bleiben
  - Wichtig ist, dass man sich der Problematik bewusst ist
  - Für den FPS-Modus kann man auf Vereinfachungen zurückgreifen

Die Objekthierarchie

- Das Tickable Interface
  - Ein Interface erbt nicht von BaseObject, aber andere Objekte können vom Interface erben (Mehrfachvererbung)
  - Tickable ist die Basisklasse aller Objekte, welche in jedem Tick (eine virtuelle Spielrunde) etwas machen wollen
  - `virtual void tick(float dt)`
  - dt ist die Zeit seit dem letzten Tick in Sekunden
  - Damit Orxonox nicht von der Framerate abhängig ist, muss dieses dt in verschiedenen Formen im Code verwendet werden

Die Objekthierarchie

- Die Zukunft der Objekthierarchie:
  - Trennung von Spielfigur (SpaceShip) und Controller (Player oder künstliche Intelligenz)
    - Basisklassen für steuerbare Objekte und steuerfähige Controller
    - Interface zwischen beiden Gattungen
  - Ein Playermanager kontrolliert die vorhandenen Spieler
    - Jedem Spieler (Client) muss ein Schiff zugewiesen werden
    - Der Client darf sich sein Schiff nicht selbst aussuchen
    - Zwei Clients dürfen nicht das selbe Schiff steuern

Die Objekthierarchie

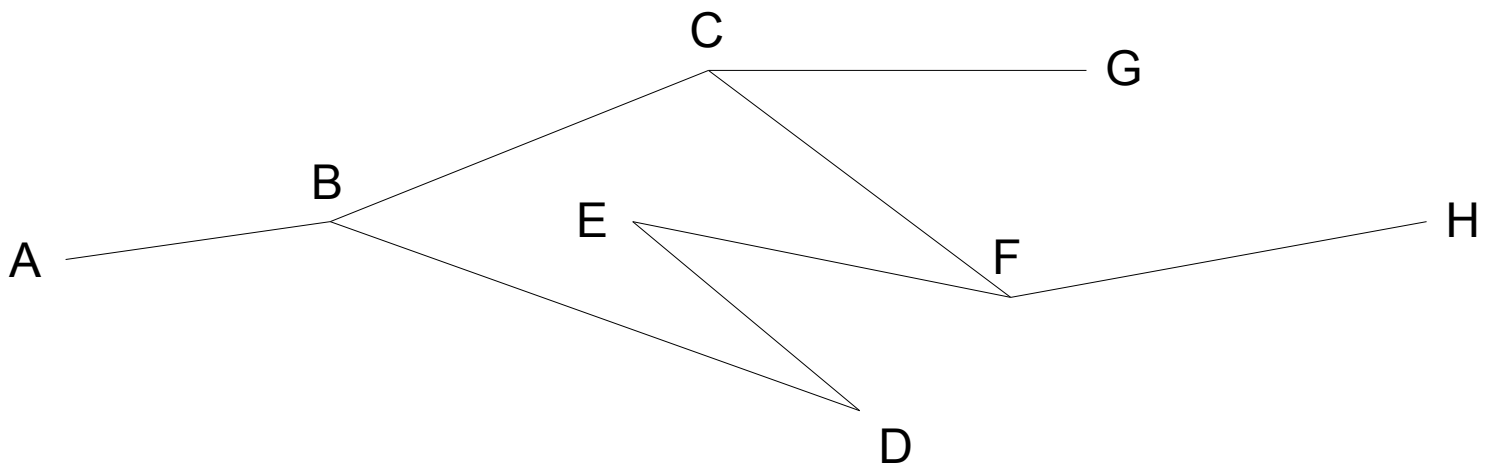
- Die Zukunft der Objekthierarchie:
  - Spawn (Betreten des Levels) ist nicht Aufgabe des SpaceShips, sondern des Levels
    - Eine Kontrollklasse lässt ein SpaceShip spawnen
    - Ort und Ausrichtung beim Start wird durch Spawnpunkte definiert
    - Das Schiffsklasse des Spielers ist der Kontrollklasse bekannt
  - Eine Kontrollinstanz überwacht den Start und den Verlauf eines Spiels
    - Messen der Spielzeit
    - Definieren von Regeln (wofür gibt es Punkte, wann ist das Spiel zu Ende)
    - Eventuell Einteilung in Teams

Designgrundsätze

- Generisch Programmieren
  - Generische Programmierung ist das Gegenteil eines Hacks
  - Man löst ein Problem allgemein und nicht spezifisch
  - Kapselung eines Problems/Features in einer Klasse
  - (Möglichst) keine Abhängigkeiten zu und in andere Klassen
  - Kann mit jedem Input umgehen
  - Funktioniert (theoretisch) immer
  - Eröffnet manchmal ungeahnte Möglichkeiten

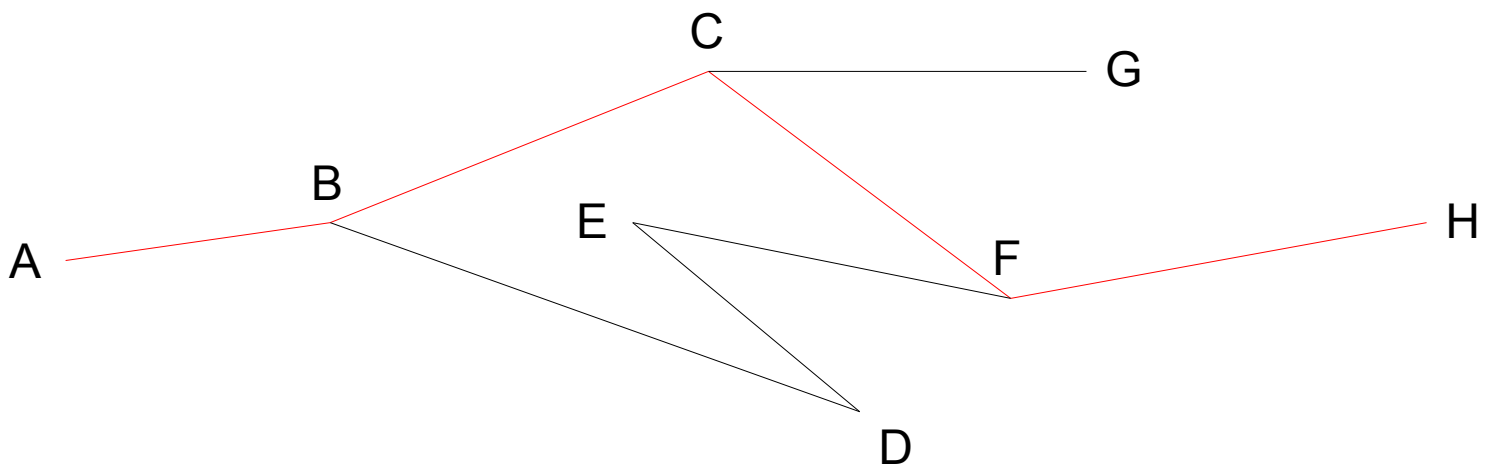
Designgrundsätze

- Generisch Programmieren
  - Beispiel: A -> H



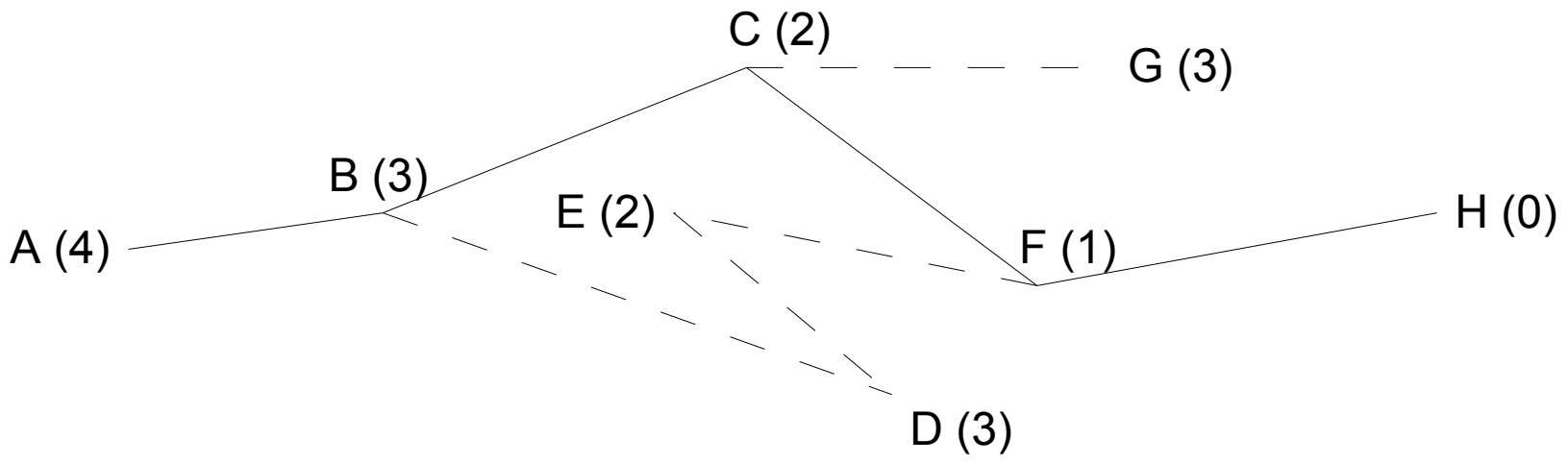
Designgrundsätze

- Generisch Programmieren
  - Hack: Hardcoded A -> B -> C -> F -> H



Designgrundsätze

- Generisch Programmieren
  - Generisch: Wegfindungs KI



Designgrundsätze

- Klassenhierarchie
  - Wenn zwei Klassen starke Ähnlichkeiten aufweisen, kann man eine gemeinsame Basisklasse einfügen.
  - Beispiel:
    - Klasse 1: AABCD
    - Klasse 2: AABEF

    - Basisklasse: AAB
    - Klasse 1: CD
    - Klasse 2: EF
  - Reduktion von redundantem Code erhöht Flexibilität und verringert Fehleranfälligkeit

Designgrundsätze

- Klassenhierarchie
  - Grosse Klassen die viele Funktionen haben, können ihn mehrere kleine Klassen aufgeteilt werden
  - Manchmal bietet sich auch eine Auftrennung in ein Master/Slave Verhältnis an (z.B. SpaceShip und Controller)

Designgrundsätze

- Kapselung
  - Allgemein gilt: Jedes Problem/Feature sollte in sich geschlossen behandelt werden und keine Abhängigkeiten in andere Klassen auslagern.
  - Eine Klasse ist von aussen betrachtet immer perfekt, d.h. man kann nichts Falsches machen. Die komplizierte Logik findet im Inneren der Klasse statt und ist generisch programmiert, so dass sie mit jedem Input umgehen kann.
  - Ausnahmen sind zulässig wenn zu viel Sicherheit der Performance schadet und höchstens bei idiotischen Inputs ein Fehler entsteht.

Erfahrungen aus dem PPS

- Beispiel 1: Die ClassTreeMask
  - Problemstellung: Definiere welche Klassen aus einem XML-File geladen werden sollen und welche nicht.
    - Beispiel: Lade keine Gegner wenn man nur den Level erkunden möchte.
  - Erster Ansatz:
    - Eine Liste mit erlaubten Klassen
    - Problem: Die Liste wird verdammt lang wenn man alles laden möchte
  - Zweiter Ansatz:
    - Eine Liste mit nicht erlaubten Klassen
    - Problem: Was, wenn ich zwar alle Gegner ausschliessen möchte, aber den Endboss trotzdem laden?

Erfahrungen aus dem PPS

- Beispiel 1: Die ClassTreeMask
  - Dritter Ansatz:
    - Eine Liste mit nicht erlaubten Klassen
    - Eine zweite Liste mit Klassen die eben doch erlaubt sind
    - Problem: Was, wenn ich noch feinere Unterteilungen möchte?

Erfahrungen aus dem PPS

- Beispiel 1: Die ClassTreeMask
  - Lösung
    - Warum Listen machen, wenn man einen Baum beschreiben will?
    - Implementierung eines gigantischen Konstrukts um eine Maske über die Klassenhierarchie zu legen
    - Inklusive Operatoren (and, or, xor, not, usw...)
    - Grosser Aufwand für ein Feature, das vermutlich gar nicht benötigt wird
  - Warum es sich trotzdem gelohnt hat
    - Monate später hat jemand einen Trigger geschrieben, der ein Ereignis auslöst, wenn ein Objekt in seine Nähe kommt
    - Die ClassTreeMask wird nun verwendet, um die Klassen zu beschreiben, welche den Trigger auslösen können
  - Fazit: Guter Code lohnt sich immer

Erfahrungen aus dem PPS

- Beispiel 2: Der Projektilmanager im Mainloop
  - Beschreibung:
    - Ein Student hat sich mit einem Waffen- und Projektilsystem beschäftigt
    - Um die Projektile zu verwalten, hat er einen Projektilmanager geschrieben
    - Dieser Projektilmanager musste im Mainloop (das Herz von Orxonox) erzeugt werden
  - Problem:
    - Orxonox sollte sich nicht um einzelne Features kümmern müssen
    - Features müssen sich selbst organisieren
    - Prinzip der Kapselung
  - Lösung:
    - Zur Zeit nicht implementiert, aber man könnte den Projektilmanager statisch verfügbar machen oder aber die Projektile sich selbst zerstören lassen.

Erfahrungen aus dem PPS

- Beispiel 3: Verkehrte Welt im HUD
  - Beschreibung:
    - Im HUD gibt es eine SpeedBar welche die Geschwindigkeit darstellt
    - Die SpeedBar besteht aus einem veränderlichen Balken und einem festen Hintergrund
  - Problem:
    - Das SpeedBar-Objekt hat sich selbst einen Hintergrund erzeugt und sich an diesen Hintergrund angehängt
    - Die Hierarchie der HUD Elemente war dadurch total chaotisch

Erfahrungen aus dem PPS

- Beispiel 3: Verkehrte Welt im HUD
  - Lösung:
    - Das SpeedBar-Objekt ist nun selbst der Hintergrund und erstellt sich einen Balken
    - Dadurch ist die Hierarchie geordnet und man kann die SpeedBar auch an andere Elemente anhängen, verschieben und sogar rotieren, ohne dass unvorhergesehene Effekte eintreten
  - Fazit: Wenn die Hierarchie stimmt, wird die Handhabung intuitiver und es kommt viel seltener zu Fehlern.

Erfahrungen aus dem PPS

- Beispiel 4: Der bewaffnete Partikel-Effekt
  - Beschreibung:
    - In der ersten Version von Orxonox hatten wir eine extrem flache Klassenhierarchie.
    - Vererbung unter Objekten existierte praktisch nicht
    - Es gab keine Interfaces oder abstrakte Basisklassen welche eine sinnvolle Struktur ermöglicht hätten
  - Problem:
    - Menschliche Spieler und Computergegner waren zwei unterschiedliche Klassen
    - Ihre gemeinsame Basisklasse war das WorldEntity
    - Damit Spieler und Computer gleichermassen mit Waffen ausgerüstet werden konnten, musste das WorldEntity einen Weaponslot besitzen.

Erfahrungen aus dem PPS

- Beispiel 4: Der bewaffnete Partikel-Effekt
  - Problem (Fortsetzung):
    - Weil auch viele andere Objekte von WorldEntity erbten, hatten auch diese die Fähigkeit, Waffen zu tragen
    - Darunter auch Partikel-Effekte, Türen oder Waffen selbst

    - Natürlich mussten die Spieler auch sterben können
    - Auch dies wurde im WorldEntity implementiert
    - Man konnte Türen erschiessen
    - (Zum Glück haben sie nicht zurückgeschossen)

Erfahrungen aus dem PPS

- Beispiel 4: Der bewaffnete Partikel-Effekt
  - Lösung:
    - Das Chaos war so gross, dass keine direkte Lösung möglich war
    - Es wurde beschlossen, die Objekthierarchie im zweiten Anlauf ganz anders zu implementieren
    - Wie ihr gesehen habt, wird noch immer daran gearbeitet ;-)
    - (Aber immerhin sind unsere Partikel-Effekte nun demilitarisiert)
  - Fazit: Eine breite Klassenhierarchie kann zwar manchmal verlockend sein, da man weniger planen muss. Sobald man aber ein ernsthaftes Game entwickeln will, kommt man um eine durchdachte Hierarchie nicht herum.

Erfahrungen aus dem PPS

- Beispiel 5: Der Seitenangriff
  - Beschreibung:
    - Vor zwei Jahren war noch geplant, dass Orxonox ein Vertical Scroller werden soll, d.h. ein Game, in dem man von oben auf das Geschehen blickt und die Gegner von oben nach unten über den Bildschirm fliegen, während sich der Spieler in die entgegengesetzte Richtung bewegt.
    - Auf Grund der Beschaffenheit eines Vertical Scrollers konnte der Spieler nur nach vorne schiessen
    - Es wurde eine künstliche Intelligenz implementiert, welche den Spieler von vorne attaktierte
    - Parallel dazu wurde ein Track-System entwickelt, das die Bewegung des Spiels nicht nur geradeaus ermöglicht, sondern auch um Kurven und entlang eines vordefinierten Pfades. Das Ziel war, spannende Levels zu erstellen, um die starre Ansicht im Vertical Scroller aufzulockern.

Erfahrungen aus dem PPS

- Beispiel 5: Der Seitenangriff
  - Problem:
    - Kurz vor Ende des Semesters wurden die branches gemerged. Die künstliche Intelligenz funktionierte prächtig und auch das Track-System war ein echter Hingucker.
    - Für die Schlusspräsentation wurde ein Level erstellt, in dem das Schiff einen Bogen fliegt um sich einer verlassenen Station zu nähern. Gleichzeitig wurde der Spieler von Gegnern attakiert.
    - Leider war die künstliche Intelligenz so programmiert, dass sie nur dann von vorne attakierte, wenn der Spieler auch nach vorne flog.
    - Als das Schiff in die Kurve flog, kamen die Gegner plötzlich von der Seite
    - Leider konnte man im Vertical Scroller Modus nicht seitlich schiessen
    - Die Präsentation war trotzdem irgendwie lustig, wenn auch unfreiwillig

Erfahrungen aus dem PPS

- Beispiel 5: Der Seitenangriff
  - Fazit:
    - Nicht erst kurz vor der Präsentation mergen
    - Generisch programmieren und nicht bloss auf einem momentanen Zustand (der Flugrichtung) aufbauen
    - Den Level vor der Präsentation zumindest einmal durchspielen

Erfahrungen aus dem PPS

- Beispiel 6: Abhängigkeit von der Framerate
  - Beschreibung:
    - Ein Student hatte eine Schiffsteuerung implementiert
    - Dabei wurde mit Geschwindigkeiten und Beschleunigungen gerechnet
    - tick(dt) wurde verwendet, um in jedem Frame auf die Eingabe des Spielers zu reagieren
    - Mit dt wurden Geschwindigkeit und Beschleunigung normalisiert
    - Trotzdem war die Steuerung abhängig von der Framerate
  - Problem:
    - Um eine Dämpfung zu erreichen, wurde die Beschleunigung nach einem Tick auf Null gesetzt.
    - Leider ist „nach einem Tick“ aber abhängig von der Framerate
    - Bei wenigen Frames ist ein Tick länger und daher konnte auch länger beschleunigt werden

Erfahrungen aus dem PPS

- Beispiel 6: Abhängigkeit von der Framerate
  - Lösung:
    - Die Dämpfung linear vornehmen und in jedem Tick einen bestimmten Betrag von der Beschleunigung abziehen (in Abhängigkeit von dt):
    - `beschleunigung -= (dt * dämpfungskonstante);`

Erfahrungen aus dem PPS

- Beispiel 6: Abhängigkeit von der Framerate
  - Auch Profis machen solche Fehler
    - In der Demoversion von Unreal Tournament 2004 konnte man mit Fahrzeugen nicht anfahren, wenn die Tickrate des Servers zu hoch war.
    - Durch die hohe Tickrate wurde die Beschleunigung pro Tick kleiner als die konstante Haftreibungsschwelle.
    - Dadurch wurde die Geschwindigkeit immer wieder auf Null gesetzt
    - Nur wenn man dem Fahrzeug einen Stoss versetzte, konnte man anfahren
    - Alternativ konnte man natürlich die Tickrate des Servers reduzieren

Erfahrungen aus dem PPS

- Beispiel 6: Abhängigkeit von der Framerate
  - Auch wir machen nach wie vor Fehler
    - Die Kamera von Orxonox fliegt leicht träge hinter dem SpaceShip her
    - Bei niedriger Framerate fliegt das Schiff jedoch aus dem Blickwinkel der Kamera heraus
    - Das Spiel wird dadurch praktisch unspielbar