Cross Platform Game Development

ArrowShooting3000

Final Report

Magdalena Eibensteiner, Bianca Gotthart, Fabian Schmidt

Allgemeine Spielbeschreibung:

Es wurde ein einfaches Arrow Shooting Spiel entwickelt. Das Hauptaugenmerk lag nicht auf der Komplexität des Gamedesigns, sondern auf der Umsetzung für die verschiedenen Plattformen.

ArrowShooting3000 ist ein 2D Spiel, bei dem der Spieler an einer fixen Position im rechten unteren Eck des Fensters steht. Sobald das Spiel startet werden Ziele platziert, auf die der Spieler mittels Maus/Touch Input Pfeile abschießt, um sie zu treffen. Die Schussstärke des Pfeils hängt von der Dauer des Inputs ab. Vorteil dieses einfach gehaltenen Konzepts ist das statische Setup (fixierte Kamera) sowie nicht Vorhandensein von Spielerbewegungen.

Das Spiel wurde für die Plattformen iOS, Android und PC umgesetzt. Fokus der Umsetzung lag auf der Verwendung gemeinsamer Ressourcen für die verschiedenen Plattformen und der unterschiedlichen Umsetzung eines Games für Desktop- und Mobile- Plattformen. Sämtliche Game-Logik (Entities, einfache Physik, Score-System, Input-Weiterverarbeitung,…) wurde in C++ programmiert. Es werden auch die gleichen Grafiken verwendet soweit dies möglich ist, iOS benötigt für Animationen eigene Files.

Für iOS ist ein `.plist` File nötig, worin die einzelnen Frames und weitere Metadaten gespeichert sind und ein dazugehöriges SpriteSheet. Wenn mit cocos2d eine Animation (`CCAnimation`) aufgerufen wird, wird sowohl das Animated-Sprite als auch das `.plist` File mit der Animation verknüpft. Die Frames werden automatisch aus dem `.plist` File gerendert (Zugriff über das Setzen des Namens z.B.: target_1.png, target_2.png möglich, wird im plist-File definiert). Die Zuweisung der Bilder zur Animation passiert in einer Schleife wo auf das `.plist` File zugegriffen wird und die Bilder Frame für Frame in einem Array gespeichert werden.

Das Grundgerüst des Spiels wurde für jede Plattform individuell aufgebaut, dafür wurden plattformspezifische Engines (keine Editoren, sondern Frameworks) verwendet:

- iPhone (Objective-C): cocos2D for iPhone
- Android (Java): AndEngine
- PC (Java): Slick Engine

Für Android und PC wird als Schnittstelle JNI verwendet. Die Methoden die für die Game Logik notwendig sind werden als native Methoden in eigenen Java Klassen (Proxies) definiert. Mit Hilfe des Werkzeugs `javah` wird für jede dieser Klassen ein eigenes Headerfile erstellt, wo die Funktionsdeklaration definiert ist. Diese Headerdateien können dann in C++ inkludiert und weiterverwendet werden.

Die Proxy Klassen können auch um Methoden und Eigenschaften erweitert werden die für die Plattform spezifische Programmierung notwendig sind, z.B.: hinzufügen eines Sprites oder Images zum rendern der Grafiken,...

Eine iPhone App basiert auf der Sprache Objective C und hat als Basis C als Programmiersprache. Die Integration von C++ Klassen ist somit kein großes Problem. Die benötigten Klassen werden in die Objective C Klassen inkludiert. Die einzige Restriktion, die eingehalten werden muss ist das Verwenden einer speziellen Dateiendung. Standardmäßig gibt es ein `.h` File (Header) und ein `.m` File (Implementierung). Damit die Integration von C++ Klassen und somit die Erstellung und Verwendung von C++ Objekten und Funktionsaufrufen möglich wird, muss die Dateiendung des Implementierungs-Files auf `.mm` geändert werden.

Um in einem Array, das auf dem Konzept von Objective C beruht (`NSArray`), ein C++ Objekt zu hinterlegen, z.B.: für die Verwaltung der im Spiel verwendeten Pfeile, muss allerdings das Objekt in eine Wrapper-Klasse gespeichert werden, um darauf über das Array zugreifen zu können. Somit wurden für die iPhone Version ebenfalls Wrapper-Klassen für Pfeile und Ziele erstellt, um auf diese in der `update()` Methode zugreifen zu können.

Die Android Version des Spiels implementiert ebenso wie die Windows Version die plattformspezifischen Komponenten in Java im eigenen Programm. Zusätzlich gibt es Proxy-Klassen, die die Verbindung zu den Native- Aufrufen herstellt. Da Android jedoch Linux-basierend ist können keine DLL Files verwendet werden. Zur Umwandlung der C++ Klassen in eine Linux spezifische Library wurde ein entsprechendes Make-File und das Android Native Development Kid (NDK) verwendet. Unter Windows wird NDK benötigt um für Linux kompilieren zu können.

Ursprünglich geplante Umsetzung:

Anfänglich war die Umsetzung anders geplant. Die Game-Logik sollte in eigene Java-Klassen ausgelagert werden um sie anschließend in die jeweilige Plattform integrieren zu können. Um dies zu erreichen, wäre eine Verbindung zwischen Objective C und Java mittels Cross-Language-Aufrufen nötig gewesen.

Wir versuchten in Objective C eine Java VM zu initialisieren, in die die erstellten Java-Klassen geladen werden können, um sie dann im Objective C Code aufrufen zu können. Die Erstellung der VM funktionierte allerdings nicht, weil wir den Grund für die Fehlermeldung `Library not loaded. Reason: image not found` in XCode nicht finden bzw. beseitigen konnten.

Durch Recherche wurden einige verschiedene Lösungsansätze gefunden, mit denen wir das Problem allerdings nicht lösen konnten:

- Die Frameworks optional statt required machen
- Kompletten Rebuild und Cleanup machen
- Das Framework in die Build Phase einbinden
- Projekteinstellungen anpassen (“Include Libraries” und “Dynamic Library Install Name” setzen)
- Die Umgebungsvariable “`DYLD_LIBRARY_PATH`” anpassen
- Probleme mit 32/64 Bit- Versionen
- Probleme mit der JDK/JRE Version
- Probleme mit der Mac OS Version

Daher entschieden wir uns für eine alternative Herangehensweise an die Umsetzung, und implementierten die plattformunabhängigen Teile des Spiels in C++ um sie in Java und Objective C aufrufen zu können.

Tatsächliche Umsetzung:

Wie erwähnt entschieden wir uns dafür, möglichst viel Spiel Logik in unabhängige C++ Klassen auszulagern, um diese dann aus den verschiedenen Plattformen aufrufen zu können. Folgende Komponenten sind von uns einmalig in C++ implementiert worden:

`class vector:` Wird benötigt, um einfache Physikberechnungen wie Schwerkraft oder Flugbahn durchführen zu können. Sie stellt einige einfache Vektor Berechnungen zur Verfügung.

`class arrow:` Übernimmt die physikalischen Berechnungen des Pfeils. Sie berechnet mit Hilfe von Vektoren die Rotation, die Stärke und die Richtung, in die der Pfeil abgeschossen wird und übernimmt die Positionsberechnung anhand der aktuellen Geschwindigkeit und der Schwerkraft.

`class arrowhud:` Diese Klasse berechnet, wie schnell und wie weit sich der Hud unter dem Bogen auffüllen soll.

`class target` und **`class flyingtarget:`** Diese Klassen beinhalten die Koordinaten der Ziele und die Neuberechnung der Position. Außerdem kann man hier eine Pfeilposition übergeben, um eine einfache Kollisionserkennung durchzuführen.

`class player:` Enthält die Koordinaten des Spielers

`class score:` Speichert den aktuellen Punktestand ab und stellt ihn zur Verfügung.

`class terrain:` Diese Klasse kümmert sich um die zufällige Positionierung der Ziele unter Berücksichtigung der Position des Spielers.

Design Patterns

Singleton: Für die Score Klasse wurde das Singleton Pattern verwendet. Dadurch kann von überall im Programm, sowohl von den C++ Klassen als auch in den Java bzw. Objective C Klassen auf die gleiche Score Instanz zugegriffen werden. Dies ist nötig damit der Score, wenn ein Ziel getroffen wurde in den C++ Klassen erhöht und dann im Plattform spezifischen Code ausgelesen werden kann. *(Quelle: "Head First Design Patterns" - Autoren: Eric Freeman, Elisabeth Freeman, Kathy Sierra & Bert Bates - O'Reilly Media; 1 edition (November 1, 2004))*

Experience:

Das Einbinden der ausgelagerten Spiellogik in die verschiedenen Plattformen, haben wir uns wesentlich einfacher vorgestellt. Der Aufwand, auf jeder Plattform ein funktionierendes Grundgerüst zum Laufen zu bringen, das unsere ausgelagerten Klassen problemlos aufrufen kann, war sehr hoch und es hat lange gedauert, bis diverse Fehlermeldungen und andere Probleme aus dem Weg geräumt waren.

Des Weiteren ist uns aufgefallen, dass sobald die Grundgerüste mit den Native-Aufrufen funktionieren, die Erweiterbarkeit beim Hinzufügen neuer Klassen einerseits zwar sehr einfach ist, da die Klassen nur einmal geschrieben werden müssen, andererseits aber die Erstellung der native Headers und der Proxy Klassen auf den einzelnen Plattformen (Android und Desktop) nicht umgangen werden konnte.

Am meisten Probleme bereiteten uns jedoch die verschiedenen Koordinatensysteme. Während auf Windows und Android der Ursprungspunkt links oben ist, ist dieser in der cocos2d engine auf dem iPhone links unten. Außerdem ist der Ankerpunkt der Sprites, die die Position bestimmen, in den Windows und Android Engines in der linken oberen Ecke des Sprites, in cocos2d jedoch in der Mitte des Sprites.

Des Weiteren ließ sich der Drehpunkt, um den das Sprite rotiert wird, in cocos2d nicht extra verschieben, sondern ist ident mit dem Ankerpunkt, was vor allem bei der Drehung des Pfeils

ein großes Problem war. Durch diese Umstände mussten wir einige komplizierte Umrechnungen machen, an die wir im Vorfeld des Projektes überhaupt nicht gedacht hatten und die uns einiges an Zeit gekostet haben.

Falls wir wieder in so eine Situation kommen sollten, würden wir im Vorhinein auf diese Umstände achten, und entsprechende Umrechnungsmethoden implementieren, die uns später viel Arbeit erspart hätten.

Positiv herauszustreichen ist natürlich die jetzt vorhandene Änderungsfreundlichkeit. Sollten in den bestehenden ausgelagerten Klassen Änderungen vorgenommen werden, werden diese ohne weitere Anpassungen sofort in alle drei Plattformen übernommen.

Zeitplan:

Task	geschätzter Zeitaufwand	tatsächlicher Zeitaufwand
Recherche, Cross Language Calls	2 Tage	5 Tage (Variante 1: Basis Java + Variante 2: Basis C++)
Einarbeitung in die Plattformen und Engines	3 Tage	iPhone (1 Tag - da Vorkenntnisse vorhanden) PC (3 Tage) Android (3 Tage)
Implementierung der Game-Logik	4 Tage	3 Tage
Verknüpfung der Plattformen mit der Game-Logik	3 Tage	iPhone (4 Tage) PC (4 Tage) Android (5 Tage)