

Heinz Seyringer
Heinz Seyringer

Spielprogrammierung für PC, Handy und PDA

Plattformunabhängig programmieren
mit PPL



2 Spiele-Projekte

2.1 Einleitung

Nachdem wir im ersten Abschnitt die verschiedenen Möglichkeiten von PPL kennengelernt haben, schauen wir uns in diesem Kapitel an, wie wir diese Möglichkeiten für die Entwicklung von Spielen verwenden können. Die einzelnen Kapitel sind dabei weitgehend unabhängig voneinander und können daher in beliebiger Reihenfolge gelesen werden. In jedem Kapitel schauen wir uns verschiedene Techniken an, die wir für die unterschiedlichen Spieltypen nutzen können. Die Idee dabei ist, dass Sie die Anwendung der verschiedenen Techniken auf möglichst kompakte Art anhand eines praktischen Beispiels erlernen. Nun wünsche ich Ihnen viel Spaß bei der Entwicklung Ihrer ersten PPL-Spiele.

2.2 Winterwunderland

2.2.1 Überblick

In diesem Projekt entwickeln wir eine kleine Demo, bei der Schnee auf eine Winterlandschaft fällt, ein kleines Feuer brennt, ein golden glänzender Text angezeigt wird und im Hintergrund ein Weihnachtslied gespielt wird.

Um die Sache etwas interessanter zu gestalten, nehmen wir an, es sei ein relativ warmer Wintertag und die Schneeflocken würden schmelzen bevor sie den Boden erreichen. Schneeflocken im Vordergrund simulieren wir durch schön große Flokken, während wir für die weiter entfernten Schneeflocken nur kleine Sternchen verwenden.





Abbildung 2.1: Schnee fällt auf die Landschaft und im Vordergrund brennt ein Feuer.

2.2.2 Das Teilchensystem

Als Ausgangspunkt für unser Projekt verwenden wir wieder das Game-API-Skelett (USERS/PPL/GAMEAPI/GAMEAPI SKELETON), welches bereits den Basiscode samt Initialisierung der Game-API enthält. Als Erstes kümmern wir uns dabei um den Schnee. Die Schneeflocken sollen vom oberen Bildschirmrand nach unten fallen, das heißt, wir müssen für jede Schneeflocke ein Teilchen generieren und diesem eine positive Y-Geschwindigkeit geben. Denken Sie daran, dass der Koordinatenursprung in der linken, oberen Ecke ist und somit eine positive Y-Geschwindigkeit notwendig ist, wenn etwas nach unten fallen soll.

Damit die Schneeflocken nicht wie von Geisterhand plötzlich am oberen Rand erscheinen, sondern wirklich in den Bildschirm hineinfallen, müssen wir sie etwas oberhalb des Bildschirmrands erzeugen (mindestens eine Spritehöhe oberhalb des Bildschirmrands). Und um es möglichst natürlich wirken zu lassen, müssen auch Schneeflocken vom linken und rechten Rand in den Bildschirm hineinwehen und über den Rand hinauswehen können.

Die Schneeflocken über den Bildschirmrand hinauszubewegen, ist recht einfach: Wir definieren den Bildschirmrand nicht als Grenze, an der es Kollisionen gibt. Um von außen in den Bildschirm hineinwehen zu können, müssen wir auch links und rechts vom Bildschirm Schneeflocken erzeugen, die sich in den sichtbaren Bereich hineinbewegen können. Wichtig ist dabei aber, dass sie trotzdem oberhalb des oberen Bildschirmrands erzeugt werden, da sie ja in jeder beliebigen Höhe von der Seite hineinkommen sollen.

Als Teilchen verwenden wir Sprites von Schneeflocken. Um diese aber in der `GameProc` verwenden zu können, müssen wir die Sprites zuerst in der Funktion `WinMain` vorbereiten. Bei der Gelegenheit ergänzen wir gleich auch die Gravitationskraft, welche unsere Teilchen zusätzlich nach unten ziehen wird, und wir entfernen die überflüssige `ShowFPS`-Anzeige aus dem Standardcode. Damit sieht unsere `WinMain`-Funktion nun folgendermaßen aus:

```

01 func WinMain
02 // neues Formular für die Game-API erzeugen
03 h$ = newform({Winterwunderland}, {GAPIClass}, &mainproc);
04 // Formular auf dem Bildschirm anzeigen
05 ShowWindow(h$, SW_SHOW);
06
07 // Initialisiere die Game-API
08 InitGameAPIEx(h$, &GameProc, 240, 320, false, 5, 60);
09
10 Global(Schnee1$, Schnee2$);
11
12 // große Schneeflocke laden
13 Schnee1$ = LoadSprite(AppPath$ + "Schnee1.bmp", G_RGB(0, 0, 0), 1, -1,
                        NULL);
14 SetSpriteAlpha(Schnee1$, 255); // Schneeflocke ist undurchsichtig
15 MoveSprite(Schnee1$, -25, 0); // Schneeflocke aus dem Display
                                raus schieben
16
17 // kleine Schneeflocke laden
18 Schnee2$ = LoadSprite(AppPath$ + "Schnee2.bmp", G_RGB(0, 0, 0), 1, -1,
                        NULL);
19 SetSpriteAlpha(Schnee2$, 255); // Schneeflocke ist undurchsichtig
20 MoveSprite(Schnee2$, -25, 0); // Schneeflocke aus dem Display
                                raus schieben
21
22 SetGravity(0.05); // Gravitation fürs Teilchensystem setzen
23
24 return (true);
25 end;

```

In den Zeilen 11 und 16 laden wir die große und die kleine Schneeflocke als Sprite. Damit man die Sprites aber nicht die ganze Zeit sieht, sondern nur die Teilchen, die wir daraus generieren, müssen wir die Sprites außerhalb des sichtbaren Bildschirms positionieren (Zeilen 13 und 18). Wenn die Schneeflocken schmelzen, werden sie auch immer durchsichtiger, bis sie schließlich ganz verschwunden sind. Diese Eigenschaft steuern wir über das sogenannte *Alpha-Blending*, mit dem wir die Transparenz durch abnehmende Werte (0 bedeutet unsichtbar) erhöhen können. Wenn die Teilchen aber erzeugt werden, sollen sie noch völlig undurchsichtig sein, das heißt, für die Sprites müssen wir den maximalen Alpha-Wert von 255 setzen, damit die Schneeflocken am Anfang wirklich völlig undurchsichtig sind. In Zeile 20 setzen wir dann noch die Gravitation auf einen eher kleinen Wert, damit die Schneeflocken langsam nach unten gezogen werden. In Zeile 10 definieren wir die Sprites als globale Variablen, damit wir von der Funktion `GameProc` auch auf sie zugreifen können.

Wenn wir das Programm jetzt starten, sieht man allerdings noch gar nichts, da unsere Sprites außerhalb des Bildschirms positioniert sind und wir noch keine Teilchen daraus generiert haben. Dies holen wir nun nach, indem wir in unserer Funktion `GameProc` den entsprechenden Code im `WM_Timer`-Abschnitt ergänzen:

```
01 func GameProc(hWnd$, Msg$, wParam$, lParam$)
02     case (Msg$)
03         WM_PAINT:
04             g_clear(0);
05             RenderSprites; // Zuerst Sprites zeichnen, da sonst die
06             RenderParticles(0); // Teilchen vom Hintergrund verdeckt werden
07
08
09         WM_TIMER:
10             Speed$ = (Random(10) + 1) * 0.02; // horizontale Geschwindigkeit
11             velx$ = RandomSet(Speed$, -Speed$);
12
13             vely$ = (Random(20) + 1) * 0.02; // vertikale Geschwindigkeit
14
15             if (CountParticles(s$) < 150) // maximale Anzahl Schneeflocken
16                 if (random(10) < 1) // Verhältnis kleine:große Schneeflocken
17                     NewParticle(s$, Schnee1$, Random(300) - 30, -20, 0, 0,
18                         velx$, vely$, random(1300), false, SpriteAlpha(Schnee1$),
19                         false, false); // große Schneeflocke erzeugen
20                 else
21                     NewParticle(s$, Schnee2$, Random(300) - 30, -20, 0, 0,
22                         velx$, vely$, random(1300), false, SpriteAlpha(Schnee2$),
23                         false, false); // kleine Schneeflocke erzeugen
24             end;
25         end;
26
27         if (g_key.vkA$) // Programm beenden falls Sondertaste A
28             auf dem PDA gedrückt wurde
29             PostMessage(hWnd$, WM_CLOSE, 0, 0);
30         end;
31     end;
32     return (true);
33 end;
```

Nachdem wir Sprites und Teilchen gleichzeitig darstellen wollen, ist es sehr wichtig, in welcher Reihenfolge wir die Befehle zum Rendern (Zeilen 5 und 6) aufrufen. Wenn wir zuerst `RenderParticles` und danach `RenderSprites` verwenden, werden die Sprites über die Teilchen gezeichnet (die zuletzt aufgerufene Funktion zeichnet über alles, was zuvor gezeichnet wurde). Damit würde der Hintergrund (ein Sprite) über die Teilchen gezeichnet und man würde von den Schneeflocken nichts mehr sehen.

Damit sich die Schneeflocken nicht alle genau gleich bewegen, geben wir jeder eine zufällige Geschwindigkeit in horizontaler und vertikaler Richtung. Nachdem die vertikale Geschwindigkeit immer positiv sein muss (andernfalls würden die Schneeflocken ja nicht in den sichtbaren Bereich fallen), erhalten wir die Y-Geschwindigkeit einfach, indem wir eine positive Zufallszahl zwischen 1 und 20 generieren und dann noch mit 0.02 multiplizieren, um die Geschwindigkeit auf einen realistischen Wert zu verlangsamen. Damit die vertikale Geschwindigkeit immer größer als 0 ist (`Random(20)` würde Werte zwischen 0 und 19 liefern), addieren wir in Zeile 13 noch 1 zum Zufallswert hinzu.



Abbildung 2.2: Die Schneeflocken fallen bereits wie echter Schnee.

In horizontaler Richtung machen wir genau das Gleiche, nur erlauben wir hier positive und negative Geschwindigkeiten. Der `RandomSet`-Befehl in Zeile 11 weist der horizontalen Geschwindigkeit zufällig den Wert `Speed$` oder `-Speed$` zu. Natürlich hätte man das Gleiche auch durch die Zeile

```
10 Speed$ = (Random(20) + 1 - 10) * 0.02; // horizontale Geschwindigkeit
```

erreichen können, aber da der Befehl `RandomSet` in vielen Situationen recht nützlich sein kann, ist es nicht schlecht, ihn zu kennen.

In den Zeilen 15 bis 21 erzeugen wir unser Teilchensystem. Zuerst schauen wir mit der Funktion `CountParticles`, wie viele Teilchen sich im Moment in unserer Gruppe befinden. Nachdem die Teilchen eine endliche Lebenserwartung haben, wird ihre Anzahl mit der Zeit kontinuierlich abnehmen. Haben wir weniger als 150 Teilchen in unserer Teilchengruppe, dann erzeugen wir neue Teilchen. Möchten Sie gerne ein dichteres Schneegestöber, dann müssen Sie nur diesen Wert erhöhen beziehungsweise für weniger Schnee entsprechend senken.

Als Nächstes entscheiden wir dann, ob wir eine große oder eine kleine Schneeflocke erzeugen möchten. Nachdem der Großteil der Schneeflocken im Hintergrund fallen, benötigen wir mehr kleine als große Schneeflocken. Die `If`-Abfrage in Zeile 16 sorgt dafür, dass im Schnitt nur jede zehnte Schneeflocke eine große Schneeflocke ist. Möchten Sie mehr große Schneeflocken verwenden, müssen Sie anstelle von zehn einfach einen kleineren Wert einsetzen.

Die Erzeugung der Schneeflocken erfolgt mit dem Befehl `NewParticle`, wobei zunächst angegeben wird, dass alle Sprites zur Gruppe `s$` gehören sollen und als Sprite wird für die großen Schneeflocken `Schnee1$` verwendet und für die kleinen Schneeflocken `Schnee2$`. Nachdem das Standard Game-API Code-Skelett eine Bildschirmbreite von 240 Pixel verwendet (wenn Sie es geändert haben, müssen Sie an dieser Stelle die Position auch entspre-

chend ändern), verwenden wir als X-Position einen zufälligen Wert zwischen -30 und 270 (also jeweils 30 Pixel über die tatsächliche Bildschirmbreite hinaus). Nachdem auch unsere großen Schneeflocken nicht höher als 20 Pixel sind, genügt es, wenn wir bei der Erzeugung der Flocken eine Y-Position von -20 verwenden.

Da wir bereits die Gravitation aktiviert haben, müssen wir den Schneeflocken keine zusätzliche Beschleunigung mit auf den Weg geben und setzen einfach nur ihre vertikale und horizontale Geschwindigkeit. Als Lebenserwartung wählen wir einen zufälligen Wert zwischen 0 und 1299. Da wir den Fade-Parameter nicht auf -1 gesetzt haben, schmelzen die Teilchen auf ihrem Weg nach unten und die Lebenserwartung gibt somit an, wie weit die Schneeflocken nach unten fallen, bis sie völlig geschmolzen sind. Möchten Sie die Schneeflocken weiter fallen lassen, dann erhöhen Sie die Lebenserwartung einfach.

Wenn Sie Teilchensysteme verwenden, sollten Sie in der Regel immer darauf achten, für die Lebenserwartung einen zufälligen Wert oder zumindest für unterschiedliche Teilchen auch unterschiedliche Lebenserwartungen zu verwenden. Würden wir in unserem Beispiel allen Teilchen die gleiche Lebenserwartung zuweisen, so würde der Schnee zunächst ganz normal fallen, aber sobald die Teilchen ihre Lebenserwartung erreicht haben, würden alle Schneeflocken gleichzeitig verschwinden und von oben gewissermaßen eine Lawine neuer Schneeflocken in den Bildschirm hereinfallen, was nicht gerade besonders realistisch aussehen würde. Haben die einzelnen Teilchen aber unterschiedliche Lebenserwartungen, so werden kontinuierlich Teilchen verschwinden und neue Teilchen erzeugt werden, was sehr viel besser aussieht.

2.2.3 Hintergrund und Dekorationen

Was uns jetzt noch für eine hübsche Winterszene fehlt, ist natürlich die Winterlandschaft, und um das Ganze noch etwas zu verschönern, setzen wir ein hell loderndes Feuer in die Landschaft und platzieren einen glänzenden Text über das Ganze.

Als ersten Schritt bereiten wir die Grafiken für die Animationen (Feuer und glänzenden Text) vor. Am schönsten ist es natürlich, wenn wir ein richtiges Lagerfeuer vor schwarzem Hintergrund mit der Videokamera aufnehmen können und dann direkt diese Videoaufnahme als Vorlage verwenden. Der schwarze Hintergrund wird später einfach transparent gemacht. Da wir aber nicht ohne Weiteres mit den uns zur Verfügung stehenden Mitteln einen schönen, schwarzen Hintergrund für unser Lagerfeuer erstellen können, ist es in der Praxis am einfachsten, wir nehmen entweder ein Raytracing-Programm, das ein Feuer simulieren kann, oder wir zeichnen selbst ein Feuer in einem Zeichenprogramm.



Abbildung 2.3: Das animierte Feuer besteht aus einer Folge von Einzelbildern, die wir nebeneinander in einer einzigen Bitmap-Grafik anordnen.

Beim Zeichnen des Feuers sollten wir als Hintergrundfarbe eine Farbe wählen, die ähnlich zum Hintergrund in unserem Bild ist. Da wir eine Schneelandschaft entwickeln, ist es besser, einen weißen Hintergrund als einen schwarzen Hintergrund zu verwenden. Natürlich könnte man jetzt sagen, dass die Hintergrundfarbe eigentlich egal ist, weil wir sie ja ohnehin transparent machen, aber das Problem ist, dass wir unschöne Ränder am Rand der Flammen bekommen würden, wenn wir beim Zeichnen Funktionen verwenden, die Antialiasing nutzen (also die Farben mit den Nachbarfarben mischen).

Als Startpunkt empfiehlt es sich, zunächst eine Flamme zu zeichnen. Beim nächsten Einzelbild benutzen wir dann diese Flamme als Vorlage (zum Beispiel indem wir sie in einen halbtransparenten Layer platzieren oder direkt ins Bild kopieren) und verändern diese dann geringfügig. Dieses Vorgehen wiederholen wir jetzt so oft, bis wir genug Einzelbilder erstellt haben. Ich habe für dieses Beispiel 21 Einzelbilder gemalt, aber Sie können natürlich ganz nach ihrem Geschmack auch mehr oder weniger Bilder zeichnen. Es sollten aber nicht zu wenige (weniger als zehn) sein, weil man sonst zu deutlich bemerkt, dass nur ein paar Bilder wiederholt werden. Beim Zeichnen der Bilder sollten Sie auch darauf achten, dass der Unterschied zwischen den einzelnen Bildern immer in etwa gleich groß ist. Wenn es einen größeren Sprung zwischen zwei Bildern gibt als zwischen den anderen, so wird dem Spieler dies auffallen und er wird merken, dass sich diese Stelle immer wiederholt.

Das Gleiche führen wir noch für den Text durch. Zunächst geben wir den Text beispielsweise in einem gelb-oranger Ton aus, damit die Grundfarbe schon ähnlich wie Gold aussieht und dann kopieren wir diesen Text 20 Mal. Im nächsten Schritt benutzen wir die Aufhellfunktion unseres Zeichenprogramms, um einige Stellen des Textes aufzuhellen, damit es aussieht, als hätten wir Lichtreflexionen auf dem goldenen Text. Bei den Folgebildern verschieben wir die Position dieser hellen Stellen, so dass es aussieht, als würde ein Licht am reflektierenden Text vorbeibewegt werden. Wenn wir alle Bilder kreiert haben, setzen wir sie nebeneinander in eine Bitmap und speichern diese.

Der schwierigste Teil ist das Zusammensetzen der Einzelbilder in eine große Bitmap. Das Problem dabei ist nämlich, dass sich die Bilder einerseits nicht überlappen dürfen und andererseits aber auch keine Pixelpalte zwischen zwei benachbarten Bildern frei bleiben darf. Ideal ist es, wenn Ihr Zeichenprogramm ein magnetisches Raster unterstützt und Sie die Grafiken anhand dieses Rasters ausrichten können. Ein Freeware-Zeichenprogramm, welches diese Funktion bietet, ist beispielsweise *GIMP*, welches auch für verschiedene Betriebssysteme (Windows, Linux, ...) angeboten wird.

Ob Sie die Grafiken richtig nebeneinander platziert haben, sehen Sie einerseits daran, dass es bei der letzten Grafik keine Pixelpalte zu viel oder zu wenig geben darf, und andererseits bleiben im Spiel die Grafiken nur dann an ihrer Position, wenn sie sauber nebeneinander kopiert wurden. Überlappen sich einzelne Grafiken oder gibt es Freiraum zwischen den Einzelbildern, so wird es im Programm aussehen, als bewege sich die Animation nach links oder rechts. In dem Fall muss man die Einzelbilder noch einmal korrekt nebeneinander platzieren.

Nachdem wir nun die Grafiken vorbereitet haben, können wir sie in der Funktion `WinMain` als Sprites laden und an den entsprechenden Stellen positionieren:

```
55 func WinMain
56 // neues Formular für die Game-API erzeugen
57 h$ = newform((Winterwunderland), {GAPIClass}, &mainproc);
58 // Formular auf dem Bildschirm anzeigen
59 ShowWindow(h$, SW_SHOW);
60
61 // Initialisiere die Game-API
62 InitGameAPIEx(h$, &GameProc, 240, 320, false, 5, 60);
63
64 Global(Schnee1$, Schnee2$, Hintergrund$, Feuer$, Text$);
65
66 // Feuer als Sprite laden
67 Feuer$ = loadsprite(AppPath$ + "Feuer.jpg", G_RGB(255, 255, 255), 21,
68                     100, NULL);
69
70 MoveSprite(Feuer$, 40, 265); // Feuer positionieren
71
72 // Text als Sprite laden
73 Text$ = loadsprite(AppPath$ + "Text.bmp", G_RGB(0,0,0), 15, 400, NULL);
74 MoveSprite(Text$, 45, 0); // Text positionieren
75
76 // Hintergrund als Sprite laden
77 Hintergrund$ = loadsprite(AppPath$ + "Hintergrund.jpg", G_RGB(0, 0, 0),
78                          1, 0, NULL);
79 MoveSprite(Hintergrund$, 0, 0); // Hintergrund positionieren
80
81 // Feuer vor den Hintergrund holen
82 // SETSPRITEORDER(Feuer$, SpriteOrder(Hintergrund$) + 1);
83 // Text vor den Hintergrund holen
84 // SETSPRITEORDER(Text$, SpriteOrder(Hintergrund$) + 2);
85
86 // große Schneeflocke laden
87 Schnee1$ = LoadSprite(AppPath$ + "Schnee1.bmp", G_RGB(0, 0, 0), 1, -1,
88                      NULL);
89 SetSpriteAlpha(Schnee1$, 255); // Schneeflocke ist undurchsichtig
90 MoveSprite(Schnee1$, -25, 0); // Schneeflocke aus dem sichtbaren Bereich
91 // hinaus schieben
92
93 // kleine Schneeflocke laden
94 Schnee2$ = LoadSprite(AppPath$ + "Schnee2.bmp", G_RGB(0, 0, 0), 1, -1,
95                      NULL);
96 SetSpriteAlpha(Schnee2$, 255); // Schneeflocke ist undurchsichtig
97 MoveSprite(Schnee2$, -25, 0); // Schneeflocke aus dem sichtbaren Bereich
98 // hinaus schieben
99
100 SetGravity(0.05); // Gravitation fürs Teilchensystem setzen
101
102 return (true);
103 end;
```

Probieren Sie das Programm jetzt einmal aus, aber kommentieren Sie bitte die Zeilen 79 und 81 aus, bevor Sie das Programm starten. Was sehen Sie?



Abbildung 2.4: Das Hintergrund-Sprite verdeckt das Feuer und den Text.

Die Schneeflocken fallen vor unserem Hintergrund herunter, aber der Text und das Feuer fehlen, obwohl wir beides richtig geladen und positioniert haben. Wo liegt also das Problem?

Wenn wir mehrere Sprites verwenden (in diesem Fall den Hintergrund, Text und Feuer), dann muss PPL entscheiden, welche Sprites weiter vorne dargestellt werden und welche Sprites weiter hinten. Diese Entscheidung trifft PPL basierend auf der Reihenfolge, in der die Sprites geladen werden: Sprites, die später geladen werden, sind weiter vorne und überdecken somit die Sprites, die schon früher geladen wurden. In unserem Beispiel wurde der Hintergrund absichtlich an der falschen Stelle geladen, um diesen Effekt zu zeigen. Die einfachste Lösung wäre in diesem Fall, dass wir den Hintergrund einfach als Erstes laden.

In vielen Fällen ist es aber nicht ganz so einfach und insbesondere bei komplexeren Spielen können wir nicht immer mit Sicherheit im Vorhinein bestimmen, in welcher Reihenfolge die Sprites geladen wurden. Für diesen Fall bietet uns PPL die Funktion

```
SPRITEORDER (Sprite$);
```

welche uns die Z-Position des Sprites angibt und die Funktion

```
SETSPRITEORDER (Sprite$, Z-Position$);
```

mit welcher wir die Z-Position des Sprites bestimmen können. Die Bezeichnung *Z-Position* wird verwendet, weil die X- und Y-Koordinaten entlang der Breite und Höhe des Bildschirms verlaufen und die Richtung aus dem Bildschirm heraus oder hinein dementsprechend die Z-Koordinate des Sprites ist.

Wenn wir also ein Sprite vor einem anderen positionieren möchten, müssen wir gar nicht wissen, an welcher Z-Position unser Sprite sein muss, um vor dem anderen Sprite darge-

stellt zu werden, sondern es reicht, wenn wir den Z-Wert des anderen Sprites plus eins verwenden und schon ist das Sprite vor dem anderen. Genau das machen wir in den Zeilen 79 und 81. Wir addieren beim Text und beim Feuer einen Wert zur Position des Hintergrund-Sprites hinzu und schon sind sie vor dem Hintergrund.



Abbildung 2.5: Mit der richtigen Sprite-Reihenfolge sieht man auch das Feuer und den Text.

Vielleicht ist Ihnen aufgefallen, dass wir bei den Grafiken manchmal BMP-Grafiken und manchmal die kompakten JPG-Grafiken verwenden, wobei wir insbesondere bei den großen Animationen meistens die BMP-Dateien verwenden. Auf den ersten Blick mag das recht paradox erscheinen, weil es gerade bei den großen Dateien ja wichtig wäre, dass wir Speicher nicht nutzlos verschwenden, indem wir BMP-Grafiken anstelle der sehr viel kleineren JPG-Grafiken verwenden.

Der Grund dafür liegt in der Komprimierung der JPG-Grafiken. Durch deren Komprimierung kann es gewisse Artefakte geben, die sich beispielsweise in recht störenden Pixeln in der Nähe von Farbübergängen zeigen können. Wenn wir den Hintergrund dann transparent machen wollen, stören diese Punkte und Linien natürlich sehr.



Abbildung 2.6: JPG-Bilder können bei hoher Kompression Artefakte entlang der Umrisse enthalten, welche bei transparenten Grafiken stören.

Wenn man die Kompression nicht zu hoch wählt, sind diese Artefakte nur schwach und man kann sie in einem Nachbearbeitungsschritt wieder entfernen (was wir bei der Feuer-Animation gemacht haben). Damit kann man dann auch JPG-Grafiken für größere Animationen verwenden.

2.2.4 Die Hintergrundmusik

Um die weihnachtliche Stimmung dieser Demo abzurunden, ergänzen wir noch die Hintergrundmusik ganz am Ende der Funktion `WinMain`:

```

95 // Hintergrundmusik laden und in einer Schleife abspielen
96 w$ = LoadSound(AppPath$ + "Leise rieselt der Schnee.wav", false);
97 PlaySound(w$); // Musik abspielen
98 SetLoop(w$, true); // Musik in Endlosschleife spielen

```

Die Verwendung von Musik funktioniert ganz analog zur Verwendung von Sprites. Zuerst laden wir die Musik in den Speicher und erhalten dadurch ein Handle, den wir an die verschiedenen Musikbefehle übergeben müssen, damit diese wissen, welche Musik wir abspielen wollen. Das ist besonders dann wichtig, wenn wir neben der Hintergrundmusik noch alle möglichen Soundeffekte verwenden. Mit der Funktion `PlaySound` können wir WAV- oder MOD-Dateien abspielen und mit `SetLoop` können wir bestimmen, ob das Musikstück wiederholt werden soll, nachdem es abgespielt wurde.

2.3 Wasserfall, Springbrunnen und Geysir

2.3.1 Überblick

In diesem Kapitel schauen wir uns an, wie wir Teilchensysteme verwenden können, um Wassereffekte wie Wasserfälle, Springbrunnen oder Geysire zu implementieren. Wir werden für jeden dieser drei Typen ein eigenes Teilchensystem verwenden und mithilfe der Physik-Engine simulieren wir das Verhalten der einzelnen Wassertropfen.



Abbildung 2.7: Für die Implementierung des Wasserfalls, Springbrunnen und Geysirs wird jeweils ein eigenes Teilchensystem verwendet.