KÜRBIS AUF ALLEN KANÄLEN

Spiele von Windows Phone 7 für PC und Xbox zu kompilieren, ist mit XNA vergleichsweise einfach. Mit welchen Tools und Tricks dies auch für iOS und Android möglich ist, erklärt Daniel Springwald anhand des Multiplattform-Projekts PumpkinJumpin.

Häufig hört man über mobile Spiele, dass für eine Multiplattform-Fähigkeit entweder vieles doppelt programmiert oder eine (teure) Engine verwendet werden muss. Als wir im Frühjahr 2011 mit der Entwicklung unseres Knobelspiels »PumpkinJumpin« starteten, planten wir ursprünglich, das Spiel exklusiv für Windows-Phone-7-Geräte zu erstellen. Schnell zeigte sich jedoch, dass eine Portierung auf Apple- und Android-Geräte mit geringem Aufwand und vor allem ohne doppelte Code-Basis möglich ist.

XNA als Basis

Wie für XNA-Projekte üblich, erstellten wir für »PumpkinJumpin« als Erstes in Visual-Studio ein Windows-7-Phone-Projekt. Dabei entsteht neben dem eigentlichen Spiel automatisch ein zweites Content-Projekt, in welches später die Grafiken und Audiodateien abgelegt werden.

Mono macht's möglich

Wie kommt das XNA-Spiel aber nun auf die anderen Plattformen? Dass man ein XNA-Spiel leicht als PC- oder Xbox-Spiel neu kompilieren kann, ist bekannt – aber für iOS oder Android? Um diese Plattformen zu erschließen, mussten wir den Microsoft-Pfad ein wenig verlassen, konnten aber den bereits erstellten C#-Code komplett weiterverwenden. Das Zauberwort hierfür heißt »Mono« (www.mono-project.com) und ermöglicht es, .NET-Projekte auch für Android- oder Apple-Geräte zu kompilieren.

Wer Mono kennt, wird jetzt wahrscheinlich einwenden, dass damit primär Quellcode wiederverwendet werden kann. Oberflächen-Elemente wie zum Beispiel Forms oder WPF-Steuerelemente können gar nicht oder nur sehr sporadisch übertragen werden. Glücklicherweise verwendet XNA keinerlei Steuerelemente oder Layout-Manager. Stattdessen besteht das XNA-Spiel ausschließlich aus Quellcode, Grafik- und Sound-Ressourcen.

Für die Portierung unseres Spiels fehlte innerhalb von Mono natürlich die Funktionalität von XNA – also vieles aus dem Namespace »Microsoft.Xna.Framework«. Glücklicherweise gibt es auch hier Abhilfe: Andrew Russell hat diese Lücke mit seiner per Crowdfunding finanzierten und nun kostenfreien Mono-Bibliothek »EXEN« geschlossen. Bindet man den von ihm unter http://andrewrussell.net/exen bereitgestellten Quellcode in ein iOS- oder Android-Projekt ein, so ergänzt er die fehlenden XNA-Funktionen für die Anzeige von 2D-Grafiken, Sounds und Steuerung. Weil die Bibliotheken im Quellcode vorliegen, können vermisste Funktionen bei Bedarf auch einfach selbst ergänzt werden.

Der Level-Editor ist nur einen Steinwurf entfernt

Während der Entwicklung auf dem PC bietet XNA die Möglichkeit, das Fenster mit der Grafikausgabe mit normalen Windows-Fenstern zu kombinieren. Dadurch wird zum Beispiel das Erstellen eines Level-Editors sehr komfortabel: Im XNA-Fenster läuft das Spiel in Echtzeit, während man sich zur Auswahl von

Daniel stammt aus dem Herzen des Ruhrgebiets und kann auf 25 Jahre Erfahrung in der Computerspiele-Entwicklung zurückblicken. Bereits Anfang der 1990er-Jahre veröffentlichte er Grafik-Adventures und Jump & Runs. In jüngster Vergangenheit lag sein Fokus der Spieleentwicklung primär auf Videospielautomaten. www.springwald.de

Foto: © 2010 Stefanie Kusemann

Daniel Springwald
ist Geschäftsführer von Springwald Software.

Bausteinen oder zum Laden und Speichern von Levels der vorhandenen Windows-Steuerelemente bedienen kann (siehe **Abbildung 1**).

Bei »PumpkinJumpin« erstellten wir für den Level-Editor ein zusätzliches Projekt vom Typ »Windows-Steuerelemente-Bibliothek«. Dieses beinhaltet den Code für die Darstellung von Fenstern und Dialogen des Level-Editors. Hinzu kamen einige Projekte mit zentralen Datenstrukturen und gemeinsamem Code.

Ausgeführt wird in der Projektmappe immer das XNA-Code-Projekt – selbst beim Start als Level-Editor. In diesem Fall hat das Projekt die zusätzliche Aufgabe, die Windows-Fenster des Level-Editors bereitzustellen. Die Implementierung des Datenaustauschs zwischen dem XNA-Code und dem Editor ist nicht ohne Weiteres möglich, weil das XNA-Projekt für Windows-7-Phone kompiliert und nach dem Start entweder im Emulator oder auf dem echten Telefon ausgeführt wird.

Um trotzdem einen Datenaustausch zu ermöglichen, kann ein Klon des Spielprojekts angelegt werden. In der Visual-Studio-Projektmappe erstellt man dazu per Rechtsklick und Auswahl des Befehls »Kopie des Projekts für Windows erstellen« das entsprechende Windows-XNA-Projekt. Dabei kopiert Visual Studio nicht den Code, sondern referenziert dieselben C#-Dateien in einem neuen Projekt. Es gibt daher nur einen gemeinsamen Spiel-Quellcode. Anschließend existieren in der Projektmappe zwei ausführbare Projekte – eines für Windows Phone 7 und eines für Windows.

Damit die Funktionalität der Windows-Anwendung bei der Auslieferung des fertigen Spiels nicht stört, kann man die entsprechenden Programmteile im XNA-Quellcode per bedingter Kompilierung »#if WINDOWS« auf den Start als Windows-Anwendung beschränken. Dadurch bleiben sie für die Windows-Phone-7-Version im Verborgenen.

Wird das Spiel später im Editor-Modus gestartet, können neben dem XNA-Fenster noch weitere Fenster zum Laden/Speichern der Level, Auswahl von Level-Objekten und dergleichen dargestellt werden. Dem Windows-Phone-7-Spiel sind diese Fenster aber unbekannt. Damit bleibt XNA inklusive der Game-Loop die primär auszuführende Anwendung, was den Vorteil hat, dass das Spielprojekt jederzeit erstellt und ausgeliefert werden kann.

Tipps und Tricks

Die Beachtung einiger Dinge zu Anfang des Projekts kann für die spätere Portierung auf die weiteren Plattformen viel Zeit und Ärger sparen:

Sprite-Fonts

Die Verwendung von Schriftarten ist bei der aktuellen XNA-Version sehr komfortabel. Man muss nur den Namen der gewünschten Schriftart und die Höhe in Pixel angeben, um den Rest kümmert sich dann Visual Studio und erzeugt die entsprechenden Grafiken. Auch für

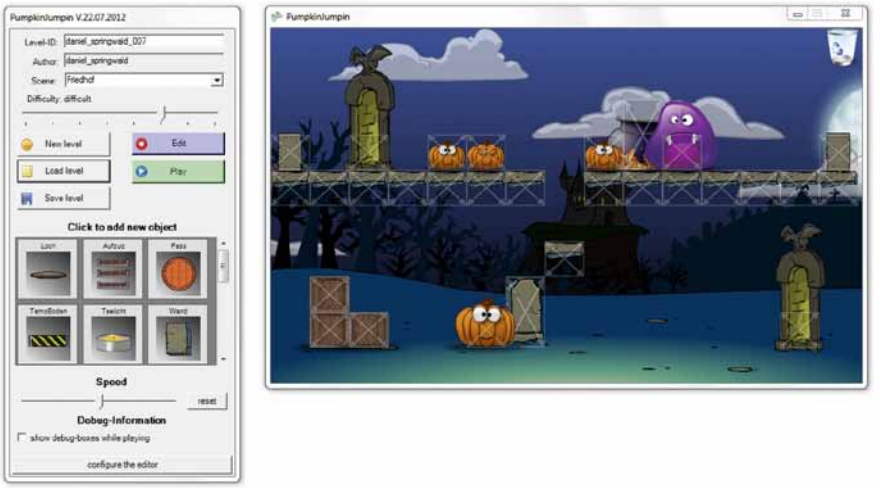

Abbildung 1: Der Level-Editor wird im Windows-Fenster angezeigt, das Spielfeld in der XNA-Umgebung.

das Zeichnen des Textes auf den Bildschirm existieren fertige Methoden, welche die gewünschte Zeichenkette entgegennehmen und die Grafiken an die richtigen Stellen zeichnen. Dieser Mechanismus existiert in der Mono-Umgebung leider nicht und wird auch nicht durch EXEN ergänzt. Daher ist es empfehlenswert, benötigte Sprite-Fonts extern als Grafik-Ressource zu erzeugen, so wie es bei früheren XNA-Versionen üblich war (siehe **Abbildung 2**).

Grafiken

Am besten berücksichtigt man beim Design der Grafiken bereits frühzeitig, dass das Spiel auf ganz unterschiedlichen Gerätetypen und Auflösungen funktionieren soll. Für uns bedeutete das, ausschließlich Vektorgrafiken zu gestalten. Wir konnten dadurch später für die unterschiedlichen Plattformen leicht PNG-Grafiken in mehreren Auflösungen erzeugen. Ein weiterer Vorteil der Vektorformate ist, dass Grafiken in Druckauflösung verfügbar sind, zum Beispiel zum Aufdruck auf Merchandising-Artikel oder für Presse-Informationen.

Bei iOS-Geräten funktioniert die automatische Skalierung von großen Grafiken sehr gut, sodass theoretisch eine einzelne, hoch aufgelöste Grafikversion für alle Geräte genügt. Unter Android-Mobiltelefonen mit niedriger Auflösung sieht eine große, herunterskalierte Grafik hingegen eher unschön aus. Wir haben in unserem Spiel daher für jede Grafik drei unterschiedliche Versionen erzeugt, welche entsprechend der Bildschirmauflösung des Geräts ausgewählt werden (siehe **Abbildung 3**).

Für große Displays verwenden wir eine hochauflösende Grafik und für mittelgroße Bildschirme eine verkleinerte Version mit leichter Unschärfe. Für sehr kleine Geräte empfiehlt sich eine gering aufgelöste Grafik mit starker Unschärfe. Die entsprechende Auswahl der Grafiken geschieht dann automatisch zur Laufzeit. Im Quellcode wird nur der grundsätzliche Name der Grafik-Ressource an-

Abbildung 2: Für die Darstellung von Schrift empfehlen sich die klassischen Sprite-Fonts, abgelegt in einer großen Grafik.

Abbildung 3: Alle Grafiken wurden in drei Auflösungen produziert, die passende wird je nach Anzeigeleistung des Geräts ausgewählt.

gegeben und je nach Display-Beschaffenheit um die Kennung »L«, »M« oder »S« ergänzt.

Bei Windows Phone hat sich für die optimale Auflösung eine Mischung aus den Empfehlungen für iOS und Android bewährt. Grafiken live zu skalieren führt im Windows Phone zu einem optisch guten Ergebnis, qualitativ vergleichbar mit iPad und iPhone. Allerdings braucht XNA unter Windows Phone verhältnismäßig lange für das Laden der entsprechenden Ressourcen. Der Emulator lädt die Grafiken aber vergleichsweise schnell, sodass man die langen Ladezeiten im ungünstigsten Fall erst auf der echten Hardware bemerkt.

Emulator und Testumgebung

Für die ersten Tests unseres Spiels war der Windows-Phone-Emulator die beste Wahl, weil er bis auf wenige Ausnahmen das Echtzeitverhalten der originalen Hardware abbildet.

Der Android-Emulator war für Tests unseres Spiels nicht sinnvoll zu gebrauchen, weil er das Programm nur in Zeitlupe ablaufen lässt. Für Android mussten wir daher bereits früh auf echte Endgeräte zurückgreifen. Der Emulator für Apple-Geräte ist ähnlich wirklichkeitsgetreu wie der Emulator für Windows Phone 7 und erlaubte umfangreiche Tests bereits am Rechner.

Die Auslieferung von »PumpkinJumpin« an unsere Betatester war bei Android am einfachsten, weil die Software nicht über Zertifikate an die Geräte-IDs gebunden werden muss. Dafür wiederum unterstützt MonoTouch auf iOS-Geräten inzwischen auch TestFlight, was die Verteilung sehr komfortabel macht.

Sounds und Musik

Bei Sound- und Musik-Ressourcen benötigten wir von Anfang an zwei parallele Formate: unkomprimierte Wave-Dateien für Windows Phone 7 und MP3-Dateien für iOS und Android. Die Auswahl der Bitrate und Frequenz für die MP3-Dateien ist dabei weitgehend beliebig und kann primär nach dem besten Kompromiss zwischen Qualität und Dateigröße gewählt werden.

Abbildung 4: Für die meisten Tests war der Windows-Phone-Emulator die beste Wahl.

Obfuskatoren und Ressourcen-Verschlüsselung

Bei einem Export für Android oder Windows Phone 7 sollte man wissen, dass das Programm (im Gegensatz zu iOS) in einem gut lesbaren Format ausgeliefert wird. Hier wird beim Build-Prozess zunächst Bytecode erzeugt und dieser wird erst auf dem Gerät zur Laufzeit in maschinennahe Befehle übersetzt. Der Bytecode enthält bis dahin noch viele Struktur- und Namensinformationen des vom Entwickler geschriebenen Quellcodes. Er kann mit entsprechenden Werkzeugen leicht wieder in gut lesbaren Quellcode umgewandelt werden.

Um das Durchstöbern des eigenen Programmcodes für neugierige Augen schwieriger zu machen, kann man bei Android recht unkompliziert einen sogenannten Obfuskator einsetzen. Dazu muss nur das Werkzeug »ProGuard« (http://proguard.sourceforge.net) in das Projekt integriert werden.

Für Windows Phone 7 stehen ebenfalls Obfuskatoren zur Verfügung. Den »Dotfuscator Windows Phone Edition« von PreEmptive Solutions legt Microsoft dem SDK gleich mit bei.

Weil die auszuliefernde Android-Datei eigentlich nur ein umbenanntes ZIP-Archiv ist, liegen auch die enthaltenen Grafiken offen vor. Wenn man diesen Zugriff verhindern möchte, kann man die Dateien verschlüsseln. Android unterstützt dies allerdings bisher nicht automatisiert – eine eigene Implementierung ist aber mit relativ wenig Aufwand möglich. Dazu muss man wissen, dass ein Android-Programm auf eine Grafik zugreifen kann, indem es von einem Stream liest. Dieser Stream enthält den binären Inhalt der ursprünglichen Ressourcen-Datei, zum Beispiel einer PNG-Datei. Für eine Absicherung sollten nun im ersten Schritt die einzubindenden Ressourcen-Dateien auf der Festplatte des Entwicklungsrechners verschlüsselt werden. Im Android-Programm bindet man dann statt des normalen Streams einen selbst programmierten ein. Dieser nimmt auf dem Weg von der Datei zum Programm die entsprechende Entschlüsselung vor. Weil der Zugriff auf den Stream sequenziell erfolgt, kann man den Ver- und Entschlüsselungsalgorithmus nahezu frei wählen.

Am Ende des Tages

Mit den genannten Mitteln konnten wir den Entwicklungsprozess von »PumpkinJumpin« ohne Notwendigkeit einer plattformübergreifenden Engine gestalten. Die Versionen für Android, Windows Phone 7, iPad, iPod und iPhone können alle gleichzeitig und sozusagen auf Knopfdruck produziert werden. Einen eigenen Level-Editor konnten wir ebenfalls leicht erstellen und komfortabel per ClickOnce ausliefern. Bei einer Änderung oder Erweiterung der Spiellogik muss Quellcode nur noch an einer einzigen Stelle, nämlich im Visual Studio für die Windows-Phone-7-Version angepasst werden.

Daniel Springwald