Nordpfalzgymnasium
Kirchheimbolanden

2008/2009

Besondere Lernleistung
in
Informatik

Thema: Entwicklung einer 2D Grafikbibliothek

Verfasser: Andreas Stöckel
Kurs: Informatik
Kursleiter: Herr Bernhard Cuntz
Abgabetermin: 24. Juni 2009

Erzielte Note: _____ In Worten: _____

Unterschrift des Kursleiters: _____

Die ersten Computerspiele bewegten sich ausschließlich in zwei Dimensionen. Erst mit dem Aufkommen von leistungsfähigeren Computerprozessoren war es in den 1990er Jahren erstmals möglich dreidimensionale Spiele zu entwickeln. Um 3D-Grafiken immer schneller und detailreicher berechnen zu können, haben sich heutzutage 3D-Beschleunigerkarten etabliert, die viele tausende Dreiecke in der Sekunde darstellen können. Mit dem Boom der 3D Hardware wurde jedoch die Weiterentwicklung von 2D Grafikschnittstellen vernachlässigt — diese sind nach wie vor auf dem Stand des Jahres 2000. Da es sich bei 2D jedoch nur um einen Sonderfall von 3D handelt, liegt es nahe eine leistungsfähige Grafikbibliothek, die sich auf das 3D-hardwarebeschleunigte Darstellen von 2D-Grafiken spezialisiert, zu entwickeln.

Inhaltsverzeichnis

1 Grundlagen

1.1 Einleitung

1.1.1 Geschichtliche Entwicklung der Grafikausgabe

Die ersten Computerspiele fanden nur in zwei Dimensionen statt. Auch die aufkommenden graphischen Benutzeroberflächen verwendeten 2D-Elemente. Musste die Grafikausgabe zunächst komplett auf dem Hauptprozessor vorberechnet werden (wobei bei komplexeren Ausgaben nicht an Echtzeit zu denken war), so implementierten die Grafikkarten später bestimmte Ausgaberoutinen, die es ermöglichten 2D Grafikoperationen (zum Beispiel das Zeichnen von Linien, Rechtecken und Bildern) schnell auszuführen. Mit zunehmender Rechenleistung der Prozessoren gab es Anfang der neunziger Jahre erste 3D-Spiele. Auch diese berechneten die gesamte Szene auf dem Hauptprozessor, was entsprechend langsam war und nur detailarme 3D-Grafiken ermöglichte. So entwickelten sich auch im 3D-Grafikbereich 3D-Beschleunigerkarten, die den Hauptprozessor entlasteten und detailreichere Grafiken mit zahlreichen Effekten ermöglichten. 1999 kamen erstmals Grafikkarten mit der so genannten TnL-Pipeline (Transform and Lightning) auf dem Markt [10] - hierbei kümmert sich der Grafikprozessor nicht nur um die Rasterisierung (d.h. das Umwandeln der geometrischen Daten in einzelne Pixel), sondern auch um die Transformation und Beleuchtung der 3D-Modelle im Raum.

Abbildung 1.1: Screenshot der Spiele „The Elder Scrolls: Arena“ (1994) und „The Elder Scrolls: Oblivion“ (2006): Während „Arena“ unter DOS die gesamte Grafikausgabe auf dem Hauptprozessor durchführt, verwendet „Oblivion“ moderne Direct3D 9c Grafikhardware.

Mit dem Aufkommen der 3D-Hardware wurde jedoch der 2D-Grafikbereich nach und nach vernachlässigt. So wurde Microsofts 2D-Grafikschnittstelle „DirectDraw“ mit Versionsnummer 7 im Jahr 2000 eingestellt [8]. Zudem verwendet DirectDraw bei vielen

Grafikoperationen, wie zum Beispiel dem transparenten Zeichnen von Bildern nach wie vor den Hauptprozessor, was entsprechend langsam ist.
3D-Beschleunigerkarten hingegen sind beim Zeichnen von transparenten Objekten kaum langsamer — auch Operationen wie das Drehen und Verzerren von Grafiken gehören zu ihrem Standardrepertoire.

1.1.2 Warum 2D?

Man kann sich nun Fragen, was man überhaupt mit 2D-Grafiken anfangen will. Verschafft man sich einen Überblick über die erscheinenden Spieletitel, so findet man fast ausschließlich Spiele mit 3D-Grafiken.
Allerdings werden besonders im Hobbyentwicklerbereich gerne 2D-Spiele entwickelt, da die dafür benötigten mathematischen Grundlagen nicht sonderlich kompliziert sind und viele Menschen Probleme mit dem Zeichnen von gut aussehenden 3D-Objekten haben.
Zudem gibt es nach wie vor sehr gute Spielekonzepte, die sich problemlos in 2D umsetzen lassen: Jump'n'Runs (vgl. „Mario“), Rollenspiele (vgl. „Diablo“), Click and Point Adventures (vgl. „Monkey Island“) oder Strategiespiele (vgl. „Age of Empires“) sind nur einige Beispiele.
Außerdem werden 2D-Grafiken nicht nur für Spiele benötigt, sondern auch bei Simulationen oder allgemein Anwendungen mit graphischer Benutzerschnittstelle (GUI, „Graphical User Interface“).

1.1.3 2D-Grafiken mittels 3D-Hardware - Das Andorra 2D Projekt

Bei 2D handelt es sich ausschließlich um einen Sonderfall von 3D - lediglich die Tiefenkomponente (Z-Achse) muss nicht verwendet werden. Daher liegt es nahe, zur Ausgabe von 2D-Grafiken die 3D-Grafikhardware zu verwenden und von den zahlreichen Möglichkeiten zur beschleunigten Ausgabe der Grafiken zu partizipieren.
Ältere Bibliotheken (zum Beispiel „DelphiX“) setzen auf den oben beschriebenen, veralteten 2D-Schnittstellen (in diesem Fall „DirectDraw“) auf, was die oben beschriebenen Nachteile mit sich bringt.
Der Zugriff auf die 3D-Hardware ist jedoch relativ kompliziert und besonders Anfänger sind damit schnell überfordert. Auch für fortgeschrittene Entwickler lohnt es sich nicht jedes mal „das Rad neu erfinden zu müssen“, wenn schnell eine Softwarelösung entwickelt werden muss. Es ist also eine Schnittstelle von Nöten, die die 3D-Hardware entsprechend kapselt und einfache Funktionen und Klassen zur Ausgabe von 2D-Grafiken bereitstellt.
Aus diesem Grund entschloss ich mich dazu, das „Andorra 2D“-Projekt ins Leben zu rufen, welches im Nachfolgenden beschrieben wird. „Andorra 2D“ ist eine Bibliothek für die Programmiersprache „Object Pascal“ beziehungsweise die „Delphi Language“. Für nähere Informationen zum Projekt siehe auch Anhang A.3. „Andorra 2D“ soll im Endeffekt die folgenden Aufgaben übernehmen, welche in den nächsten Kapiteln näher beschrieben werden:

- **Kapselung der 3D-API**
- **Grafikausgabe**
  - Ausgabe (und Laden) von Bildern mit verschiedensten Effekten (Rotation, Skalierung, Transparenz)
  - Ausgabe von einfachen geometrischen Objekten (Linien, Rechtecke, Ellipsen, Kreise, Polygonzüge)
  - Ausgabe von Texten
- **Bereitstellen von „erweiterten Techniken“** basierend auf der „Grafikausgabe“
  - Ein einfaches „Spiele-Framework“ (die „Sprite Engine“)
  - Wiedergabe von Videos
  - Graphische Benutzer Schnittstelle (GUI)
- **Erweiterbarkeit** (hinzufügen benutzerspezifischer Module)
  - Eigene Grafik- und Videoformate
  - Eigene GUI-Komponenten

Andorra 2D beschränkt sich jedoch nicht auf die reine Darstellung von 2D-Grafik — Da es auf der 3D-Hardware aufbaut können durchaus 3D-Objekte in das Spiel integriert werden, was zum Beispiel beim Darstellen von Charakteren, die sonst aus vielen verschiedenen Blickwinkeln gezeichnet werden müssten, von Vorteil ist.
Da Andorra 2D mittlerweile recht komplex ist, ist es nicht möglich in diesem Dokument jedes einzelne Modul im Detail zu beschreiben. Daher können jeweils nur die wichtigsten Funktionsweisen skizziert werden. Für weitergehende Informationen und die eigentliche Anwendung der Bibliothek sei auf den recht gut dokumentierten Quellcode bzw. die Andorra 2D Internetseite (siehe Materialverzeichnis) und die dort vorhanden Tutorials/Dokumentation verwiesen.

1.2 Die Grafik-Schnittstelle

Der Zugriff auf die 3D-Hardware eines Computers ist in modernen Betriebssystemen nur über spezielle 3D-Grafikschnittstellen möglich.
Hierbei bildet die Grafikschnittstelle eine so genannte Hardwareabstraktionssebene (auch engl. Hardware Abstraction Layer (HAL) genannt), die es dem Entwickler erspart, selbst jeden auf dem Markt erhältlichen Grafikchip manuell ansteuern zu müssen. Stattdessen setzt die Grafikschnittstelle abstrahierte Befehle (die Menge dieser Befehle bildet eine so genannte API, engl. für „Application Programmers Interface“, zu deutsch Anwendungsentwicklerschnittstelle) in Zusammenarbeit mit dem entsprechenden Gerätetreiber in entsprechende Hardwareaufrufe um.

1.2.1 Verschiedene Grafikschnittstellen im Vergleich

Die bekanntesten und am häufigsten verwendeten Grafikschnittstellen, mit deren Hilfe direkt die 3D Grafikkarte angesteuert werden kann, sind „OpenGL“ und „Direct3D“. Es existieren noch weitere 3D-Grafik-APIs, die jedoch meistens ein Nischendasein haben und nur auf spezielle Produkte wie z.B. Spielekonsolen abgestimmt sind.

1.2.2 DirectX

DirectX ist eine von Microsoft entwickelte, objektorientierte Multimediabibliothek, die bis jetzt in der Version 10.1 vorliegt. Version 11 ist momentan in Entwicklung. Der Grundgedanke von DirectX ist es, Entwicklern einen schnellen Zugriff auf die Multimediahardware zu ermöglichen. DirectX ist in mehrere Teilbibliotheken gegliedert. Die wichtigsten davon im Überblick:

- Direct3D ist für die Darstellung von hardwarebeschleunigten 3D-Grafiken verantwortlich. Der Name „DirectX“ wird fälschlicherweise häufig als Synonym für „Direct3D“ verwendet.
- DirectDraw ist eine Bibliothek um 2D-Grafiken auszugeben. Sie ermöglicht hardwarebeschleunigte Manipulation des Grafikspeichers [3]. Weitergehende Grafikeffekte werden nicht hardwarebeschleunigt. Seit DirectX 7 (Jahr 2000) wird DirectDraw nicht mehr erweitert [8].
- DirectSound ist eine Bibliothek, mit deren Hilfe direkt auf die Audiohardware zugegriffen werden kann, wobei deren Echtzeiteffekte (Mehrkanalton, Reverb, Echo, Equalizer, Dopplereffekt, etc.) verwendet werden können.
- DirectInput dient zum einheitlichen Zugriff auf verschiedenste Eingabegeräte (Tastatur, Maus, Joystick, Lichtgriffel, Lenkräder, etc.)

Im Folgenden werden wir uns mit der Teilbibliothek „Direct3D“ beschäftigen.
Eine Anwendung, die Direct3D verwendet, ruft zunächst die einheitliche Direct3D Bibliothek, die von Microsoft zur Verfügung gestellt wird, auf. Diese leitet die Befehle über die von Microsoft vorgeschriebene Grafikkartentreiberschnittstelle [4] an die Grafikkarte weiter. Dies hat den Vorteil, dass die Schnittstelle zwischen Anwendung und Direct3D immer gleich ist.
In Andorra 2D wird Direct3D in der Version 9c eingesetzt werden, da mit Direct3D 10 viele potentielle Benutzer ausgeschlossen würden: Direct3D 10 setzt gewisse Hardwarespezifikationen zwingend voraus und ist erst ab Windows Vista verfügbar. Es ist nicht vertretbar, dass eine Bibliothek, die zu großem Teil für das Anzeigen von einfachen 2D Inhalten ist, nur auf der neusten Hardware läuft.

1.2.3 OpenGL

OpenGL ist eine offene Spezifikation für die Ansteuerung von 3D-Grafikhardware. Diese Spezifikation liegt momentan in der Version 3.1 vor [2]. Im Gegensatz zu Direct3D ist OpenGL nicht Objektorientiert, sondern Prozedural aufgebaut.
Viele Betriebssysteme besitzen nach der Installation nur eine eingeschränkte oder keine OpenGL-Unterstützung. Erst mit der Installation eines Grafikkartentreibers wird - je nach Hersteller - eine entsprechende OpenGL-Bibliothek zur Verfügung gestellt, die der OpenGL Spezifikation entspricht. Eine Anwendung, die OpenGL verwendet ruft also direkt die Bibliothek des Grafikkartenherstellers auf. Aus dieser Tatsache folgt leider, dass man sich als Programmierer (im Gegensatz zu Direct3D) nie sicher sein kann, dass jede OpenGL-Implementierung genau das gleiche Verhalten zeigt.
Ein weiterer Nachteil von OpenGL ist, dass es seit seiner Veröffentlichung im Jahre 1992 immer nur um so genannte „Extensions“ erweitert wurde, wodurch die API mehr einem Flickenteppich, als einer Bibliothek, die aus einem „Guss“ entstanden ist, gleicht.
Mit der neuesten Version 3.1 wurde dieser Nachteil durch komplette Neuentwicklung zwar entfernt, aber ähnlich zu Direct3D 10 wird die OpenGL 3.1 Spezifikation nur von relativ neuer Hardware unterstützt. Der größte Vorteil von OpenGL gegenüber Direct3D ist jedoch die Offenheit der Schnittstelle. Somit gibt es OpenGL auf fast jeder Plattform. Es existieren sogar komplett offene Implementierungen, die es ermöglichen OpenGL Applikationen über einen Softwarerender auf Systemen ohne 3D-Karte auszuführen.

1.2.4 Fazit

Wie wir gesehen haben, haben beide Grafiksysteme ihre Vor- und Nachteile. Möchte man seine Applikationen auf möglichst vielen Systemen zum Laufen bringen, so fällt die Wahl auf OpenGL. Jedoch liefert nicht jeder Grafikkartenhersteller (besonders bei „Onboard“ Chips) eine OpenGL Bibliothek mit, die auch nicht unbedingt alle „Extensions“ unterstützen.
Direct3D hingegen ist nativ nur für Microsoft Windows verfügbar.
Beide Grafikbibliotheken benötigen einen nicht unerheblichen Vorbau, der darüber entscheidet, welche Grafikfunktionen vom Grafikchip unterstützt werden.
Für Andorra 2D wäre es also am besten, wenn man über ein Pluginsystem ohne weiteres zwischen beiden Grafiksystemen wechseln könnte. Dies würde außerdem ermöglichen, einfach Unterstützung für neue Versionen der Grafiksysteme hinzuzufügen, was bei der aktuell sehr schnellen Entwicklung im Grafikbereich von hoher Priorität ist. Auch könnte man zum Beispiel ein Softwarerendering[1] Plugin bereitstellen, über das Andorra 2D generell auf jeder beliebigen Plattform und jedem Betriebssystem funktioniert.
Auch ist Andorra 2D dann für neue Entwicklungen auf dem Hardwaremarkt gerüstet: Während andere Grafikbibliotheken durch ungenügende Trennung des Codes für die

[1] Unter „Softwarerendering“ versteht man (genauso wie es früher geschah) das Berechnen der kompletten Szene auf dem Hauptprozessor.

Grafik-API mit dem Rest der Bibliothek mit der API mit altern [2] und so unbrauchbar werden, muss für Andorra 2D nur ein neues Plugin geschrieben werden, um es an neue Hardware anzupassen.

1.3 Die Rendering-Pipeline

Um die Pluginstruktur zu realisieren, ist es wichtig, die Funktionsweise der 3D-Hardware zu kennen, nach der sich auch die abstrahierenden Funktionen des Plugins richten sollten.
Seit Jahren hat sich auf dem Grafikkartenmarkt die so genannte „Transform and Lightning“ (TnL) Rendering Pipeline etabliert [10]. Diese beschreibt einen festen Pfad, auf dem die Daten, die von unserer Anwendung an die Grafikkarte gesendet wurden, bearbeitet und schließlich auf dem Monitor dargestellt werden.
Erwähnenswert ist, dass sich auf neueren Grafikkarten Teile der Rendering Pipeline durch eigene Programme ersetzen lassen, die dann direkt auf der Grafikkarte ausgeführt werden. Somit lassen sich manche grafischen Effekte auf dem stark parallelisierten Grafikchip sehr effizient erzeugen [3].
Der Ablauf des Renderingvorgangs ist in Abbildung 1.2 schemenhaft dargestellt.

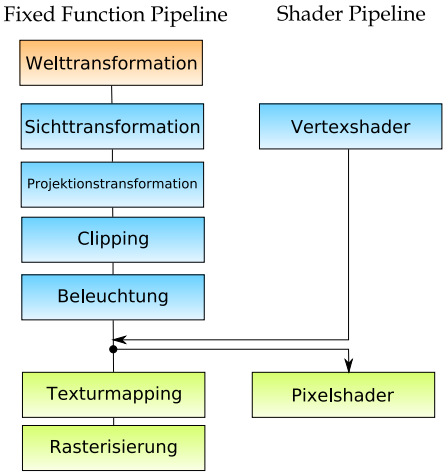


Abbildung 1.2: Die Rendering-Pipeline eines modernen Grafikprozessors [6].

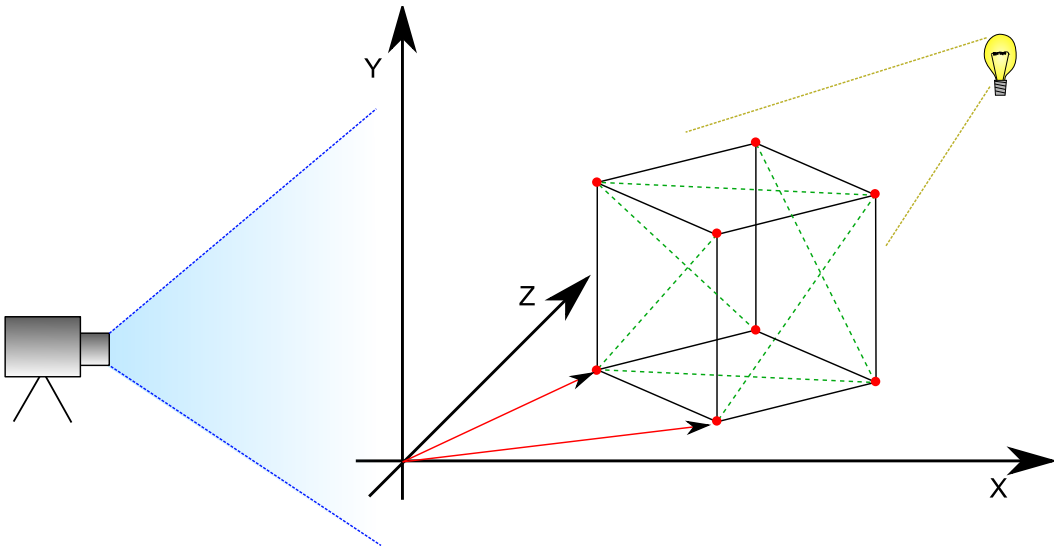


Abbildung 1.3: Eine 3D-Szene, wie sie innerhalb der Grafikkarte abgelegt ist: Das Modell (oftmals auch „Mesh“ genannt, in diesem Fall ein Würfel) ist durch Vektoren, die vom Koordinatenursprung ausgehen (angedeutet durch die roten Pfeile/-die roten Punkte) dargestellt. Da die Grafikkarte jedoch nur mit Dreiecken umegehen kann, muss der Würfel in Dreiecke unterteilt (trianguliert) werden (grüne Linien). Die Kamera im Bild bestimmt, aus welcher Position die Szene betrachtet wird (Beobachtungspunkt). Zusätzlich wird über ihr „Objektiv“ bestimmt, wie die Szene auf den Film (bei der Grafikkarte: den Bildschirm) projeziert wird. Außerdem beleuchtet eine Lampe den Würfel, dessen Seiten so mit unterschiedlicher Helligkeit dargestellt werden.

1.3.1 Repräsentation der 3D-Daten in der Grafikkarte

Soll ein komplexes 3D-Modell (also ein Würfel, ein Mensch, ein Baum, eine Teekanne etc.) auf dem Bildschirm dargestellt werden, so erhält die Grafikkarte dazu zunächst eine Menge aus „Vertices“ (Singular Vertex, von lat. Wirbel). Ein Vertex ist im Grunde genommen ein Vektor (hier ein Punkt im Raum), an den jedoch noch weitere Informationen wie eine Farbe, Normalenvektor und Texturkoordinaten angebunden sind (siehe Listing „Vertexdefinition“ und Abbildung 1.3).

Soll nun ein Modell dargestellt werden, so wird eine Kette (Array) dieser Vertices an die Grafikkarte gesandt. Grafikkarten können im Allgemeinen nur drei verschiedene Arten von Primitiven darstellen: Einzelne Punkte, Linien und Dreiecke. Modelle mit einer komplexeren Oberfläche müssen daher aus Dreiecken aufgebaut werden. Vielecke mit mehr als drei Punkten (Polygone) müssen zunächst trianguliert, d.h. in Dreiecke unterteilt, werden [4]. Damit die Grafikkarte die Vertices sinnvoll miteinander verbinden kann, muss ihr mitgeteilt werden, wie dies geschehen soll. Folgende Modi stehen dafür zur Verfügung:

- **Dreiecksliste:** Jeweils drei Vertices aus dem Vertexarray werden miteinander zu einem Dreieck verbunden. Dies ist die am häufigsten verwendete Methode.
- **Dreieckszüge:** Jeweils drei Vertices aus dem Vertexarray werden miteinander zu einem Dreieck verbunden - die nachfolgenden Vertices werden an die vorhergehenden Vertices angeschlossen (siehe Abbildung 1.4). Besonders bei Objekten wie Zylindern ist diese Zeichenmethode zu verwenden.
- **Dreiecksfächer:** Der erste Punkt des Vertexarrays dient als Mittelpunkt — die nachfolgenden Vertices werden jeweils mit dem Mittelpunkt zu Dreiecken verbunden. Somit lassen sich einfach Kreise darstellen.
- **Linienliste:** Jeweils zwei Punkte werden zu einer Line verbunden.
- **Linienzug:** Alle Punkte im Vertexarray werden miteinander zu einer durchgehenden Linie verbunden.
- **Punktliste:** Alle Vertices werden als einzelner Punkt (Pixel) im Raum dargestellt.

Gerade bei dem am häufigsten verwendeten „Dreieckslisten“ Zeichenmodus passiert es, dass viele Vertices mehrfach an die Grafikkarte gesendet werden müssten, wenn man geschlossene Oberflächen darstellen will. Um dies zu verhindern wird jeder Vertex nur

[2] Als Beispiel seien hier die Bibliotheken „DelphiX“ und „PhoenixLib“ (`http://www.phoenixlib.net/`, Version vom 15.08.2007) genannt

[3] Diese Technik wird in diesem Dokument nicht weiter beschrieben, auch wenn Andorra 2D sie unterstützt. In den Dateien „AdShaderClasses.pas“, „AdShader.pas“, sowie den entsprechenden Modulen der Grafikplugins ist jedoch die entsprechende Implementierung zu finden.

[4] Ein entsprechender Algorithmus (von mir implementiert) kann hier gefunden werden: `http://wiki.delphigl.com/index.php/Ear_Clipping_Triangulierung`

Vertexdefinition

```
type
  //Vektoren mit drei Komponenten werden im Nachfolgenden
  //für Beschreibung der Positon und des Normalenvektors
  //verwendet.
  TAdVector3 = record
    x, y, z: Single;
  end;

  //Ein Vektor mit zwei Komponenten wird für die
  //Beschreibung der Texturkoordinaten verwendet.
  TAdVector2 = record
    x, y: Single;
  end;

  //Farben bestehen aus 4 Komponenten Rot, Grün,
  //Blau und Transparenz (Alpha). Die Werte dürfen
  //zwischen Null und Eins liegen.
  TARGBColor = record
    a, r, g, b: Single;
  end;

  TAdVertex = record
    //Die Position des Punktes.
    Position: T3DVector;
    //Ein senkrecht zur beschriebenen Oberfläche stehender,
    //normalisierter Vektor, der zur Lichtberechnung
    //verwendet wird.
    Normal: T3DVector;
    //Die Texturkoordinaten, die dem Vertex einen Punkt auf
    //einem zweidimensionalen Bild zuordnet.
    TextureCoords: T2DVector;
    //Die Vertexfarbe.
    Color: TARGBColor;
  end;
```

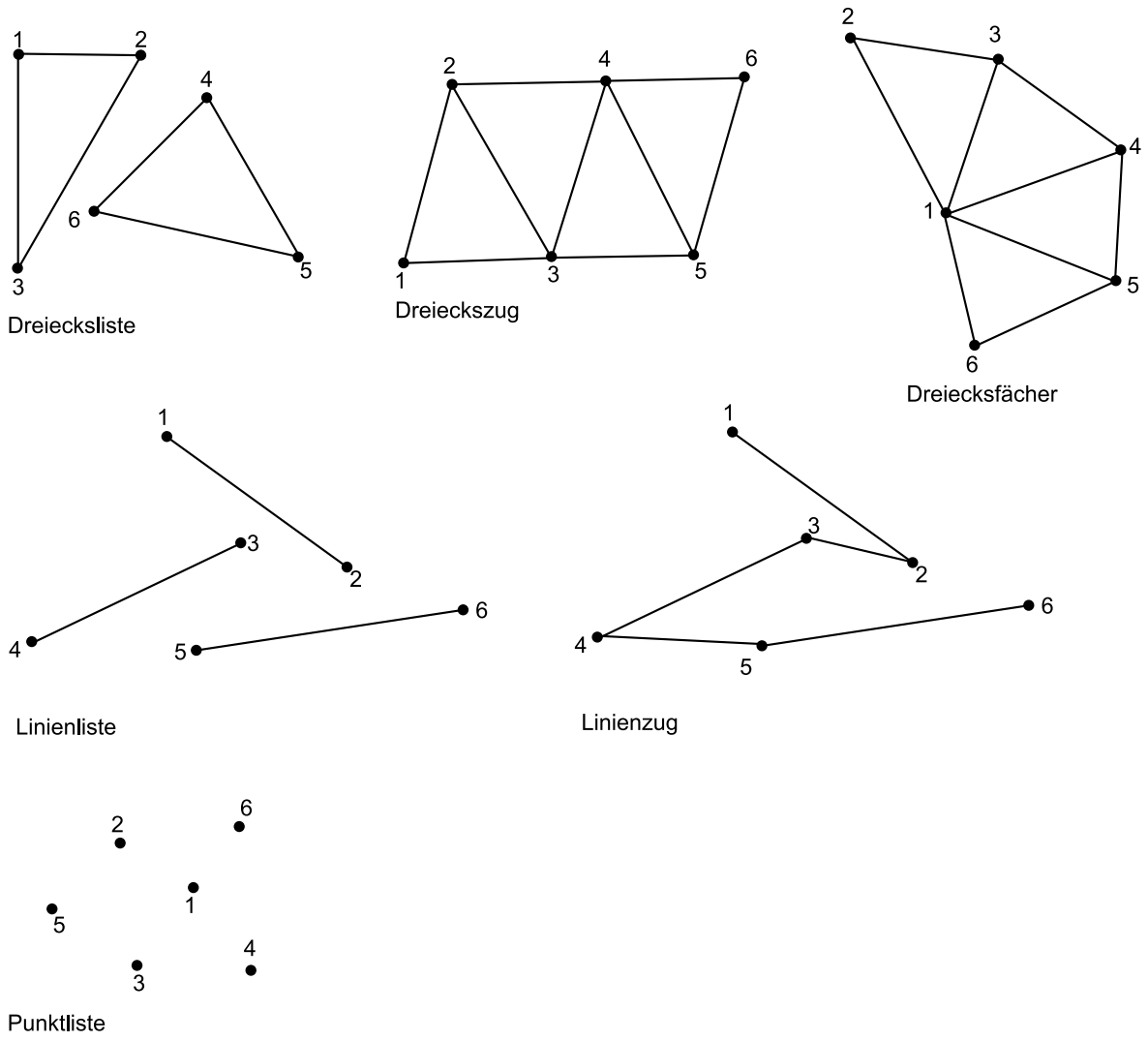


Abbildung 1.4: Die verschiedenen, von der Grafikkarte unterstützten, Primitivenmodi

einmal an die Grafikkarte gesendet, zusätzlich wird aber ein so genannter „Indexbuffer“ mitgegeben. In diesem ist die Reihenfolge, in der die Vertices gezeichnet werden, vorgegeben.

1.3.2 Vertextransformationen

Wurden die meistens um den Koordinatenursprung konstruierten 3D-Modelle an die Grafikkarte gesendet, so müssen die Vertices zunächst entsprechend der darzustellenden Szene im Raum positioniert, an der Kamera ausgerichtet (Transformation) und schließlich auf die zweidimensionale Monitoroberfläche abgebildet werden (Projektion). Beides geschieht über Multiplikation der Positionsvektoren der Vertices $\vec{v}$ mit entsprechenden Tranformations- und Projektionsmatrizen:

$$\vec{v_{trans}} = \vec{v} \cdot M_{obj} \cdot M_{view} \cdot M_{proj}$$

Diese rechenaufwendigen Multiplikationen müssen nicht auf dem Hauptprozessor ausgeführt werden — stattdessen kümmert sich eine spezielle Einheit auf dem Grafikprozessor um die Multiplikation. Dies hat den weiteren Vorteil, dass die Vertexdaten meist im Grafikspeicher gehalten werden können und nur die Matrizen für das Transformieren der Objekte ausgetauscht werden müssen.

Im nächsten Schritt werden die Vertices, sofern dies gewünscht ist, beleuchtet: Dazu wird die Position der Lichtquelle $\vec{p}$ von der des Vertices $\vec{v}$ abgezogen und normalisiert. Der so entstandene Vektor $\vec{r_0}$ wird skalar mit dem Normalenvektor des Vertices $\vec{n}$ multipliziert. Der entstehende Faktor wird mit der Vertexfarbe $\vec{C}$ multipliziert.

$$\begin{aligned} \vec{r} &= \vec{v} - \vec{p} \\ b &= \vec{r_0} \bullet \vec{n} \\ \vec{C_{bel}} &= \vec{C} \cdot b \end{aligned}$$

1.3.3 Rasterisierung

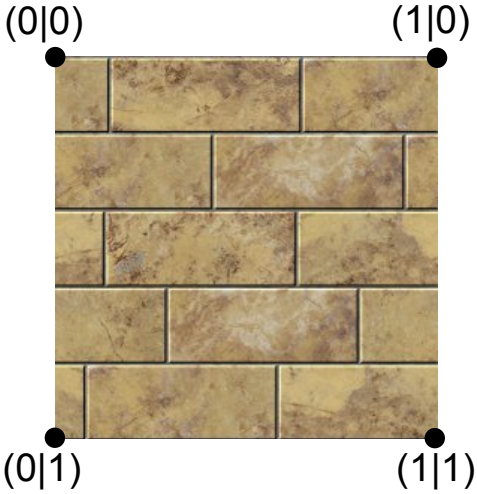


Abbildung 1.5: Abbildung der Texturkoordinaten

Bisher sind die Modelldaten nach wie vor „nur“ als mathematisches Konstrukt auf der Grafikkarte vorhanden und noch nicht auf dem Monitor bzw. dem Framebuffer[5] dargestellt. Diesen letzten Schritt übernimmt die Rasterisierungseinheit der Grafikkarte. Zunächst entscheidet der Grafikprozessor, ob ein anzuzeigendes Dreieck überhaupt sichtbar ist. Ein Dreieck ist für die Grafikkarte dann sichtbar, wenn dessen Positionsvektoren im Uhrzeigersinn (je nach Einstellung auch gegen den Uhrzeigersinn) angeordnet sind: Dieses Aussortieren wird auch „Culling“ genannt. Ist das Dreieck sichtbar, so geht die Grafikkarte alle sichtbaren Pixel des Dreiecks durch und berechnet für jeden Pixel eine Farbe: Zwischen den einzelnen Vertices interpoliert die Grafikkarte zunächst eine Mischfarbe aus den angegebenen Vertexfarben $\vec{C_m}$. Ist eine Textur eingebunden (insgesamt können bis zu acht Texturen gleichzeitig eingebunden und auf einem Dreieck überlagert werden) so wird zwischen den Texturkoordinaten (siehe Abbildung 1.5) der Vertices interpoliert und die Farbe des angegebenen Pixels aus der Textur geladen. Die endgültige Pixelfarbe $\vec{C_t}$ berechnet sich nun wie folgt (die Farben werden hierbei als 4-Dimensionaler Vektor aufgefasst, die jeweiligen Komponenten sind auf einen Wert zwischen 0 und 1 heruntergebrochen):

$$\vec{C} \;=\; \vec{C_m} \cdot \vec{C_t}$$

Nach der Projektion der Dreiecke auf die 2D-Ebene geht deren Tiefeninformation (Z-Achse) verloren. Hierdurch kann sich folgende Situation ergeben: Seien Zwei Objekte A (nahe an der Kamera) und B (weit von der Kamera entfernt) gegeben. Objekt A wird zeitlich vor Objekt B gezeichnet. Nun überdeckt Objekt B trotzdem Objekt A. Gelöst wird dieses Problem durch einen so genannten Z-Buffer: Ist dieser eingeschaltet, so speichert die Grafikkarte für jeden Pixel zusätzlich zur Farbe die Entfernung des Pixels zur Kamera. Die Farbe des Pixels wird nur dann ersetzt, wenn der neue Pixel näher an der Kamera ist.

1.3.4 Fazit

Wir wissen nun, wie der Bildaufbau in der Grafikkarte von statten geht und welche Informationen benötigt werden. Daher können wir nun beurteilen, welche Objekte für das Grafikplugin benötigt werden:

[5] Der „Framebuffer“ beinhaltet das eigentliche Bild, welches in periodischen Abständen zum Bildschirm geschickt wird

- **Zeichenoberfläche**: Wir benötigen ein Objekt, dass die Zeichenfläche an sich erstellt und verwaltet. Kameraposition (dargestellt durch die „Viewmatrix“) und Projektionsmatrix (diese kümmert sich, wie oben beschrieben, um das perspektivische Darstellen der Grafiken, das „Objektiv“ der Kamera) werden hier eingerichtet.
- **Modelle:** Die eigentlichen Objektdaten/Vertexarrays werden in diesem Objekt zusätzlich zu deren Objekttransformationsmatrix und der damit verknüpften Textur gespeichert.
- **Texturen:** Enthalten und verwalten die Bilder, die auf die Vertices gelegt werden.

Mit diesen drei Objekten/Objekttypen kann der Großteil der Grafikfunktionalität einer Grafikkarte abgedeckt werden.

2 Die fundamentale Gestaltung

2.1 Die Abstraktionsebenen

2.1.1 Abstraktion - Was ist das?

Beim Entwickeln von Software können viele Probleme durch Abstraktion gelöst werden: Das eigentliche Problem wird dazu analysiert und in mehrere Sinneinheiten unterteilt. Anschließend wird eine Schnittstelle, über welche die Sinneinheiten miteinander Daten austauschen, festgelegt. Dies hat den Vorteil, dass die einzelnen Sinneinheiten an sich einfacher zu entwickeln und daher weniger Fehleranfällig sind. Zudem können einzelne Sinneinheiten später einfach ausgetauscht werden, ohne das komplette Programm neu entwickeln zu müssen.
Diesen Vorteile wollen auch wir uns bei der Entwicklung von Andorra 2D nicht verschließen.

2.1.2 Abstraktionsebenen in Andorra 2D

Beim Programmieren von modularen Bibliotheken hat es sich für mich als sinnvoll erwiesen, die Bibliothek in drei Abstraktionsebenen zu unterteilen: Eine system-/hardwarenahe Abstraktionsebene, eine Funktionalitätsebene für die eigentliche Basisfunktionalität und eine Anwendungsebene für spezifische Anwendungsfälle.

Die Grafik-API Abstraktionsebene

Beim Überblicken der verschiedenen 3D-Grafikschnittstellen, sind wir zu dem Schluss gekommen, das unsere 2D-Grafikbibliothek am besten mehrere dieser Schnittstellen unterstützen sollte.
Um dies zu ermöglichen, müssen diese Grafikschnittstellen von unserer Grafikbibliothek aus über eine einheitliche Schicht von Befehlen/Objekten angesprochen werden. Diese Schicht bildet somit eine erste Abstraktionsebene, auf die unsere Grafikbibliothek aufbauen wird. Welche Objekte zur Verfügung stehen sollten, haben wir ebenfalls schon geklärt.

2D-Funktionalitätsebene

Würde unsere Bibliothek nur die unterste Grafik-API Abstraktionsebene bereitstellen, so hätten wir keine 2D- sondern allenfalls eine 3D-Grafikbibliothek. Durch das hinzufügen einer weiteren Abstraktionsebene wird die eigentliche 2D Funktionalität bereitgestellt.

Diese Ebene besitzt einfach zu verwendende Klassen, die in der Lage sind 2D Zeichenoperationen mit nur wenigen Befehlen durchzuführen. Somit wird zum Beispiel der 2D Befehl „Zeichne ein Bild an Bildschirmkoordinate (100 | 200)“ in das Erstellen eines aus vier Vertices bestehenden 3D-Modells, das Binden der Textur, setzen einer Verschiebungsmatrix und den abschließenden Zeichenvorgang umgesetzt.
Außerdem vereinfacht diese Abstraktionsebene weitere wichtige Funktionen, wie zum Beispiel das Erstellen einer Zeichenoberfläche oder das Laden von Texturen aus vielen verschiedenen Dateiformaten.
Diese Abstraktionsebene wird den Hauptkern unserer Bibliothek bilden. Es ist denkbar, dass sich ein Benutzer auf diese Ebene beschränkt, da zum Beispiel die Lösungen aus der folgenden Applikationsebene für die Anwendung des Benutzers nicht ausreichend sind.

Die Applikationsebene

Die Applikationsebene fügt der Bibliothek Klassen hinzu, mit deren Hilfe sich Aufgaben wie zum Beispiel das Verwalten von bewegten Spieleobjekten, das Anzeigen einer graphischen Benutzeroberfläche oder auch das Abspielen eines Videos einfach realisieren lassen.

2.2 Realisierung der Grafik-API Abstraktionsebene

2.2.1 Das Plugin

Das Grafikplugin (Client), das eine Implementierung der in 1.3.4 angesprochenen Objekte beinhaltet, sollte dynamisch austauschbar sein.
Damit ist gemeint, dass die Anwendung (bzw. der Benutzer) beim Starten der Anwendung festlegen kann, mit welcher Grafik-API sie operieren möchte.
Hierbei bieten sich dynamisch geladene und gelinkte Bibliotheken, „DLLs“ an (siehe A.1):

- Die gesamte Abstraktionsebene kann einfach ausgetauscht werden, ohne die gesamte Anwendung neu kompilieren zu müssen
- Die Anwendung kann zur Laufzeit die gewünschte Implementierung laden

In der Datei „AdClasses.pas“ befinden sich die Schnittstellendeklarationen für das Grafikplugin. Jede Andorra 2D Grafiksystem-DLL exportiert eine Funktion, die das Zeichenflächenobjekt „TAd2dApplication“ erstellt und einen Pointer darauf zurückgibt.

Auszug aus der Datei AndorraDX93D.dpr (gekürzt)

```
//Funktion, die einen Pointer auf eine Instanz der Klasse
//TAd2dApplication (oder abgeleitete) zurückgibt.
function CreateApplication:TAd2DApplication;stdcall;
begin
  result := TDXApplication.Create;
end;
```

```
exports
  CreateApplication;
```

Im Nachfolgenden besprechen wir die in der DLL zu implementierenden Klassen näher.

TAd2dApplication

Die oben angesprochene Grafik-API Abstraktionsebene wird mithilfe von abstrakten Klassen realisiert.
Den Dreh- und Angelpunkt bildet hierbei die Klasse „TAd2dApplication“, welche das „Zeichenflächenobjekt“ repräsentiert. Zudem besitzt sie Funktionen mit deren Hilfe die anderen in 1.3.4 beschriebenen Objekte erstellt werden können.

Auszug aus der Datei AdClasses.pas (gekürzt)

```
TAd2DApplication = class
  public
    //Erzeugt eine neue Textur und gibt den Pointer darauf zurück
    function CreateBitmapTexture: TAd2DBitmapTexture;virtual;abstract;
    //Erzeugt ein neues Mesh und gibt den Pointer darauf zurück
    function CreateMesh: TAd2DMesh;virtual;abstract;
    //Erzeugt ein neues Licht und gibt den Pointer darauf zurück
    function CreateLight: TAd2dLight;virtual;abstract;

    //Initialisiert die Schnittstelle
    function Initialize(
      AWnd:TAdWindowFramework):boolean;virtual;abstract;
    //Finalisiert die Schnittstelle
    procedure Finalize;virtual;abstract;

    //Löscht den Inhalt der Zeichenfläche
    procedure ClearSurface(
      ARect: TAdRect; ALayers: TAd2dSurfaceLayers;
      AColor: TAndorraColor; AZValue: double;
      AStencilValue: integer); virtual; abstract;

    //Beginnt das Zeichnen auf die Zeichenfläche
    procedure BeginScene;virtual;abstract;
    //Beendet das Zeichenen auf die Zeichenfläche
    procedure EndScene;virtual;abstract;
    //Bringt das Gezeichnete auf den Bildschirm (siehe
    //"Double Buffering")
    procedure Flip;virtual;abstract;
end;
```

TAd2dMesh

Ein Mesh repräsentiert ein in 1.3 angesprochenes Modell. Dazu beinhaltet es eine Eigenschaft „Vertexarray“, der die Vertexdaten zugeordnet werden können. Zudem können jedem „Mesh“-Objekt die Textur, „Indexarray“, Materialeigenschaften (für die Lichtbe-

rechnung) und Transformationsmatrizen zugewiesen werden. Mit der Methode „Draw“ kann das Mesh gezeichnet werden.

Auszug aus der Datei AdClasses.pas (gekürzt)

```
TAd2DMesh = class
  public
    //Lädt die angegebenen Daten (Vertex-/Indexbuffer)in die
    //Grafikkarte
    procedure Update;virtual;abstract;
    //Zeichnet das Meshobjekt im angegeben "Blendmodus" (bestimmt die
    //Farbmischung) und den "Primitivenmodus"
    procedure Draw(ABlendMode:TAd2DBlendMode;
      ADrawMode:TAd2DDrawMode);virtual;abstract;
    //Setzt die Materialeigenschaften. Ist PAd2dMaterial nil, so
    //wird kein Material verwendet.
    procedure SetMaterial(AMaterial: PAd2dMaterial);virtual;abstract;
    //Gibt zurück ob Daten vorhanden sind und an die Grafikkarte
    //geschickt wurden.
    property Loaded:boolean read GetLoaded;
    //Das Vertexarray, das an die Grafikkarte gesendet werden soll.
    property Vertices:TAdVertexArray read FVertices
      write SetVertices;
    //Das Indexarray, das die Indexierung der Vertices vorgibt.
    property Indices:TAdIndexArray read FIndices write SetIndices;
    //Bestimmt, wie viele Primitiven (Dreiecke, Linien, Punkte)
    //gezeichnet werden sollen.
    property PrimitiveCount:integer read FPrimitiveCount
      write FPrimitiveCount;
    //Bestimmt die Textur, mit der das Mesh texturiert werden soll.
    //Ist "Texture" nil, so wird das Mesh nicht texturiert.
    property Texture:TAd2DTexture read FTexture write SetTexture;
    //Die Transformationsmatrix des Modells (bestimmt die Position
    //im Raum)
    property Matrix: TAdMatrix read FMatrix write FMatrix;
    //Die Texturmatrix: Über diese Matrix können die Texturkoordinaten
    //transformiert werden, ohne die Vertices neu in die Grafikkarte
    //laden zu müssen.
    property TextureMatrix: TAdMatrix read FTextureMatrix
      write FTextureMatrix;
end;
```

TAd2dTexture

Es gibt zwei verschiedene Arten von Texturen:

- **Bitmap-Textur**: Texturtyp, dessen Daten aus dem Hauptspeicher des Computers in die Grafikkarte geladen wurden („TAd2dBitmapTexture“). Dieser Texturtyp ist am häufigsten anzutreffen.
- **Rendertarget-Textur**: Texturen dieses Typs werden direkt von der Grafikkarte

beschrieben: Anstatt die darzustellenden Grafiken auf dem Bildschirm (im Framebuffer) auszugeben, können sie auch in ebendiesem Texturtyp abgelegt werden.

Um möglichst flexibel zu bleiben und einem Mesh-Objekt beide Texturtypen zuweisen zu können, sind sie von TAd2dTexture abgeleitet.

Auszug aus der Datei AdClasses.pas (gekürzt)

```
TAd2DTexture = class
  public
    {Die Breite und Höhe des Texturobjektes im Speicher - kann von der
     Texturbreite des eigentlichen Bildes abweichen, da die meisten
     Grafikkarten nur "power of two", also Texturen mit Seitenlängen
     von 2 hoch n unterstützen.}
    property Width:integer read FWidth;
    property Height:integer read FHeight;
    {Diese Werte repräsentieren die Originalgröße der Textur.}
    property BaseWidth:integer read FBaseWidth;
    property BaseHeight:integer read FBaseHeight;
    {Bittiefe des Bildes. 16 und 32 Bit pro Pixel (ARGB) werden
     unterstützt.}
    property BitDepth:TAdBitDepth read FBitDepth;
    {Gibt zurück, ob die Textur geladen ist.}
    property Loaded:boolean read GetLoaded;
    {Pointer auf die interne Darstellung des Texturobjektes im
     jeweiligen Grafiksystem.}
    property Texture: Pointer read FTexture;
end;

TAd2DBitmapTexture = class(TAd2DTexture)
  public
    {Löscht den für die Textur reservierten Texturspeicher.}
    procedure FlushTexture;virtual;abstract;
    {Lädt die Textur aus einem Bitmap.}
    procedure LoadFromBitmap(ABmp:TAd2dCustomBitmap;
      ABitDepth: TAdBitDepth);virtual;abstract;
    {Speichert die Textur wieder in ein Bitmap.}
    procedure SaveToBitmap(ABmp:TAd2dBitmap);virtual;abstract;
end;

TAd2DRenderTargetTexture = class(TAd2dTexture)
  public
    {Setzt die Größe und Bittiefe der Textur und reserviert in der
     Grafikkarte Speicher dafür.}
    procedure SetSize(AWidth, AHeight: integer;
      ABitDepth: TAdBitDepth);virtual;abstract;
    {Löscht den für die Textur reservierten Texturspeicher.}
    procedure FlushMemory;virtual;abstract;
    {Speichert das aktuell im Grafikkartenspeicher abgelegte Bild in
     ein Bitmap.}
    procedure SaveToBitmap(ABmp:TAd2dBitmap);virtual;abstract;
end;
```

2.2.2 Objekte auf der Hostseite

Der das Grafikplugin ladenden Anwendung (Host) stehen ebenfalls einige Objekte bereit um das Plugin zu laden und Daten damit auszutauschen. Zunächst wäre die Klasse „TAdDllLoader“, die sich um das Laden der DLL und überprüfen der Kompatibilität kümmert, zu nennen. Mit „TAdDllExplorer“ kann nach kompatiblen Grafikplugins in einem beliebigen Verzeichnis gesucht werden. „TAd2dBitmap“ (bzw. deren Basisklasse „TAd2dCustomBitmap“) ist eine Schnittstellenklasse um Bitmapdaten zwischen Host und DLL auszutauschen.

2.2.3 Windowframeworks

Um die Zeichenfläche von der VCL (Visual Component Library; Delphis, an Windows gebundene, Komponentenbibliothek) unabhängig zu machen, existiern so genannte „Windowframeworks“. Diese sind dafür zuständig, dem Grafikplugin das Steuerelement zu übergeben, auf dem die Zeichenfläche (hier auch „Kontext“) erstellt werden soll. Bestimmte Windowframeworks sind außerdem in der Lage selbst einen 3D-Kontext für das jeweilige Grafiksystem zu erstellen und diesen zu übergeben. Vor allem bei der portieren von Anwendungen nach Linux/MacOSX ist diese Methode sehr praktisch:

- **AdVCLWindow** integriert die Zeichenfläche in eine bestehende VCL-Anwendung. Dieses Windowframework wird normalerweise automatisch verwendet. Da die eingebundene VCL recht viel Speicherplatz in der ausführbaren Datei benötigt, kann dieses Verhalten per Kompilierschalter abgestellt werden.
- **AdLCLOGLComponentWindow**: Unter Linux ist es dem OpenGL-Plugin nicht ohne weiteres möglich einen 3D-Kontext zu erstellen (aufgrund der Vielzahl der Fenstersysteme). Dieses Windowframework nimmt dem Plugin diese Arbeit ab und erstellt auf der angegebenen Komponente einen Kontext.
- **AdLCLOGLWindow** erstellt ein eigenes Fenster, das einen OpenGL-Kontext beinhaltet.
- **AdSDLWindow** erstellt mithilfe der Bibliothek „SDL“ ein eigenes Fenster, das einen OpenGL-Kontext beinhaltet
- **AdGLFWWindow** erstellt mithilfe der Bibliothek „GLFW“ ein eigenes Fenster, das einen OpenGL-Kontext beinhaltet
- **AdWin32Window** erstellt mithilfe der nativen Microsoft Windows API ein eigenes Fenster und überlässt dem Plugin das Erstellen des Kontexts.

Das Windowframeworksystem verwendet das flexible System zur Registrierung von Klassen. Siehe dazu auch Kapitel A.2.

2.2.4 Fazit

Ein einfaches Beispielprogramm, das die Ansteuerung dieser untersten Abstraktionsebene demonstriert, findet sich in B.1. Mit diesem Beispiel wird auch klar, dass es relativ kompliziert ist, nur dise Ebene zu verwenden. Es sollte niemanden zugemutet werden, für das Anzeigen eines einfachen Bildes ein Mesh- und Texturobjekt erzeugen zu müssen, sowie alle Koordinaten selbst zu setzen. Auch die Transformationen über die Matrizen sind recht kompliziert. Deshalb folgt im nächsten Kapitel die 2D-Funktionalitätsebene, die einfache 2D-Zeichenfunktionen bereitstellt. Auf die Möglichkeiten der untersten Abstraktionsebene kann dann natürlich nach wie vor zugegriffen werden.

2.3 Die 2D-Funktionalitätsebene

Mit den im vorherigen Kapitel vorgestellten Objekten lassen sich bereits ohne weiteres die komplexesten Spiele programmieren. Wie jedoch beschrieben wurde, sind diese Objekte relativ kompliziert zu verwenden - besonders dann, wenn man „nur“ 2D-Grafiken ausgeben möchte. Folgende Dinge sind nun zu vereinfachen:

- **Initialisierung:** Die Initialisierung und das Bereitstellen der Zeichenfläche sollte mit nur einem einzelnen Objekt möglich sein. Außerdem sollte ohne Umwege die Möglichkeit bestehen, die Zeichenfläche an eine VCL-Applikation zu binden. Die dafür benötigten Windowframeworks sollten also automatisch eingebunden werden. Diese Funktionalität wird in dem Objekt „TAdDraw“ zusammengefasst.
- **Zeichnen von Bildern:** Auch das Zeichnen von Bildern sollte über ein einziges Objekt möglich sein. Mit einfachen einfachen Befehl sollten die Bilder geladen und gezeichnet werden können. Darum wird sich die Klasse „TAdImage“ kümmern.
- **„Canvas“-Objekt, Textausgabe:** Es sollte über ein einfaches „Canvas“-Objekt, wie dies auch in der VCL existiert, die Möglichkeit gegeben werden einfach geometrische Formen zu zeichnen. Zudem sollte man in der Lage sein, Texte auszugeben. Für diese Funktionalität sind die Klassen „TAdCanvas“ und „TAdFont“ zuständig.
- **Zeichenflächen:** Es sollte die Möglichkeit gegeben sein, auf verschiedene Zeichenflächen zeichnen zu können. Dabei handelt es sich um eine Kapselung der oben beschriebenen Rendertarget-Texturen. Auch „TAdDraw“ sollte eine solche Zeichenfläche sein. Die zugehörige abstrakte Oberklasse ist „TAdSurface“.

Für einen Überblick über die Funktionalität der 2D-Funktionalitätsebene siehe das Programmbeispiel in Anhang B.2 sowie das zugehörige Klassendiagramm am Ende des Dokuments.

2.3.1 Initialisierung und Zeichenflächen

„TAdDraw“ kapselt das oben beschriebene TAd2dApplication Objekt, verwaltet das geladene Plugin, kümmert sich um das Auswählen eines kompatiblen Windowframeworks

und initialisiert die Applikation. Zugleich handelt es sich um die Hauptzeichenfläche. Alle Grafikausgaben werden standardmäßig hier durchgeführt.
Zudem stellt TAdDraw so genannte „Surfaceevents“ bereit: Wird die Zeichenoberfläche finalisiert und anschließend wieder initialisiert (zum Beispiel um die Auflösung zu wechseln), so werden alle Objekte der 2D-Funktionalitätsebene darüber benachrichtigt. Da während dem Finalisierungsvorgang alle Objekte aus dem Grafikplugin ihre Gültigkeit verlieren, haben die neuen Objekte die Chance alle Daten im Arbeitsspeicher zwischenzuspeichern und beim Initialisieren wieder in die Grafikkarte zu laden. Hierzu besitzen die meisten Objekte aus der 2D-Funktionalitätsebene eine Funktion „Notify“, welche mit „TAdDraw“ verknüpft wird, sowie eine „Initialize“ und „Finalize“ Funktion.
Um mehrere Zeichenflächen gleichberechtigt nebeneinander nutzen zu können, existiert die abstrakte Oberklasse „TAdSurface“: Sie bildet die Basisklasse für Zeichenoberflächen und stellt die abstrakten Funktionen „Activate“ und „Deactivate“ bereit. Die abgeleitete Klasse „TAdRenderingSurface“ fügen Informationen über die Ausmaße der Oberfläche, Szenenmanagement („TAdScene“) und ein „Canvas-Objekt“ („TAdCanvas“), wie es aus der VCL bekannt ist, hinzu. Von „TAdRenderingSurface“ wiederum sind das zentrale Objekt „TAdDraw“, sowie „TAdTextureSurface“ abgeleitet. „TAdTextureSurface“ verwaltet ein „TAdCustomImage“ und ein „TAdRenderTargetTexture“ Objekt.

2.3.2 Bildausgabe

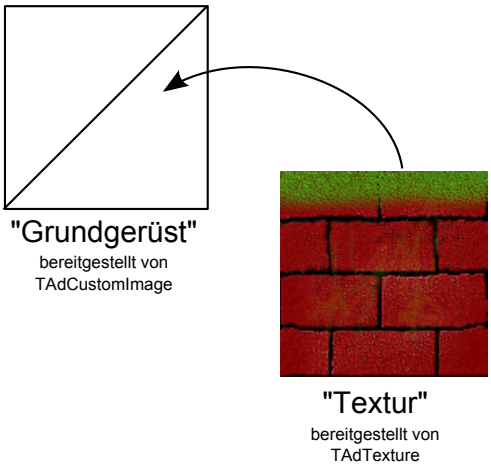


Abbildung 2.1: Verhältnis zwischen TAdCustomImage und TAdTexture: TAdCustomImage stellt nur das Grundgerüst (die Vertices) bereit, auf die die Textur gelegt wird.

Andorra 2D bietet einfache Möglichkeiten, ein einzelnes Bild zu zeichnen. Dafür ist in der Praxis die Klasse „TAdImage“ verantwortlich. Intern ist „TAdImage“ von „TAdCustomImage“ abgeleitet. Diese Klasse kümmert sich um das Erstellen eines des Meshs und Generieren der Vertexdaten und Setzen von Transformationsmatrizen. Wird dem Objekt eine Textur zugewiesen, so muss das interne Mesh durch Aufrufen der Methode „Restore“ an die Textur angepasst werden. Siehe auch Abbildung 2.1
„TAdCustomImage“ stellt zudem bereits eine Reihe von Zeichenmethoden bereit. So

können die Bilder gestreckt/gestaucht, gedreht, durchsichtig oder nur Teile aus der Textur gezeichnet werden. Außerdem ist „TAdCustomImage“ in der Lage Bilder zu animieren: Dazu muss die Textur (wie bei einem Filmstreifen) aus mehreren zusammengesetzten Einzelbildern bestehen. Bei jeder Zeichenoperation kann die Bildnummer („Pattern“) angegeben werden, die gezeichnet werden soll. Die Ausgabe eines Bereiches der Textur erfolgt intern durch das Anpassen der Texturtransformationsmatrix. Folgende Zeichenausgaben sind möglich:

Verschiedene Zeichfunktionen von TAdCustomImage

```
//Zeichnet das Bild auf der Oberfläche "Dest" an den Koordinaten
//"X", "Y" mit dem Teilbild "PatternIndex"
procedure Draw(Dest: TAdSurface; X, Y: Single; PatternIndex: integer);
//Streckt das Bild auf das angegebene Rechteck.
//Auch vorhanden: DrawAlpha, DrawAdd und DrawMask, die das Bild
//mit dem entsprechenden Zeichenmodus zeichnen
procedure StretchDraw(Dest:TAdSurface; const DestRect:TAdRectEx;
  PatternIndex:integer);
//Zeichnet das Bild mit der angegebenen Höhe und Breite an die
//Position X, Y mit dem relativen Rotationszentrum CenterX, CenterY
//im Winkel "Angle" rotiert.
//==>DrawRotateAlpha, DrawRotateAdd, DrawRotateMask
procedure DrawRotate(Dest: TAdSurface; X, Y, Width, Height: Single;
  PatternIndex: Integer; CenterX, CenterY, Angle: Single);
//"DrawEx" erlaubt es alle möglichen Zeichenmodi miteinander zu
//kombinieren: "Sourcerect" gibt einen Bereich aus der Textur an,
//der gezeichnet werden soll, "DestRect" das Rechteck, auf dem
//das Bild ausgegeben werden soll. Über "BlendMode" kann der
//Zeichenmodus angegeben werden.
procedure DrawEx(Dest: TAdSurface; SourceRect, DestRect: TAdRectEx;
  CenterX, CenterY, Angle: Single; Alpha: integer;
  BlendMode: TAd2dBlendMode);
```

Der wohl größte Unterschied zu „TAdImage“ ist, dass „TAdCustomImage“ nur ein abstraktes Texturobjekt des Typs „TAdCustomTexture“ beinhaltet. Ähnlich wie bei „TAd2dTexture“ kann es sich dabei sowohl um eine Bitmaptextur und um eine Rendertargettextur handeln. Die abgeleiteten Klassen sind hierbei „TAdTexture“ und „TAdRenderTargetTexture“. „TAdTexture“ bietet zudem Möglichkeit die Textur aus einer Datei zu laden oder die Bilddaten komprimiert in eine Textur zu speichern. Auch „TAdImage“ fügt diese Funktionalitäten für den Dateizugriff ein. Zudem können mehrere „TAdImage“-Objekte in der Liste „TAdImageList“ verwaltet werden.

2.3.3 Laden von Bildern aus verschiedenen Grafikformaten

Bisher konnten Bilder nur direkt aus dem Speicher an das Grafikplugin übergeben werden („TAdCustomBitmap“, „TAd2dBitmap“). Nun sollen die Bilder auch aus verschiedensten Grafikformaten (wie BMP, JPG, PNG etc.) geladen werden. Hierzu stellt Andorra 2D ein modulares System zur Verfügung, durch das einzig die Units, die entsprechende Gra-

fikformatklassen (abgeleitet von TAdGraphicFormat) beinhalten in eine „Uses-Klausel“ der Applikation aufgenommen werden. Näheres ist in Anhang A.2 beschrieben.
Außerdem implementieren die Grafikformatklassen Funktionen um andere Grafikobjekte (zum Beispiel TGraphic der VCL oder TPNGImage der Bibliothek „TPNGImage“) in ein TAdBitmap zu konvertieren. Folgende Formate und Kompressoren werden derzeit unterstützt:

Unit	Funktion
AdBMP.pas	Verwendet einen eigenen Lader um Windows-BMP-Bilder zu laden.
AdDevIL.pas	Verwendet die Bibliothek „DevIL“ um verschiedenste Bildformate zu laden.
AdFreeImage.pas	Verwendet die Bibliothek „FreeImage“ um verschiedenste Bildformate zu laden.
AdPNG.pas	Verwendet einen von Manuel Eberl entwickelten Lader um PNG-Bilder zu laden.
AdTGA.pas	Verwendet einen von Manuel Eberl entwickelten Lader um TGA-Bilder zu laden.
AdVCLFormats.pas	Fügt die Funktionalität hinzu alle von der VCL unterstützten Grafikformate auch mit Andorra 2D laden zu können. Integriert außerdem alle von Andorra 2D unterstützten Grafikformate im Gegenzug in die VCL.

2.3.4 Ausgabe einfacher geometrischer Objekte

„TAdCanvas“ kümmert sich um das schnelle Ausgeben von einfachen, geometrischen Objekten auf einer Zeichenoberfläche. Um so wenig Daten wie möglich an die Grafikkarte schicken zu müssen, puffert das Objekte alle Grafikausgaben in so genannten „Displaylisten“. Nur wenn sich an den gezeichneten Objekten wirklich etwas verändert, werden diese erneut an die Grafikkarte gesendet.
Wichtig ist hierbei, dass der Benutzer der Bibliothek die Grafikausgaben „gruppiert“: Ist eine logische Gruppe von Grafikbefehlen abgeschlossen (zum Beispiel das Zeichnen eines aus mehreren Zeichenoperationen bestehenden Objektes), so sollte der „TAdCanvas.Release“ Befehl aufgerufen werden. „TAdCanvas“ fasst nun die bisher gezeichneten Objekte in einer Displayliste zusammen. Anschließend werden die Objekte gezeichnet. Alle Listen eines kompletten Bildschirmbildes (Frame) werden im Speicher gehalten und im nächsten Frame mit den vorherigen Grafikausgaben abgeglichen.

2.3.5 Textausgabe

Ein wichtige Funktion ist die Ausgabe von Text. In Andorra 2D wird auch dieses Pro-

blem äußerst flexibel gelöst: So wird über einen „Fontgenerator“ (von „TAdFontGenerator“ abgeleitete Klasse) eine Textur erstellt, die alle ASCII-Zeichen der gewählten Schriftart beinhaltet (siehe Abbildung 2.2). Je nach dem wie „TAdFontGenerator“ implementiert ist, kann die Textur auf verschiedene Arten und mit verschiedensten Effekten (Schatten, schräge Kanten, etc.) erstellt werden. Die Standardimplementierung „TAdVCLFontGenerator“ verwendet die einfachen Textausgabemöglichkeiten der VCL (TBitmap.Canvas). Soll nun ein Text ausgegeben werden, so kommt die Klasse „TAdFont“ ins Spiel: Diese schickt den Auszugebenden Text und die vom Fontgenerator erhaltenen Buchstabengrößen zunächst an einen von der abstrakten „TAdTypeSetter“-Klasse abgeleiteten Textsetzer: Dieser setzt die vorhandenen Buchstaben nach den vorgegebenen Parametern (Ausrichtung, Zeilenabstand, Umbruchverhalten) und gibt den gesetzten Text an das „TAdFont“ Objekt zurück: Hier wird ein Meshobjekt erstellt und dieses mit der oben erzeugten Textur verknüpft. Zum Erzeugen von Instanzen der Klasse „TAdFont“ existiert die Klasse „TAdFontFactory“, die für die von uns angegebenen Daten einen passenden Fontgenerator sucht.

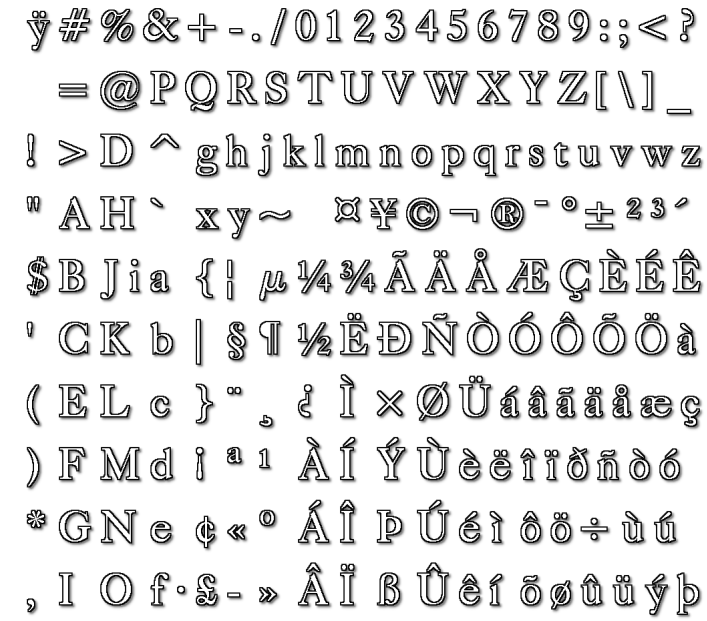


Abbildung 2.2: Eine Fonttextur, wie sie von Andorra 2D verwendet werden kann.

3 Die Applikationsebene

Die Applikationsebene fügt aufbauend auf der 2D-Funktionalitätsebene weitere Funktionen hinzu, die jedoch recht Anwendungsspezifisch sind und teilweise schon fertige Komplettlösungen darstellen. Exemplarisch sind im Folgenden einige Teile der Applikationsebene aufgegriffen und näher beschrieben.

3.1 Die „Sprite Engine“

3.1.1 Sprites

Bei „Sprites“ handelt es sich im ursprünglichen Sinne um Rastergrafiken, die von der Grafikhardware eines Computersystems unabhängig voneinander auf dem Bildschirm dargestellt werden können, ohne den eigentlichen Framebufferinhalt zu verändern, wodurch hohe Datenübertragungsrate zwischen Haupt- und Grafikprozessor vermieden wird. Diese Technik machte es erst möglich grafisch anspruchsvolle 2D-Echtzeitanwendungen wie Spiele oder grafische Benutzeroberflächen zu erstellen.
Aufgrund der hohen Leistung moderner Computersysteme und schneller Speicheranbindung findet diese Technik kaum noch — meistens nur für den Mauszeiger — Anwendung. Vielmehr werden heute einzelne grafische (bewegte) Objekte in einem 2D-Computerspiel als „Sprites“ bezeichnet.
Andorra 2D stellt einige Klassen zur Verfügung um diese Objekte auf einfache Weise verwalten zu können. Dabei kümmert sich die Bibliothek um das Zeichnen, Bewegen und Kollisionskontrolle der Sprites. Mit der Sprite-Abstraktion können viele verschiedene Spieletypen umgesetzt werden.

3.1.2 Klassenaufbau

Die Basisklasse des „Sprite-Systems“ bildet die Klasse TSprite. Diese bietet überschreibbare Methoden an, die später dazu verwenden werden können das Sprite zu zeichnen, zu bewegen und auf Kollisionen mit anderen Sprites zu reagieren. Zudem beinhaltet diese Spriteklasse Informationen über Welt- und Bildschirmposition und Ausmaße des Objektes.
Eine Besonderheit der Spriteengine ist die Tatsache, dass jedes Spriteobjekt Kinderobjekte besitzen kann. Diese werden mit dem Elternobjekt gezeichnet und verschoben. Dadurch wird erreicht, dass ein Spriteobjekt im Prinzip zur Verwaltung seinesgleichen eingesetzt werden kann.

Diese Eigenschaft machen wir uns im Folgenden zu nutze: Die von „TSprite“ abgeleitete Klasse „TSpriteEngine“ ist für die Verwaltung der Sprites einer Szene zuständig.

3.1.3 Kollisionsoptimierung

Soll überprüft werden, ob ein Sprite A mit einem anderen auf dem Spielfeld kollidiert (wie dies in Spielen oftmals der Fall ist) so ist die einfachste und naive Lösung alle vorhanden Sprites auf ein Überschneiden mit Sprite A zu überprüfen. Nehmen wir nun an, dass alle anderen Sprites ebenfalls eine solche Überprüfung durchführen, so steigt der Aufwand quadratisch mit der Anzahl der Sprites: Bei tausend, über das Spielfeld verteilten Objekten sind also eine Millionen Überprüfungen notwendig.
Eine mögliche Vereinfachung besteht in folgender Überlegung: Ein Spielfeldobjekt kann nur mit seinen benachbarten Objekten kollidieren. Um dies zu ermöglichen wird das Spielfeld in virtuelle Felder aufgeteilt, die eine Liste der in ihnen befindlichen Sprites besitzen. Die Kollisionskontrolle muss nun nur noch innerhalb der Felder in denen sich ein Sprite befindet stattfinden. Bei tausend relativ gleichmäßig verteilten Sprites, einer Spielfeldgröße von 1024 mal 1024 Pixeln und einer Feldgröße von 256 Pixeln, unter der Annahme, dass die meisten Objekte nur ein Feld gleichzeitig besetzten sind so durchschnittlich nur ca. 3907 Überprüfungen von Nöten:

$$\begin{aligned}\left(\frac{1024}{256}\right)^2 &= 16\\ \frac{1000}{16} &= 62,5\\ 62,5^2 &= 3906,25\end{aligned}$$

In Andorra 2D ist die Klasse „TAd2DSpriteList“ für die (schnelle) Organisation der Objekte in Zeilen und Spalten sowie dynamische Vergrößerung und Verkleinerung des Spielfeldes und die damit verbundene Speicheroptimierung verantwortlich.

3.2 Wiedergabe von Videodateien

Oftmals werden in Spielen Videodateien, die zum Beispiel als Zwischensequenzen den Handlungsverlauf darstellen, wiedergegeben. Daher ist es praktisch, wenn die verwendete 2D-Grafikbibliothek Klassen zur Videowiedergabe bereitstellt. Wichtig ist hierbei, dass sich Andorra 2D zwar um das Bereitstellen der mit der Videodatei verknüpften Audiodaten kümmern kann, diese jedoch nicht wiedergibt, da diese Aufgabe in einem zweitem Projekt, einer Audiobibliothek, münden würde. Der Programmierer der Anwendung muss sich also selbst darum kümmern die Audiodaten zum Lautsprecher zu bringen - daher werden wir im Folgenden nicht näher darauf eingehen.

Abbildung 3.1: Andorra 2D bei der Wiedergabe eines Videos

3.2.1 Problemstellung

Es mag zunächst trivial klingen Videos wiedergeben zu wollen, doch dahinter verbergen sich echte technische Probleme. Wie wir wissen, bestehen Videos im Grunde genommen aus einer Reihe von Bildern, die in regelmäßigem Intervall gewechselt werden. In unserem Fall (bei einem Video im PAL-Format) sind das 25 Bilder pro Sekunde, bei 768 x 576 Pixeln à 3 Bytes pro Pixel. Das macht pro Sekunde eine Datenmenge von

$$25\frac{Bilder}{Sekunde} \cdot 768 \cdot 576\frac{Pixel}{Bild} \cdot 3\frac{Byte}{Pixel} = 31,64\frac{MByte}{Sekunde}$$

Um Speicherplatz zu sparen, werden Videos oftmals verlustbehaftet komprimiert abgelegt. Dabei werden meistens nach einem physiologischen Modell Bildanteile, die von Menschen nicht wahrgenommen werden, herausgefiltert und nur die veränderten Bildstellen gespeichert [11]. Eine der ältesten und am weitesten verbreiteten Standards ist hierbei MPEG. Der auf DVDs verwendete MPEG II Standard ist in der Lage Videos bei einer Datenrate von ca. $1,2\frac{MByte}{Sekunde}$ in hoher Qualität zu speichern [9].
Leider basieren die verlustbehafteten Videoformate auf komplizierten mathematischen Modellen. Hinzu kommt, dass die meisten Videos nicht standardkonform sind. Einen Dekoder für (MPEG-)Videoformate zu schreiben, ist also eine längere Angelegenheit.
Zum Glück existieren bereits quelloffene Bibliotheken wie zum Beispiel „Acinerella“[1], die das Dekodieren der Videos stark vereinfachen.

[1]Siehe `http://acinerella.sourceforge.net/`, eine FFMpeg (`http://ffmpeg.org/`) Wrapper-Bibliothek zur Wiedergabe verschiedenster Mediendateien

Dennoch bleibt das Problem, die dekodierten Daten effizient vom Prozessor zur Grafikkarte zu senden. Es ist keine Lösung, die Bilder zu dekodieren und erst wenn sie benötigt werden direkt an die Grafikkarte zu schicken. Da das dekodieren der Videos relativ viel Rechenaufwand ist und die komprimierten Daten ohne Unterbrechung an unsere Anwendung übertragen werden müssen, würde jede andersweitige Auslastung des Systems zu einer ruckelnden Videowiedergabe führen. Eine weiter Hürde bildet die Tatsache, dass Bild und eventuelle Toninformationen nach einiger Zeit beginnen „auseinander zu laufen“: Dieser Effekt wird meistens durch kurzzeitig hohe Systemauslastung oder Plattenzugriffe ausgelöst. Audio und Video müssen also stets synchronisiert werden.

3.2.2 Lösung des Problems in Andorra 2D

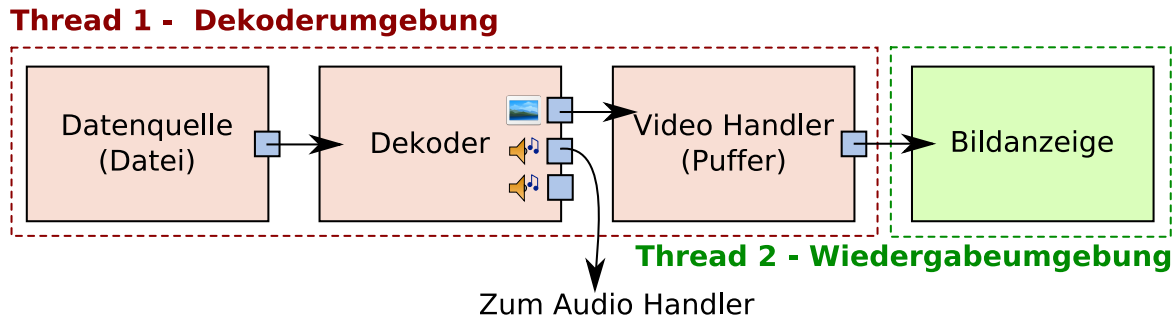


Abbildung 3.2: Der Weg der Videodaten im Andorra 2D Videomodul. Die Ausgänge am Dekoderblock stellen die verschiedenen Spuren der Mediendatei dar.

Der Aufbau des Videomoduls in Andorra 2D ist äußerst modular. Die Hauptklasse, die sich um die Organisation der Videowiedergabe kümmert, ist hierbei „TAdCustomVideoTexture“. Wie der Name schon sagt, repräsentiert diese Klasse eine Textur, auf der ein Video abläuft. Die „Videotextur“ kann demnach jedem beliebigen Andorra 2D Objekt zugewiesen werden - sie kann somit auch im Canvas verwendet oder auf 3D-Oberflächen gelegt werden.
„TAdCustomVideoTexture“ verwaltet ein Objekt der Klasse „TAdMediaDecoderThread“. Diese ließt über eine Callbackfunktion Daten aus einer beliebigen Datenquelle ein und schickt diese an eine registrierte Mediendekoderklasse (hierbei wird das in A.2 beschriebene Verfahren angewandt). Die Mediendekoderklasse gibt Informationen über die vorhandenen Teilströme der Mediendatei zurück: Eine Videodatei auf einer DVD zum Beispiel enthält oftmals mehrere Ton- und Videospuren. Mit den detektierten Teilströmen kann nun ein so genannter „Media Handler“ verknüft werden. Dieser kümmert sich um das Zwischenspeichern (Puffern) der dekodierten Daten und gibt zurück, ob dieser Puffer vollständig gefüllt ist. „TAdMediaDecoderThread“ versucht währenddessen alle Speicher immer gefüllt zu halten.
Die Klasse „TAdCustomVideoTexture“ registriert automatisch einen „TAdVideoHandler“ mit dem ersten Videoteilstrom und ist in der Lage das letzte Bild von dessen Puffer zu nehmen und in die Textur zu schreiben. Wichtig ist hierbei, dass die Zugriffe auf

den Puffer synchronisiert werden: Der Videohandler agiert im Kontext des Mediendekoderthreads, „TAdCustomVideoTexture“ im Kontext des Hauptthreads[2].
Die von „TAdCustomVideoTexture“ abgeleitete Klasse „TAdVideoTexture“ fügt zeitgesteuertes Wechseln der Bilder (= Wiedergabe des Videos) hinzu. „TAdCustomVideo“ erlaubt es, das Video nicht nur auf eine Textur auszugeben, sondern Zeichnet das Video auch in einem beliebigen Rechteck auf die Zeichenfläche. „TAdVideo“ erweitert die Hauptklasse um Datenzugriffsmethoden und die Möglichkeit im Datenstrom zu springen.
Um Bild und eventuell wiedergegebenen Ton synchron halten zu können, wird mit jedem Videobild und jedem Audioblock ein so genannter „Timecode“ mitgeführt. Dieser gibt die aktuelle Position der jeweiligen Spuren im Video in Millisekunden an. Mithilfe dieser Daten kann eine Synchronisierung (zum Beispiel durch schnellere/langsamere Bildfolge) durchgeführt werden.

3.3 Die graphische Benutzeroberfläche

Abbildung 3.3: Die GUI zweier Spiele („Oblivion“ und „Fallout 3“) im Vergleich: Sie ist jeweils an das Szenario des Spieles angepasst. Standardkomponenten des Betriebssystems werden nicht verwendet.

Graphische Benutzeroberflächen (engl. „Graphical User Interface“, GUI) sind heutzutage in fast allen modernen Betriebssystemen vorhanden und stellen eine häufig benutzte Interaktionsmöglichkeit zwischen dem Benutzer und der Software dar. Auch in Spielen werden Eingaben oftmals mithilfe von graphischen Benutzeroberflächen ermöglicht. Um eine nahtlose Integration der GUI in das Spiel zu verwirklichen, werden meistens nicht die betriebssytemeigenen Steuerelemente/Komponenten (Schaltflächen, Fenster, Eingabefelder etc.) verwendet, sondern eigene, graphisch an das Spiel angepasste GUI-Bibliotheken. Um den Benutzern von Andorra 2D die Arbeit des selbst Entwickeln einer solchen Bibliothek abzunehmen, stellt Andorra 2D eine solche GUI-Bibliothek bereit,

[2]Die so genannten Threads entsprechen eigenen Prozessen innerhalb einer Anwendung, die gleichzeitig Ablaufen — alle Speicherbereiche auf die mehrere Threads (schreibend) zugreifen, müssen entsprechend geschützt werden.

die sich mit der Hilfe von so genannten „Skins“ (engl. „Haut“) an das Spiel anpassen lässt.

3.3.1 Allgemeiner Aufbau

Die GUI-Komponenten Bibliothek soll im Aufbau an die VCL (Visual Components Library) von Delphi angelehnt sein. Die GUI muss also folgende Eigenschaften besitzen:

- **Verschachtelung:** Bestimmte GUI-Komponenten (zum Beispiel Formulare (Fenster), Panele) dürfen Kinderkomponenten besitzen, die sich mit den Eltern mit verschieben.
- **Ereignisorientierung:** Die GUI sollte Ereignisse bereitstellen, welche der Programmierer mit seinem Programmcode verknüpfen kann.
- **Editierbarkeit:** Die GUI sollte sich wie im Formulareditor der Delphi IDE bearbeiten lassen. Die GUI muss sich in Dateien speichern und wieder daraus laden lassen.
- **Erweiterbarkeit:** Es sollten einfach neue Komponenten entwickelt und hinzugefügt werden können.

Die GUI ist intern ähnlich wie die Spriteengine aufgebaut. Die Basisklasse „TAdComponent“ führt das Basisverhalten aller Komponenten ein: Sie beinhaltet eine Liste der Kinderkomponenten, Größe und Ausmaße des Elements, kümmert sich um die korrekte Weiterleitung der Ereignisse an Kinderkomponenten, setzen des Fokus' und um das Verhalten im Editiermodus. Um eine neue Komponente in das System einzufügen, muss diese von TTAdComponentäbgeleitet werden und im „Initialization“- Abschnitt der jeweiligen Unit registriert werden. Die abgeleiteten Klassen müssen nur die Darstellung und das Verhalten implementieren.
Der mitgelieferte Editor erstellt für jede registrierte Komponente eine entsprechende Schaltfläche in der Komponentenpalette. Die veröffentlichten Eigenschaften der Komponenten werden über die Object Pascal interne RTTI (Run Time Type Information) ausgelesen und in einem Objektinspektor dargestellt (siehe auch A.2). Somit wären die oben aufgeführten Punkte verwirklicht.

3.3.2 Das Skinsystem

Wie oben beschrieben sollte sich die GUI eines Spieles an das Szenario desselben anpassen. Hierzu werden so genannte „Skins“ verwendet. In Andorra 2D werden „Skins“ folgendermaßen abstrahiert: Jedes Steuerelement besitzt eines oder mehrere „Skinitems“ (TAdSkinItem), die das Aussehen des Steuerelementes festlegen. Jedes „Skinitem“ besteht aus mehreren Regionen (TAdSkinElem), die Bildausschnitte eines Quellbildes beinhalten und das beliebige Skalieren des Steuerelements ermöglicht. Jedes „Skinitem“ kann mehrere Quellbilder beinhalten, die verschiedene Zustände des Steuerelementes anzeigen. So könnte das „Skinitem“ einer Schaltfläche zum Beispiel die Zustände „Normal“,

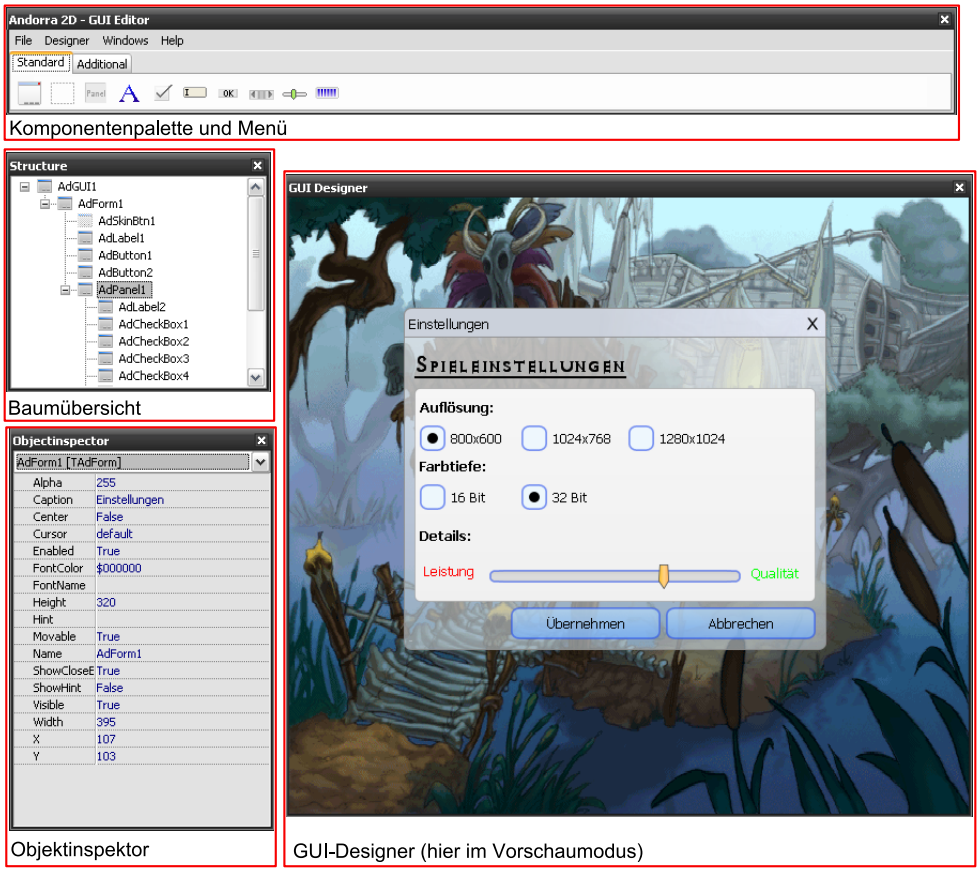


Abbildung 3.4: Der in Andorra 2D enthaltene GUI-Editor und seine verschiedenen Abschnitte.

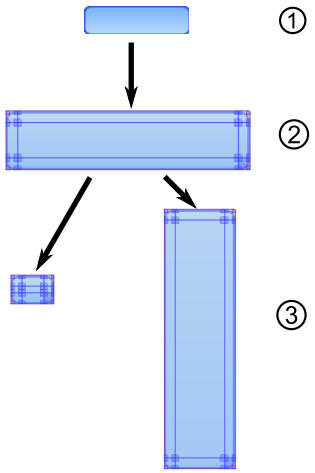


Abbildung 3.5: Funktionsweise des Skinsystems: Das Quellbild (1) wird in verschiedene Regionen (2) eingeteilt. Anschließend kann das „Skinitem“ weitestgehend ohne Qualitätseinbußen (d.h. ohne Pixelig zu werden) beliebig skaliert werden (3): Die Ecken des Elements bleiben immer in der gleichen Größe.

„Hover“ (mit der Maus darübergefahren), „Down“ (heruntergedrückt) und „Focused“ (das Element hat den Fokus) besitzen. Ein Skin „TAdSkin“ besteht in Andorra 2D aus einer Liste von „Skinitems“.
Die Komponenten fordern von „TAdSkin“ jeweils das/die zu ihnen passenden Skinitem(s) an und fordern „TAdSkin“ auf dieses mit einem bestimmten Zustand in einen Bereich auf den Bildschirm zu zeichnen.
Das GUI-System ist somit weitestgehend von Andorra 2D losgelöst: Nur über „TAdSkin“ und eventuelle manuelle Zeichenoperationen (Text, Linien) ist das System mit dem Andorra 2D Kern verbunden. Deshalb wurde das komplette GUI System von mehreren Personen schon in ihre eigenen Anwendungen eingebunden[3].

[3] siehe `http://www.delphigl.com/forum/viewtopic.php?p=65642#65642`

4 Fazit

Im vorausgegangenen Dokument wurde die Motivation hinter dem Andorra 2D Projekt und dessen grundlegender Aufbau und Funktionsweise grob skizziert. Viele Module mussten aufgrund des limitierten Platzes außer Acht gelassen werden, so zum Beispiel das Partikel- und Shadersystem.
Trotz des modularen Aufbaus der Bibliothek könnten einige Module flexibler gestaltet werden: So bietet zum Beispiel die „Sprite Engine“ keine Möglichkeit zum Einbinden von 3D-Modellen und zur perspektivischen Transformation der Sprites. Auch das Fontsystem ist nicht in der Lage Mehrbytezeichensätze (zum Beispiel UTF-8) anzuzeigen: Somit können zum Beispiel weder griechische, kyrillische, japanische, chinesische oder klingonische Schriftzeichen angezeigt werden. Weiterhin ist die Unterstützung für Mehrprozessorsysteme limitiert: Viele Prozesse der Bibliothek werden nur auf einem einzigen Prozessorkern ausgeführt. Ausnahmen bilden lediglich das (nicht besprochene) Partikelsystem und die Videowiedergabe.
Die Entwicklung der Bibliothek wird somit weitergehen und die aufgeführten Schwächen - früher oder später - beseitigt werden.
Ziel ist es, dass (nachdem ein gewisser Qualitätsstand erreicht ist) immer mehr Verbesserungen von der Entwicklergemeinschaft übernommen werden und zum Beispiel neue Grafikplugins hinzugefügt werden. Hierzu ist jedoch eine Ausweitung der Dokumentation von Nöten, die die Schnittstellen noch genauer beschreibt.

A Zusatzinformationen

A.1 Das Linken

Um Programmcode übersichtlich zu gestalten und manche Teile in späteren Projekten wiederverwerten zu können, werden Programme oftmals in mehrere Teile aufgeteilt. Auch zusätzliche Bibliotheken (in Pascal zum Beispiel die Unit SysUtils) sind somit vom eigentlichen Programm abgetrennt.

Statisches Linken

Übersetzt der Compiler ein Programm, so werden alle Programmteile zunächst unabhängig voneinander compiliert. Ein weiteres Programm, der so genannte Linker, fügt die Programmteile zu einer einzigen ausführbaren Datei (unter Windows „EXE“) zusammen.
Dies hat den Nachteil, dass die so entstandenen Programme mitunter sehr groß werden können. Wird eine verwendete Bibliothek nun auch noch von mehreren Programmen verwendet, so steht der gleiche Code mitunter mehrere Male auf der Festplatte oder im Arbeitsspeicher. Auch können einzelne Programmteile nicht unabhängig von anderen ausgetauscht werden.

Dynamisches Linken

Beim dynamischen Linken werden die Programmteile nach wie vor unabhängig voneinander compiliert. Jedoch werden sie nicht von einem Linker zusammengefügt.
Stattdessen wird aus der zusätzlichen Bibliothek eine dynamisch gelinkte Bibliothek (engl. dynamic linked library, auch DLL) erstellt. Die EXE enthält weiterhin noch nicht aufgelöste Funktionsnamen.
Verwenden mehrere Programme die Bibliothek, so befindet sich diese nur einmal auf der Festplatte und im Optimalfall nur einmal im Arbeitsspeicher. Außerdem kann eine DLL einfach ausgetauscht werden, ohne das ganze Programm neu erstellen zu müssen.
Bei der Art und Weise, wie die EXE und die DLL „zusammenfinden“ muss nun noch einmal differenziert werden:

Statisches Laden

Beim statischen Laden, befinden sich in der ausführbaren Datei, bei den noch nicht

aufgelösten Funktionsnamen, Verweise auf den Namen der DLL, aus der die Funktionen exportiert werden. In dem Moment, in dem das Betriebssystem die EXE einliest, löst es die Abhängigkeiten auf: Die Funktionen der DLL werden mit denen der EXE verbunden. Dies hat jedoch einen entscheidenden Nachteil: DLLs können nicht nachträglich geladen – zum Beispiel um die Funktionalität des Hauptprogramms um zu erweitern (z.B. Plug-Ins). Außerdem stoppt der Ladeprozess des Programms, falls eine DLL nicht gefunden oder ein Funktionsname nicht aufgelöst werden kann.

Dynamisches Laden

In der ausführbaren Datei befinden sich keine Verweise auf irgendeine DLL. Anstatt von Funktionen hat der Programmierer Pointervariablen, die auf ausführbaren Code im Speicher zeigen können (Funktionspointer) im Programm hinterlassen. Sobald die Funktionen benötigt werden, lädt das Programm die DLL über eine entsprechende Betriebssystemfunktion und setzt die Funktionspointer auf die Adressen der Funktionen im Speicher.
Ist eine DLL nicht vorhanden, so kann das Programm selbst entscheiden, ob diese für die weitere Ausführung wichtig ist. Auch können verschiedene DLLs, die jedoch die gleichen Funktionen exportieren, geladen werden.

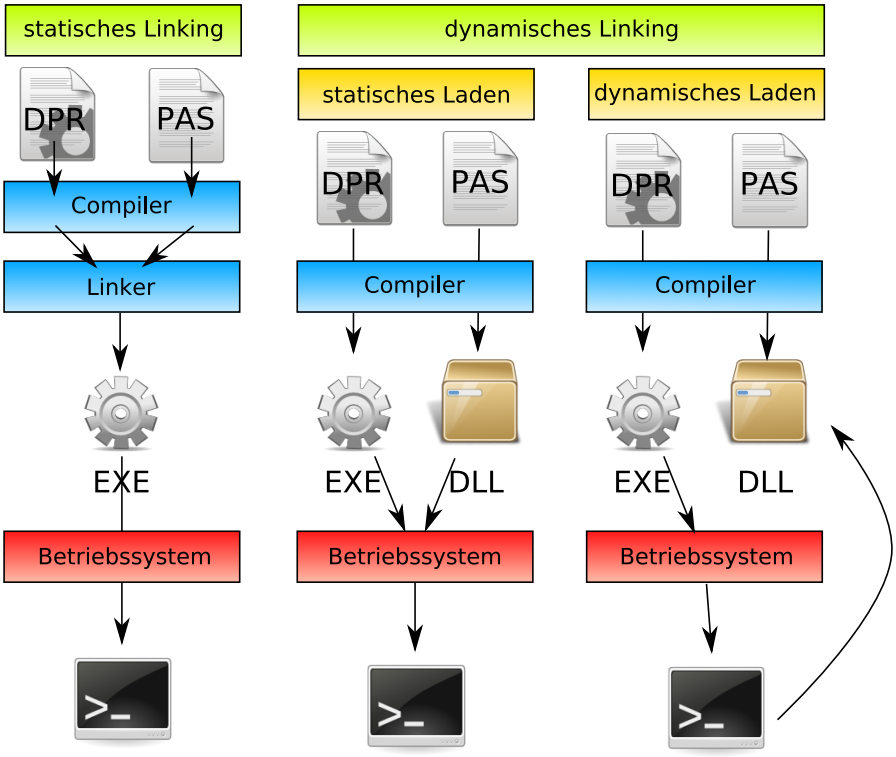


Abbildung A.1: Die verschiedenen Möglichkeiten des Bindeprozesses graphisch dargestellt. Wichtig: Delphi kann aus einer PAS-Quellcodedatei keine DLL erstellen - um dies zu erreichen muss auch hier ein eigenes Projekt erzeugt werden. Dieses Detail wurde hier jedoch der Einfachheit wegen außer Acht gelassen.

A.2 RTTI und das Registrarsystem

A.2.1 Die RTTI

Delphi bietet die Möglichkeit, Eigenschaften eines Objektes zu „veröffentlichen“. Diese veröffentlichten Eigenschaften lassen sich anschließend zur Laufzeit auslesen und verändern. Dies ist auch dann möglich, wenn vorher nichts über diese Klasse bekannt war. Verwendet wird dieses System zum Beispiel in der Delphi IDE: Der Objektinspektor zeigt die veröffentlichten Eigenschaften der Komponentenklassen an und erlaubt es diese zu verändern. Auch wenn neue Komponenten in die IDE installiert werden, so werden deren Eigenschaften im Objektinspektor angezeigt.
Andorra 2D verwendet diese Technik ebenfalls im GUI-System (siehe 3.3): Die Eigenschaften registrierter (siehe nächster Abschnitt) Komponenten werden automatisch in einem Objektinspektor angezeigt.

A.2.2 Das Registrarsystem

Wie in den vorherigen Kapiteln beschrieben, ermöglicht es Andorra 2D neue Module durch einfaches einbinden einer Unit hinzuzufügen. Dies wird durch folgendes System ermöglicht:
In der Unit „AdPersistent“ verwaltet Andorra 2D eine Liste registrierter Klassen (globaler Registrar). Diese Klassen müssen von der Basisklasse „TAdPersistent“ abgeleitet sein, welche das RTTI-System für diese und abgeleitete Klassen aktiviert. Beim Hinzufügen einer Klasse zur Liste, wird deren Name und ein Pointer auf ihre Klassendeklaration („TAdPersistentClass“) gespeichert. Eine Prozedur ermöglicht das Hinzufügen einer neuen Klasse, eine weitere das Auslesen der Klassendeklaration zu einem zugehörigen Klassennamen:

Aufbau des globalen Registrars

```
{Registriert eine Klasse und fügt sie zur Liste hinzu.}
procedure AdRegisterClass(AClass:TAdPersistentClass);
{Gibt die Klassendeklaration für einen Klassennamen zurück.}
function AdGetClass(AName:String):TAdPersistentClass;
```

Soll eine Klasse nun automatisch mit dem Einbinden der Unit registriert werden, so wird hierfür der „initialization“-Abschnitt der Unit verwendet: Befehle, die in diesem Abschnitt stehen, werden noch vor dem eigentlichen Hauptprogramm ausgeführt, sofern die Unit eingebunden ist.

Exemplarisch ist im Folgenden diese Funktionsweise anhand des Registrationssystems für die Klasse „TAdGraphicFormat“ aufgeführt: Die Unit „AdBitmap“, die deren Klassendeklaration beinhaltet, stellt eine Stringliste bereit (lokaler Registrar), in der alle registrierten, abgeleiteten Klassen von „TAdGraphicFormat“ aufgelistet werden. Außerdem ist eine Prozedur „RegisterGraphicFormat“ vorhanden, die die angegebene Klasse

zur Stringliste und globalen Registrar hinzufügt:

Spezielle Registrierungsfunktionen

```
unit AdBitmap;

interface

uses
  AdPersistent;

type
  TAdGraphicFormat = class(TAdPersistent)
    public
      function SupportsFile(AFile: string): boolean;
  end;

  TAdGraphicFormatClass = class of TAdGraphicFormat;

[...]

{Registriert die angegebene, von TAdGraphicFormat abgeleitete Klasse.}
procedure RegisterGraphicFormat(AClass:TAdGraphicFormatClass);

var
  {Beinhaltet die Namen aller registrierten Graphikformate.}
  RegisteredGraphicFormats:TStringList;

implementation

procedure RegisterGraphicFormat(AClass:TAdGraphicFormatClass);
begin
  AdRegisterClass(AClass);
  RegisteredGraphicFormats.Add(AClass.ClassName);
end;

[...]

initialization
  RegisteredGraphicFormats := TStringList.Create;

finalization
  RegisteredGraphicFormats.Free;

end.
```

In der Unit (hier als Beispiel „AdTGA“, die die zu registrierende Klasse beinhaltet, ist nun folgender Konstrukt vorhanden:

Registrieren einer Klasse

```
unit AdTGA;

interface
```

```
uses
  AdBitmap;

type
  TAdTGAFormat = class(TAdGraphicFormat)

[...]

initialization
  RegisterGraphicFormat(TAdTGAFormat);

end.
```

Möchte nun zum Beispiel die Klasse „TAdBitmap“ aus der Unit „AdBitmap“ eine Datei laden, so greift sie über folgendermaßen auf alle registrierten Formate zu:

Suchen nach einem bestimmten Lader

```
procedure TAdBitmap.LoadGraphicFromFile(AFile: string);
var
  tmp:TAdGraphicFormat;
  i:integer;
begin
  //Gehe alle registrierten Grafikformatklassen durch
  for i := 0 to RegisteredGraphicFormats.Count-1 do
  begin
    //Erzeugt eine Instanz der Klasse
    tmp := TAdGraphicFormatClass(
     AdGetClass(RegisteredGraphicFormats[i])).Create;

     //Testet, ob diese Klasse die angegebene Datei laden kann
     if tmp.SupportsFile(AFile) then
     begin
       //Lädt die Grafik in das Bitmap
       tmp.LoadGraphic(self);
       tmp.Free;

       //Bricht die Schleife ab
       break;
     end else
       tmp.Free;
  end;
end;
```

A.3 Das „Andorra 2D“ Projekt

A.3.1 Geschichte

Das Andorra 2D Projekt wurde von mir im Herbst 2006 ins Leben gerufen und sollte ursprünglich ein hardwarebeschleunigter Ersatz für das DirectDraw basierte „DelphiX“

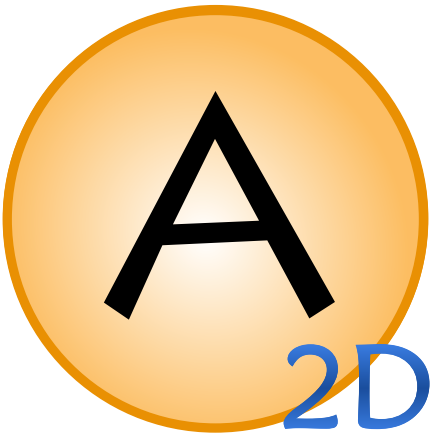


Abbildung A.2: Das Andorra 2D Logo

werden. Nach der ersten Veröffentlichung im Frühjahr 2007 (Version 0.1.5) ist die Popularität von Andorra 2D kontinuierlich gestiegen. Besonders nach jedem Release der Bibliothek stieg die Anzahl der Benutzer dauerhaft an. Mittlerweile wird die Bibliothek 20-40 mal am Tag heruntergeladen, insgesamt schon mehr als 13.000 mal. Dabei wurde eine Datenmenge von 170GB ausgetauscht [1].
Sucht man im Internet nach dem Begriff „Andorra 2D“, so finden sich unter anderem Beiträge zu der Bibliothek auf chinesischen, japanischen, russischen, spanischen, portugiesischen und arabischen Seiten.
Andorra 2D hat von mehreren Personen mitunter aufwendige Erweiterungen erhalten. Fast jede Woche treffen zudem E-Mails mit Fehlerbeschreibungen oder Fragen ein, die helfen die Bibliothek weiter zu verbessern.
Während der bisherigen Entwicklungszeit von Andorra 2D kamen zudem zwei Anfragen, gegen Bezahlung bestimmte Features zu implementieren oder Support zu leisten. Diesen Anfragen mussten wegen mangelnder Zeit jedoch Absagen erteilt werden.

A.3.2 Internetauftritt und Dokumentation

Es gibt zahlreiche exzellente Opensourceprojekte, die „in der Versenkung versinken“, oder deren Wert unterschätzt wird, weil deren Internetauftritt kaum Informationen über das Projekt beinhaltet. Zudem ist eine umfassende Dokumentation der Software von Nöten: Dazu zählen Tutorials für Einsteiger und eine ausführliche Dokumentation des Quelltextes. Auch sollten viele Beispielprogramme vorhanden sein, die die Verwendung der Bibliothek demonstrieren. Die Dokumentation und der Internetauftritt sollten vorzugsweise in englischer Sprache sein, um eine möglichst große Zielgruppe erreichen zu können.
Um ein Projekt bekannt zu machen, eigenen sich unter anderem Entwicklerforen. Besonders, wenn man in dem Forum selbst aktiv ist, wird die Software gerne von anderen,

[1] Daten entnommen von `http://sourceforge.net/project/stats/?group_id=182128&ugn=andorra&type=&mode=alltime`

dort vertretenen, Entwicklern verwendet. Auch über die „Projekte, die dieses Projekt verwenden“-Sektion von eingebundenen Bibliotheken ist eine gewisse Bekanntheit zu erreichen.

A.3.3 Opensource

Was ist Opensource?

Andorra 2D ist unter einer Opensource-Lizenz veröffentlicht. „Opensource“ bedeutet, dass der Quellcode des Projektes „frei“ ist. Dies ist nicht mit „kostenlos“ zu verwechseln: Darüber hinaus wird dem Nutzer des Quelltexts das Recht gegeben, diesen auf jegliche Weise zu nutzen: umschreiben, ausdrucken, verbessern, oder sogar verkaufen. Für diese Freiheit wird jedoch folgendes von den Nutzern eingefordert: Der veränderte oder publizierte Quellcode muss unter der Ursprungslizenz weitergegeben werden [5].
Andorra 2D ist unter der „Common Public License“(CPL) veröffentlicht. Im Unterschied zu manch anderen „restriktiveren“ Opensource-Lizenzen müssen Programme, die auf dem Quellcode aufbauen, nicht unter dieser Lizenz veröffentlicht werden - nur Änderungen an der Bibliothek selbst müssen wie oben beschrieben wieder öffentlich zugänglich gemacht werden [7].

Warum Opensource?

Viele Menschen stellen nun die Frage, warum man sein geistiges Eigentum großzügig im Internet verteilt. Schließlich könnte man seine Programme ja auch verkaufen und Geld damit verdienen. Folgende Punkte sprechen jedoch für Opensource [1]:

- **Vergrößerte Benutzergruppe:** Die wenigsten Entwickler, die freie Bibliotheken/Programme verwenden, würden diese auch verwenden, wenn sie dafür etwas bezahlen müssten. Opensource macht die Software einer größeren Benutzergruppe zugänglich.
- **Ständige Verbesserung der Codebasis:** Je mehr Benutzer die Bibliothek verwenden, desto wahrscheinlicher ist es, Fehlerbehebungen oder sogar Erweiterungen zu erhalten.
- **Weniger Verwaltungsaufwand:** Möchte man seine Programme verkaufen, so ist dies mit sehr viel bürokratischem Aufwand verbunden. Zudem werden gleichzeitig eventuelle Schadensersatz- und Supportansprüche der Käufer fällig. Das Verkaufen von Bibliotheken/Software ist nicht als „Nebenjob“ möglich.
- **Software sollte frei sein:** Jeder sollte das Recht haben, im Detail nachvollziehen zu können, was ein Softwareprodukt tut. Wer würde sich ein Auto kaufen, wenn er nicht dessen Motorhaube öffnen könnte?
- **Kostenloser Webspace für das Projekt:** Zahlreiche (zum Teil werbeinanzierte) Dienste bieten kostenlose Server und Webspace für Opensourceprojekte an.

- **Beitrag zur Entwicklergemeinschaft:** Wer verwendet nicht gerne Programme wie Mozilla Firefox, OpenOffice, Gimp oder Inkscape? Durch das Veröffentlichen der eigenen Software unter Opensource-Lizenzen kann man seinen eigenen Teil zur Entwicklergemeinschaft beitragen.
- **Mit Opensource kann man Geld verdienen:** Stellt man das Opensource-Projekt unter eine restriktivere Opensource-Lizenz, so kann man Kunden eine komerzielle Lizenz anbieten, die es ermöglicht auf der Bibliothek aufbauende Software zu verkaufen, ohne den eigenen Sourcecode veröffentlichen zu müssen. Außerdem lassen sich zum Beispiel Supportverträge verkaufen.

B Beispielprogramme

B.1 Auf dem Grafikplugin basierendes Programm

Das folgende Programm verwendet direkt die Grafik-API Abstraktionsebene um ein Fenster anzuzeigen und darin ein farbiges, texturiertes, Dreieck rotieren zu lassen. Einzig zum Laden der Textur werden Objekte, die nicht direkt aus der untersten Abstraktionsebene stammen, verwendet.

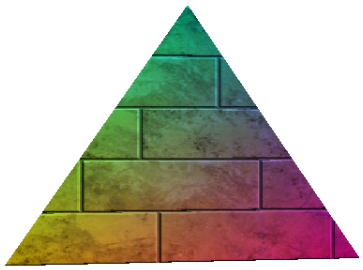


Abbildung B.1: Ausgabe des Beispielprogramms

plugin_based.dpr

```
program Project1;

uses
  SysUtils,

  //Diese Units enthalten von Andorra 2D benötigte Typ- und Klassendeklarationen
  AdClasses, AdTypes, AdMath, AdWindowFramework,

  //Diese Units werden verwendet um das Plugin zu laden und ein Fenster zu erzeugen
  AdDLLLoader, AdWin32Window,

  //Diese Units gehören streng genommen nicht zur untersten Abstraktionsschicht,
  //irgendwie müssen wir aber ein Bild laden und die Zeit, die zwischen zwei
  //Frames vergeht, messen.
  AdBitmap, AdPNG, AdPerformanceCounter;

type
  //Diese Klasse bildet unsere Testanwendung.
  TAdTestAppl = class
    private
      FLoader: TAdDllLoader;
      FAppl: TAd2DApplication;
      FMesh: TAd2DMesh;
      FTexture: TAd2DBitmapTexture;
      FWnd: TAdWindowFramework;
      FRotY: Single;
      FPerCounter: TAdPerformanceCounter;
```

```
      procedure Idle(Sender: TObject; var Done: boolean);
    public
      procedure Run;
  end;

{ TAdTestAppl }

procedure TAdTestAppl.Run;
var
  props: TAdDisplayProperties;
  verts: TAdVertexArray;
  bmp: TAdBitmap;
begin
  //Erzeuge die "DLL-Ladeklasse"
  FLoader := TAdDllLoader.Create;

  //Lade die OpenGL Bibliothek. Natürlich kann hier auch die DirectX Bibliothek
  //geladen werden.
  FLoader.LoadLibrary('AndorraOGL.dll');

  if FLoader.LibraryLoaded then
  begin
    //Das Laden der Dll ist geglückt. Lege nun die Ausmaße des Ausgabefensters
    //fest.
    props.Width := 800;
    props.Height := 600;
    props.Mode := dmWindowed;
    props.BitDepth := ad32Bit;

    //Erzeuge das Fenster
    FWnd := TAdWin32Window.Create;

    //Binde das Fenster an nichts und initialisiere es.
    if (FWnd.BindTo(nil)) and (FWnd.InitDisplay(props)) then
    begin
      //Ist dieser Schritt geglückt, so erzeuge die TAd2dApplication Hauptklasse
      //aus dem Grafikplugin
      FAppl := FLoader.CreateApplication;

      //Überprüfe, ob das Grafikplugin mit dem von uns gewähltem Fenstersystem
      //zurechtkommt.
      if not FAppl.SupportsWindowFramework(FWnd.IdentStr) then
        exit;

      //Initialisiere die Anzeige
      FAppl.Initialize(FWnd);

      //Setze einige Anzeigeoptionen
      FAppl.SetOptions([aoBlending, aoTextures]);

      //Erstelle eine 3D-Projektionsmatrix:
      FAppl.Setup3DScene(2, 2,       //Ausmaße der Szene (2 LE breit und
                                     //2 LE hoch)
        AdVector3(0, 0, -5),         //Position der Kamera
        AdVector3(0, 0, 0),          //Punkt, auf den die Kamera blickt
        AdVector3(0, -1, 0), 10, 0); //Ausrichtung der Kamera (wo ist oben)
                                     //sowie vordere und hintere Clipping-Plane

      //Erzeuge eine Bitmaptextur
      FTexture := FAppl.CreateBitmapTexture;

      //Erzeuge ein Bitmap und lade ein Bild
      bmp := TAdBitmap.Create;
      bmp.LoadGraphicFromFile('resources/floor_tex.png');

      //Speichere das geladene Bitmap in der Textur
```

```
      FTexture.LoadFromBitmap(bmp, ad32Bit);

      //Gebe den Speicher für das Bitmap frei
      bmp.Free;

      //Erzeuge ein Meshobjekt
      FMesh := FAppl.CreateMesh;

      //Erzeuge einen Vertexarray mit der Länge drei.
      SetLength(verts, 3);

      //Setze die Vertexdaten für ein einfaches, buntes und texturiertes
      //Dreieck
      verts[0].Position := AdVector3(0, -1, 0);
      verts[0].Color := Ad_ARGB(255, 0, 255, 255);
      verts[0].Normal := AdVector3(0, 0, -1);
      verts[0].Texture := AdVector2(0.5, 0);

      verts[1].Position := AdVector3(1, 1, 0);
      verts[1].Color := Ad_ARGB(255, 255, 255, 0);
      verts[1].Normal := AdVector3(0, 0, -1);
      verts[1].Texture := AdVector2(1, 1);

      verts[2].Position := AdVector3(-1, 1, 0);
      verts[2].Color := Ad_ARGB(255, 255, 0, 255);
      verts[2].Normal := AdVector3(0, 0, -1);
      verts[2].Texture := AdVector2(0, 1);

      //Lade die Vertexdaten in das Meshobjekt
      FMesh.Vertices := verts;
      FMesh.Indices := nil;
      FMesh.PrimitiveCount := 1;

      //Setze die Textur
      FMesh.Texture := FTexture;

      //Übergebe die Daten an das zugrundeliegende Grafiksystem
      FMesh.Update;

      //Erzeuge ein "PerformanceCounter" Objekt - misst die Zeit, die zwischen
      //zwei Bildern vergeht
      FPerCounter := TAdPerformanceCounter.Create;

      //Verknüpfe das "OnIdle-Event" des Windowframeworks mit unserer "Idle"
      //Prozedur und setze den Fenstertitel
      FWnd.Events.OnIdle := Idle;
      FWnd.Title := 'Andorra 2D Plugin Test Application';

      //Übergebe die Kontrolle an das Betriebsystem und warte auf Ereignisse.
      //Die Prozedur ist beendet, sobald das Fenster geschlossen wird
      FWnd.Run;

      //Räume auf.
      FPerCounter.Free;
      FMesh.Free;
      FTexture.Free;
      FAppl.Finalize();
    end;
    FWnd.Free;
  end;
  FLoader.Free;
end;

//Diese Funktion wird periodisch vom Windowframework aufgerufen.
procedure TAdTestAppl.Idle(Sender: TObject; var Done: boolean);
var
```

```
  mat: TAdMatrix;
begin
  FAppl.ClearSurface(
    //Lösche den aktuellen Bildschirminhalt in der Region von (0|0) bis (800|600),
    AdRect(0, 0, 800, 600),
    //Alle verfügbaren Puffer...
    [alColorBuffer, alZBuffer, alStencilBuffer],
    //...mit schwarzer Farbe, setze den Z- und den Stencilbuffer zurück.
    Ad_ARGB(255, 0, 0, 0), 0, 0);

  //Mit diesem Befehl werden die eigentlichen Zeichenoperationen eingeleitet
  FAppl.BeginScene;

  //Messe die Zeit, die seit dem letzen Bild vergangen ist (TimeGap),
  FPerCounter.Calculate;
  //Erhöhe "FRotY" mit der Geschwindigkeit 0.001/s
  FRotY := FRotY + 0.001 * FPerCounter.TimeGap;
  //Erzeuge eine Rotationsmatrix für eine Rotation um die Y-Achse mit dem
  //Winkel FRotY (Bogenmaß)
  mat := AdMatrix_RotationY(FRotY);
  //Weise die Rotationsmatrix dem Mesh zu
  FMesh.Matrix := mat;

  //Zeichne das Mesh im normalen "Blendmodus" (bmAlpha) und als
  //Dreiecksprimitiven
  FMesh.Draw(bmAlpha, adTriangles);

  //Beende die Szene
  FAppl.EndScene;

  //Zeige die Ergebnisse an
  FAppl.Flip;

  //Durch das Setzen von "done" auf false wird die Idle-Prozedur in einer Endlosschleife
  //ständig ausgeführt.
  done := false;
end;

//Hier folgt das eigentliche Hauptprogramm...
var
  appl: TAdTestAppl;

begin
  //Ein Objekt der oben definierten "TAdTestAppl" Klasse wird angelegt...
  appl := TAdTestAppl.Create;
  try
    //...laufen lassen...
    appl.Run;
  finally
    //...und schließlich freigegeben
    appl.Free;
  end;
end.
```

B.2 Auf der 2D-Funktionalitätsebene basierendes Programm

Im folgenden Programm wird ausschließlich die 2D-Funktionalitätsebene verwendet, um ein Fenster zu öffnen, ein Bild zu laden und dieses über den Bildschirm zu bewegen. Den meisten Platz im Code nimmt das Berechnen der Koordinaten für das Bild ein.

func2d_based.dpr

```
program func2d_based;

uses
  SysUtils,

  //Farbkonstanten, Performancecounter
  AdConsts,  AdPerformanceCounter,

  //Die Hauptklassen der 2D-Funktionalitätsebene
  AdDraws,

  //Registriere ein beliebiges Windowframework
  AdWin32Window,

  //Registriere den Handler für PNG-Bilder
  AdPNG;

type
  TAdAppl = class
    private
      AdDraw: TAdDraw;
      AdImage: TAdImage;
      AdPerformanceCounter: TAdPerformanceCounter;
      AX, AY: Single;
      ASpeedX, ASpeedY: Single;
      procedure Idle(Sender: TObject; var done: boolean);
    public
      procedure Run;
  end;

var
  appl: TAdAppl;

{ TAdAppl }

procedure TAdAppl.Run;
var
  err: string;
begin
  //Erzeuge eine neue Instanz von TAdDraw, binde sie an kein vorhandenes
  //Fenster (nil)
  AdDraw := TAdDraw.Create(nil);

  //Setze die zu ladende DLL
  AdDraw.DllName := 'AndorraOGL.dll';

  //Setze die Größe des Fensters:
  with AdDraw.Display do
  begin
    Width := 1024;
    Height := 768;
  end;

  //Initialisiere die Zeichenfläche
  if AdDraw.Initialize then
  begin
    //Erzeuge ein Image-Objekt
    AdImage := TAdImage.Create(AdDraw);
    //Lade die Textur
    AdImage.Texture.LoadGraphicFromFile('resources/andorra_systems.png');
    //Passe das interne Mesh-Objekt an die Ausmaße der Textur an
    AdImage.Restore;

    //Erzeuge den Performance-Counter
```

```
    AdPerformanceCounter := TAdPerformanceCounter.Create;

    //Setze den Titel der Anwendung
    AdDraw.Window.Title := 'Andorra 2D';

    //Verknüpfe das "OnIdle"-Event mit unserer "Idle" Funktion
    AdDraw.Window.Events.OnIdle := Idle;

    //Setze die Bewegungsgeschwindigkeit des Bildes auf 0.1 Pixel/ms
    ASpeedX := 0.1;
    ASpeedY := 0.1;

    //Übergebe die Kontrolle and das Betriebsystem. Die Funktion kehrt zurück,
    //wenn das Fenster geschlossen wurde
    AdDraw.Run;

    //Gebe alle erzeugten Objekte frei
    AdPerformanceCounter.Free;
    AdImage.Free;
  end else
  begin
    //Die Initialisierung ist fehlgeschlagen. Gebe die Fehlerursache aus.
    err := AdDraw.GetLastError;
    AdDraw.Free;
    raise Exception.Create(err);
  end;
end;

procedure TAdAppl.Idle(Sender: TObject; var done: boolean);
begin
  if AdDraw.CanDraw then
  begin
    //Berechne die Zeit, die seit dem letzten Aufruf von "Calculate"
    //vergangen ist.
    AdPerformanceCounter.Calculate;

    //Bewege das Bild und lasse es am Rahmen des Fenster abprallen
    AX := AX + ASpeedX * AdPerformanceCounter.TimeGap;
    AY := AY + ASpeedY * AdPerformanceCounter.TimeGap;

    if AX < 0 then
    begin
      AX := 0;
      ASpeedX := -ASpeedX;
    end;
    if AX > AdDraw.Width - AdImage.Width then
    begin
      AX := AdDraw.Width - AdImage.Width;
      ASpeedX := -ASpeedX;
    end;
    if AY < 0 then
    begin
      AY := 0;
      ASpeedY := -ASpeedY;
    end;
    if AY > AdDraw.Height - AdImage.Height then
    begin
      AY := AdDraw.Height - AdImage.Height;
      ASpeedY := -ASpeedY;
    end;


    //Fülle den Hintergrund mit schwarzer Farbe
    AdDraw.ClearSurface(AdCol32_Black);

    //Beginne die Zeichenoperationen
```

```
    AdDraw.BeginScene;

    //Zeichne das Bild an die vorher berechnete Stelle
    AdImage.Draw(AdDraw, AX, AY, 0);

    //Beende die Zeichenoperationen
    AdDraw.EndScene;

    //Zeige das Gezeichnete an
    AdDraw.Flip;
  end;

  Done := false;
end;

//Das eigentliche Hauptprogramm
begin
  appl := TAdAppl.Create;
  try
    appl.Run;
  finally
    appl.Free;
  end;
end.
```

Materialverzeichnis

Beiliegende CD

Auf der beiliegenden CD ist die Version 0.4.5.1 von Andorra 2D vom 31. Dezember 2008 enthalten. Alle Quelltexte in diesem Dokument beziehen sich auf diese Version.
Die CD ist in folgende Verzeichnisse untergliedert:

ad2d_451	Enthält das Andorra 2D Paket.
src	Beinhaltet den Andorra 2D Sourcecode.
dll	Enthält die Andorra 2D Plugins (DirectX und OpenGL Code).
lib	Enthält einige benötigte Header, wenn die Plugins oder weitere Komponenten verwendet werden sollen.
demos	Enthält den Sourcecode der Demoprogramme.
tools	Enthält Werkzeuge/Editoren, die den Umgang mit Andorra 2D erleichtern sowie deren Sourcecode.
bin	Enthält optional benötigte DLLs, die kompilierten Andorra 2D DLLs, sowie die kompillierten Andorra 2D Demos.
source	Enthält die für dieses Dokument geschriebenen Beispielanwendungen aus Anhang. B.
andorraplayer	Enthält den in Kapitel 3.2 gezeigten Videospieler
doc	Dieses Dokument wurde mit dem Textsatzprogramm LaTeX erstellt. Der komplette Quellcode dieses Dokumentes sowie alle dazugehörigen Dateien befinden sich in diesem Verzeichnis.
directx_runtime	Enthält den unten angesprochenen „Web Installer“ für DirectX 9c.
website	Enthält eine Kopie der Andorra 2D Website (Stand: 21. Juni 2009) .

Hardware- und Softwareanforderungen

Die Anwendungen auf der CD wurden für das Betriebsystem Windows ab Version 2000 kompiliert. Die Ausführbarkeit wurde bis inklusive Windows 7 erfolgreich getestet.

> **Die Programme sollten vor dem Starten auf Festplatte kopiert werden!**

Zum Verwenden des DirectX-Modus' wird DirectX 9c April 2007 benötigt. Ein „Web Installer", der alle fehlenden DirectX-Versionen herunterlädt, kann im Verzeichnis „directx_runtime" gefunden werden. Für die korrekte Ausführung aller Demo-Anwendungen ist eine Grafikkarte von nöten, die den DirectX 9c Standards entspricht: Dazu zählen Unterstützung für Shader, Occlusion Queries und Frame Buffer Objects. Ältere Grafikarten bzw. integrierte Grafikchips auf Notebooks etc. erfüllen diese Vorraussetzungen eventuell nicht. Unter diesen Umständen kann es beim Ausführen der Demos „Worm Hunter", „Pixel Check" und „Shader" zu Problemen kommen.

Website

Auf dem Internetauftritt von Andorra 2D, der unter

> `http://andorra.sourceforge.net/`

gefunden werden kann, findet sich die jeweils neuste Version, Anleitungen (Tutorials) zur Verwendung von Andorra 2D in eigenen Softwareprojekten, sowie weitere Informationen über das Projekt.

Beiträge von anderen Benutzern

Aufgrund des „Open Source"-Charakters der Bibliothek wurden einige Module nicht komplett von mir entwickelt. Ist dies der Fall, so ist dies entweder im Kopf der Quellcodedatei bzw. an den entsprechenden Stellen im Quellcode vermerkt.

Literaturverzeichnis

[1] BSI: *Bundesamt für Sicherheit in der Informationstechnik — Fragen & Antworten zu Open Source Software*. 2009. – URL `http://www.bsi-fuer-buerger.de/opensource/11_02.htm`

[2] Khronos: *OpenGL Specifications — Khronos Group*. 2009. – URL `http://www.opengl.org/documentation/specs/`. – [Online; Stand 13. Juni 2009]

[3] Microsoft: *About DirectDraw — Microsoft Developer Network Library*. 2009. – URL `http://msdn.microsoft.com/en-us/library/ms796322.aspx`. – [Online; Stand 14. Juni 2009]

[4] Microsoft: *Direct3D Driver — Microsoft Developer Network Library*. 2009. – URL `http://msdn.microsoft.com/en-us/library/ms793073.aspx`. – [Online; Stand 13. Juni 2009]

[5] OSI: *Homepage der Open Source Initiative*. 2009. – URL `http://www.opensource.org/`. – [Online; Stand 1. Juni 2009]

[6] Rudolph, Alexander: *3D-Effekte für Spieleprogrammierer*. Markt und Technik, 2005

[7] Wikipedia: *Common Public License — Wikipedia, The Free Encyclopedia*. 2009. – URL `http://en.wikipedia.org/w/index.php?title=Common_Public_License&oldid=286562589`. – [Online; Stand 28. April 2009]

[8] Wikipedia: *DirectX — Wikipedia, Die freie Enzyklopädie*. 2009. – URL `http://de.wikipedia.org/wiki/DirectX#DirectX_Graphics`. – [Online; Stand 14. Juni 2009]

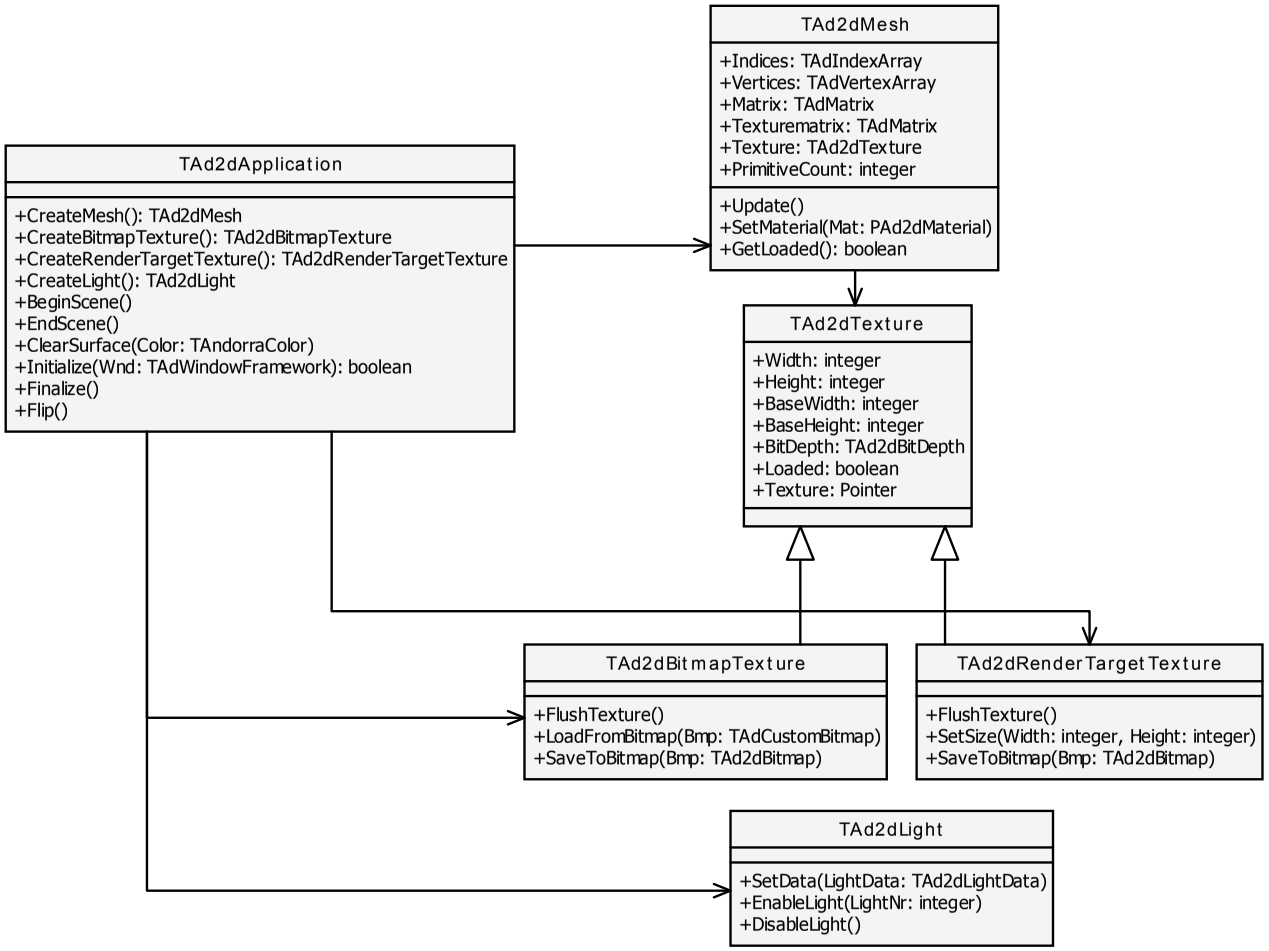
[9] Wikipedia: *Moving Picture Experts Group — Wikipedia, Die freie Enzyklopädie*. 2009. – URL `http://de.wikipedia.org/w/index.php?title=Moving_Picture_Experts_Group&oldid=58424029`. – [Online; Stand 21. Juni 2009]

[10] Wikipedia: *Transform and Lighting — Wikipedia, Die freie Enzyklopädie*. 2009. – URL `http://de.wikipedia.org/w/index.php?title=Transform_and_Lighting&oldid=55656883`. – [Online; Stand 1. Juni 2009]

[11] Wikipedia: *Videokompression — Wikipedia, Die freie Enzyklopädie*. 2009. – URL `http://de.wikipedia.org/w/index.php?title=Videokompression&oldid=61081366`. – [Online; Stand 21. Juni 2009]

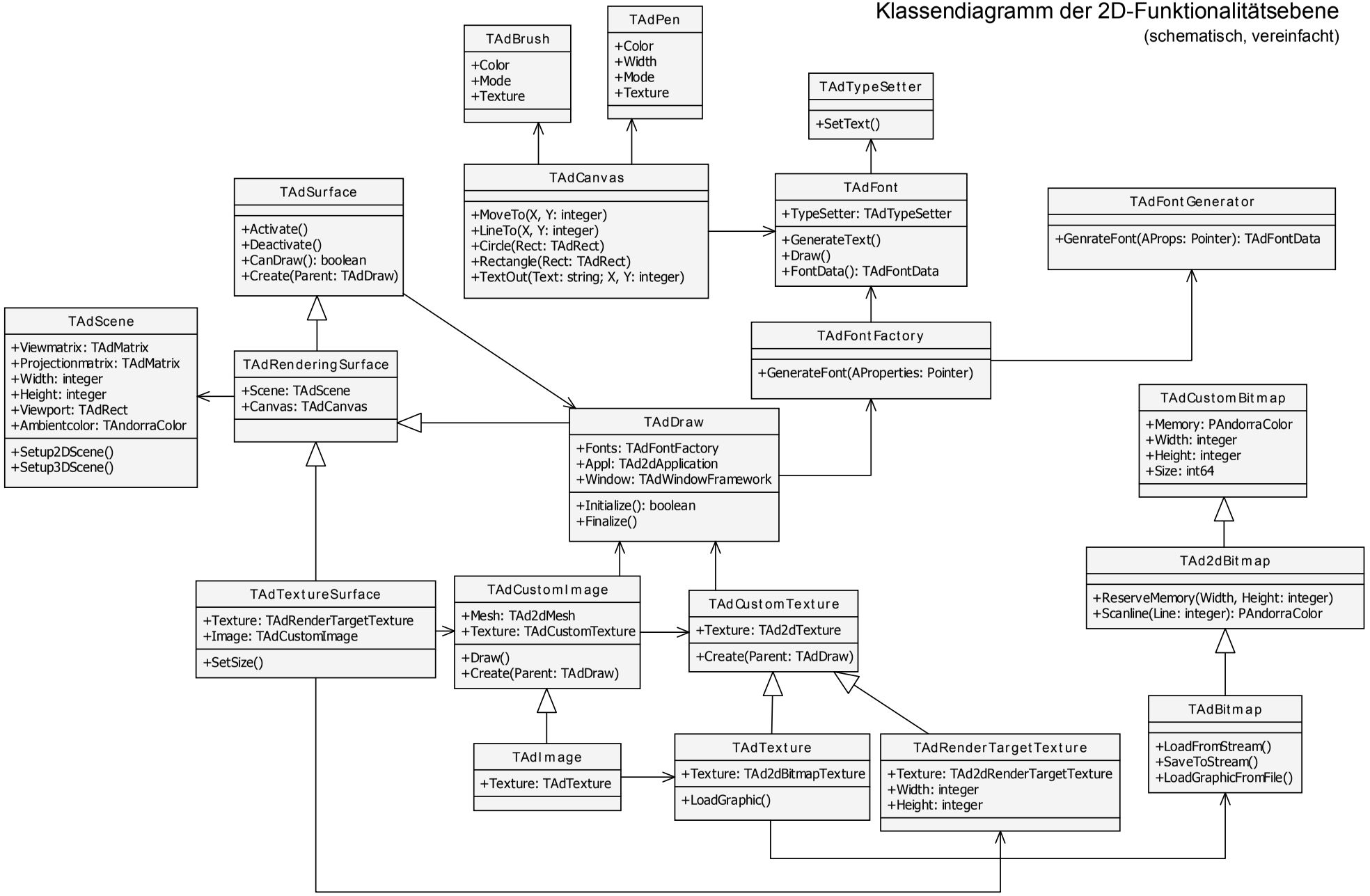
Klassendiagramm der Grafik-API Abstraktionsebene

(schematisch, vereinfacht)



Klassendiagramm der 2D-Funktionalitätsebene

(schematisch, vereinfacht)



Ich erkläre hiermit, dass ich dieses Dokument ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe. Wie in den vorhergehenden Kapiteln beschrieben, basiert ein Teil des Computerprogramms, auf das sich dieses Dokument bezieht, auf der Mitarbeit anderer Personen.

_____, den	_____	_____
Ort	Datum	Unterschrift