Starjumper

Robert Strobl, Johannes Albrecht, Tim Berning, Stefan Härtel

Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam

1 Einleitung

1.1 Spielprinzip

Starjumper ist ein Renn- und Geschicklichkeitsspiel, bei dem der Spieler mit einem Raumschiff über eine hindernisreiche Strecke mit diversen Abgründen fahren und schlussendlich sicher ins Ziel gelangen muss.
Dabei besitzen die Flächen der Streckenabschnitte unterschiedliche Eigenschaften, die sich auf das Fahrverhalten des Raumschiffs auswirken können, z.B. Abbremsen des Spielers.
Die Steuerung umfasst Beschleunigen, Bremsen, Lenken nach links oder rechts und Springen.

1.2 Spielidee

Vorlage für unser Spiel war der Spieleklassiker Skyroads aus dem Jahr 1993, welches von Bluemoon entwickelt wurde und wiederum selbst eine Neuauflage des Spiels Kosmonaut aus dem eigenen Hause darstellte.
Das Spielprinzip wurde für Starjumper größtenteils übernommen.

2 Steuerung

Die Steuerung des Raumschiffs erfolgt sehr intuitiv, der Spieler kann beschleunigen, bremsen, lenken und springen. Die folgende Tabelle zeigt die in-game Tastaturbelegung:

Beschleunigen	Pfeiltaste oben
Bremsen	Pfeiltaste unten
Links	Pfeiltaste links
Rechts	Pfeiltaste rechts
Springen	Leertaste

Tabelle 1. Tastaturbelegung

Darüber hinaus kann per `Escape` jederzeit aus dem Spiel in das Levelauswahlmenü gewechselt werden. Im Hauptmenü beendet ein Druck auf `Escape` das

Programm.
Die Navigation im Menü erfolgt über die Pfeiltasten, die ausgewählte Strecke wird mit `Enter` gestartet.

3 Architektur

Die Beschreibung der Architektur erfolgt in kleineren logischen Einheiten, eine Gesamtübersicht aller verwendeten Klassen und ihrer Verbindungen untereinander findet sich am Ende des Dokuments.
Im Folgenden sollen nun die wichtigsten architektonischen Bestandteile von Starjumper erläutert werden.

3.1 LevelMenu

Das `LevelMenu` bildet den Einstiegspunkt des Spiels. Es bietet eine Übersicht der verfügbaren Levels, die selektiert und gestartet werden können. Bei Instanziierung parst das `LevelMenu` die XML-Dateien, in denen die Levels des Spiels definiert wurden, und bereitet sie für die weitere Verarbeitung auf.
Selektiert der Benutzer ein Level, alloziiert `LevelMenu` eine Instanz der Klasse `Level` und übergibt die aufbereiteten Inhalte des entsprechenden Levelfiles. Anschließend ersetzt `Level` das `LevelMenu` als zu rendernden Szenengraphen.
Hier wird außerdem der Keyboardhandler definiert, der die Logik der Escape-Taste umsetzt (Programm beenden bzw. aus dem laufenden Spiel zur Levelübersicht zurückkehren), und in den `Viewer` eingehängt.

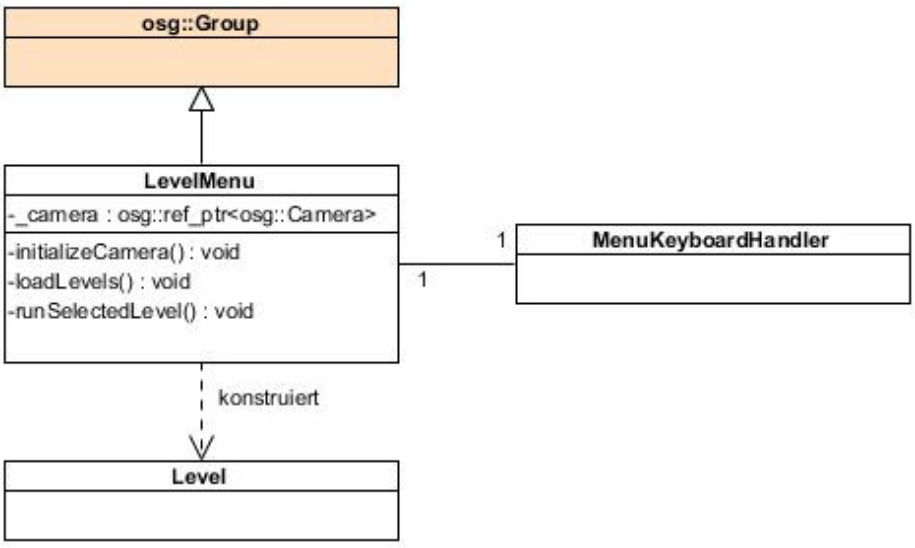

Abbildung 1. Klassendiagramm LevelMenu

3.2 Level

Die von `osg::Group` erbende Klasse `Level` stellt in sich den zu einem Spiel gehörenden Szenengraphen dar und verbindet alle Teile der Spiellogik. `Level` erstellt aus den übergebenen Leveldaten eine Reihe von Levelelementen (Kuboiden, Tunnel, ...), den `Player` (dies gilt nur für die erste Levelinstanz aufgrund der Singleton Eigenschaft, siehe Abschnitt zum Player), das `HeadUpDisplay`, den Kameramanipulator und kümmert sich um die Beleuchtung.
Neben den OpenSceneGraph-Komponenten erstellt und verwaltet `Level` zudem die Bullet Physics World, innerhalb derer physikalische Simulationen parallel zu OSG ablaufen.

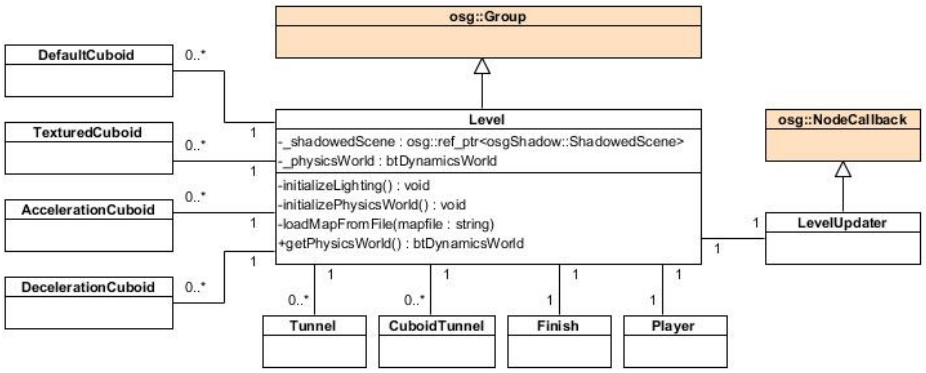

Abbildung 2. Klassendiagramm Level

3.3 Player

`Player` stellt die vom Benutzer steuerbare Entität dar in Form eines Raumschiffs, das fix aus einer Datei geladen wird. Zwecks Positionierbarkeit wird das Model in einer PositionAttitudeTransform gekapselt. Neben dem Laden dieser Modeldatei werden in `Player` die Physikkomponenten für den Spieler generiert. Die Interaktion des Benutzers mit dem Spiel erfolgt über die Komponenten `PlayerState` und `PlayerUpdater`, die im nächsten Abschnitt näher erläutert werden.
Details zu `KinematicCharacterController` und `btGhostObject` in Kombination finden sich im Abschnitt "Designentscheidungen".

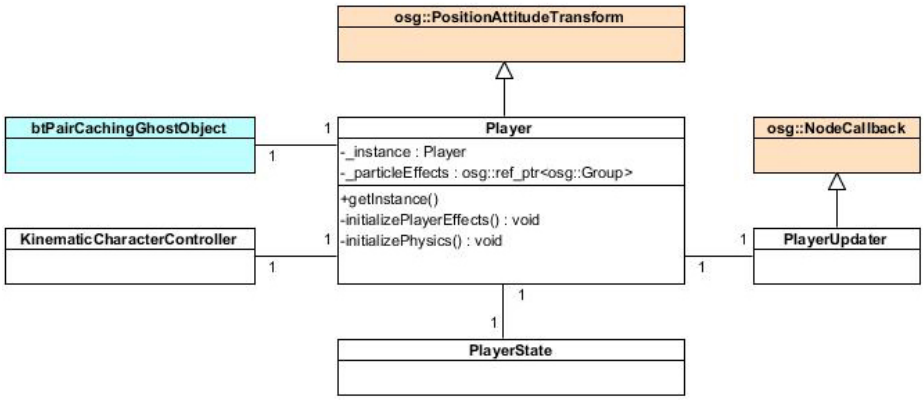

Abbildung 3. Klassendiagramm Player

4 Designentscheidungen und Design Patterns

Dieser Abschnitt beschäftigt sich mit einigen Besonderheiten der Starjumper Implementierung und erläutert deren Beweggründe und Vorteile gegenüber alternativer Implementierungsstrategien.

4.1 Physik

Da sich die physikalischen Effekte, die sich in Starjumper wiederfinden (sollten), auf korrekte Auswirkung der Schwerkraft beim Sprung oder dem Verlassen einer Plattform, sowie der Erkennung von Kollisionen mit Umgebungsobjekten beschränken, stand zu Beginn die Überlegung im Raum, diese von Hand zu implementieren. Schnell wurde jedoch klar, dass trotz des an sich geringen Umfangs der Anforderungen an die Physik eine manuelle Implementierung viel Zeit kosten würde, und so fiel die Entscheidung zugunsten der Verwendung der Physikbibliothek Bullet.
Die von Bullet zur Verfügung gestellten Klassen `btKinematicCharacterController` und `btGhostObject` boten laut Interface genau die gewünschten Funktionen: Das Ghost Object agiert als Rigid Body in der Physikwelt und kennt zu jedem Zeitpunkt alle eventuell vorhandenen Kollisionspartner, der `btKinematicCharacterController` stellt ein Steuerungsinterface für das GhostObject zur Verfügung. Aufgrund einiger weniger Unzulänglichkeiten musste allerdings der Controller angepasst werden, sodass statt dem `btKinematicCharacterController` eine modifizierte Version namens `KinematicCharacterController` Verwendung findet.
Das Rendering durch OpenSceneGraph und die Physik laufen parallel. So ist zu erklären, dass beispielsweise für Streckenelemente wie `Cuboid` jeweils eine Repräsentation für den Szenengraphen und ein `btRigidBody` für die Physics World existiert. Die Kommunikation zwischen diesen beiden Entitäten findet im `PlayerUpdater` statt. Korrekt parametrisiert (Bewegungsrichtung, Fallgeschwindigkeit, Gravitation) berechnet der Controller bei jedem Stepping der

Physics World die neue Position des Spielers, welche anschließend durch den `PlayerUpdater` abgefragt und auf die `PositionAttitudeTransform` des Spielermodells übertragen wird. Im Gegenzug werden durch den Benutzer per Tastendruck im `PlayerState` gesetzte Requests - wie der Wunsch nach links zu steuern - ausgewertet und dienen als Basis für die Aktualisierung der Parameter des Controllers.

4.2 Player

Der `Player` bzw. sein Modell wird in jeder Levelinstanz benötigt. Die im Verlauf des Spiels vorgenommenen Änderungen am `Player` beschränken sich jedoch auf Transformationen der umgebenden `PositionAttitudeTransform` und sind leicht reversibel, wie am Beispiel des Reset erkennbar ist. In einer früheren Version wurde für jedes Level eine neue Instanz der Klasse erzeugt, die logischerweise das Laden des Modells beinhaltet, was zu erheblicher Verzögerung des Spielstarts führte.
Auf Basis dieser Erkenntnisse wurde ein Refactoring vorgenommen und der `Player` zu einem Singleton erweitert. Das heißt, dass das Spielermodell nur noch ein einziges Mal beim Starten des Programms geladen werden muss und als global verfügbare Ressource gehandhabt wird, ein Aufruf der statischen Methode `Player::getInstance()` liefert auf Wunsch die Player-Instanz. Die initiale Erzeugung des `Player` Objekts erfolgt beim ersten Aufruf dieser Funktion.
Neben dem Wegfall der Ladezeiten ermöglicht dies darüber hinaus den globalen Zugriff auf den Zustand des Spielers, der z.B. die an verschiedenen Stellen des Spiels benötigten Eigenschaften Geschwindigkeit und Neigungswinkel enthält.

4.3 ParticleEffectFactory

Die Erstellung von Partikeleffekten erfordert diverse Komponenten und deren Konfiguration. So ist neben einem `ParticleSystem`, das die einzelnen Partikel über die Dauer ihres Bestehens einschließlich ihrer Löschung verwaltet, ein entsprechender `ParticleSystemUpdater` nötig - ein Callback, der das kontinuierliche Stepping des Partikelsystems übernimmt. Weitere Komponenten sind ein `Emitter`, der - bestehend aus einem `Shooter` und einem `Placer` - für die initiale Platzierung der Partikel verantwortlich ist, und ein sogenanntes `Program`, das sich beliebig konfigurieren lässt und das Verhalten der Partikel steuert.
All diese Komponenten gilt es für jeden Partikeleffekt zu erstellen, woraus sich in höchstem Maße redundanter Code ergäbe Um diesem entgegenzuwirken, existiert in Starjumper eine Klasse `ParticleEffectFactory`. Dem Factory Pattern folgend stellt sie Methoden bereit, die einen durch die Klasse `ParticleEffect` gekapselten Partikeleffekt erzeugen und je nach Situation konfigurieren.
`ParticleEffect` implementiert eine Art SStandardPartikeleffekt, dessen einzelne Komponenten und weitere Einstellungen sich durch im Konstruktor übergebene Objekte überschreiben lassen. Durch diese Aufteilung wird duplizierter Code vermieden und gleichzeitig die Verantwortung für das Erstellen von Partikeleffekten an einem Ort konzentriert.

5 Diskussion und Ausblick

Die relativ simpel gehaltene Klassenhierarchie und minimale Bindung zwischen den Komponenten hat sich während der Entwicklung als vorteilhaft erwiesen. Der Einflussbereich einer Klasse beschränkt sich in der Regel auf sie selbst, sowie die von ihr erstellen Kindknoten des Szenengraphen. So können Änderungen an Klassen vorgenommen werden, (fast) ohne die übrige existierende Funktionalität zu beeinflussen. Auch das Ausnutzen der von OpenSceneGraph bereitgestellten Möglichkeiten wie Update Callbacks oder Event Handler, die bei der Traversierung des Szenengraphen nacheinander aufgerufen werden, spart an einigen Stellen Code.
Ein zweifelhaftes Implementierungsdetail ist die Verwendung des Viewers als globale Variable. Auf den Viewer wird an diversen Stellen zugegriffen, sodass dessen Verfügbarkeit nicht optional war. Genauso wenig optimal war jedoch der Ansatz, eine Referenz auf den Viewer durch alle Ebenen durchzureichen, sodass schlussendlich die Entscheidung auf die Variante der globalen Variable fiel.
Gute Erfahrung haben wir hingegen mit der Implementierung des Players als Singleton gemacht. Neben den gesparten Ladezeiten ist dies die "bessere" Variante, globale Verfügbarkeit der Eigenschaften des Players zu realisieren.
Im Verlauf des Projekts hat sich gezeigt, dass vor allem statt der verwendeten Präprozessordirektiven für die Festlegung global konstanter Werte (wie etwa der Pfad der Spielermodelldatei) eine Überschreibbare Konfigurationsdatei sinnvoll gewesen wäre. Die hier spezifizierten Werte könnten dann zum Beispiel über Kommandozeilenargumente überschrieben werden, sodass ohne Neukompilierung des Projekts bestimmte Parameter - vor allem zu Testzwecken - schnell und unkompliziert angepasst werden könnten. Gleichzeitig würde stellenweise die Komplexität des Codes verringert bzw. die Übersichtlichkeit erhöht.

Abbildung 4. Klassendiagramm komplett