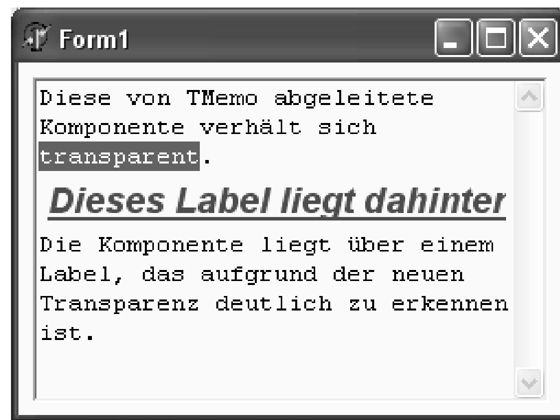
Oliver Friedmann

Delphis Komponenten Transparenz verleihen

In vielen Fällen wäre es wünschenswert, Borland hätte allen Komponenten eine Transparenz verleihende Eigenschaft mit auf den Weg gegeben. Beschäftigt man sich jedoch etwas genauer mit dieser Thematik, wird einem klar, warum dies nicht der Fall ist. Das Problem lässt sich dennoch lösen.



Von Komponenten die Möglichkeit zur Transparenz zu verlangen, erscheint zunächst recht und billig. Sie ist auch in vielen Fällen möglich, wie sich an der *Transparent*-Eigenschaft von *TLabel*, *TSpeedButton* oder *TImage* leicht erkennen und ausprobieren lässt. Anderen Komponenten fehlt jedoch die Fähigkeit zur Transparenz. So ist es beispielsweise nicht möglich, *TButton*, *TEdit*, *TMemo*, *TPanel* oder auch *TListView* transparent zu gestalten. Dies fällt in der Regel auch nicht unangenehm auf, weil sie sich durch die *Color*-Eigenschaft leicht an den Hintergrund anpassen lassen. Ist der Hintergrund jedoch nicht einfarbig, weil er beispielsweise eine Bildtextur enthält oder einen Farbverlauf, trübt die fehlende Transparenz der platzierten Komponenten das Gesamtdesign. Bei *TButton* ist es sogar noch schlimmer: Nicht einmal seine Farbe ist veränderbar.
Es stellt sich also in erster Linie die Frage, warum Borland nicht allen visuellen Komponenten eine Transparenz-Eigenschaft spendiert hat.

Zwei grundverschiedene Vorfahren

Die Antwort auf diese Frage ist in den grundverschiedenen Vorfahren visueller Komponenten zu suchen. Alle visuellen Komponenten leiten sich von *TControl* ab, wobei man in diesem Zusammenhang zwei Arten von Komponenten unterscheiden muss.
Die einen leiten sich von *TGraphicControl*, die anderen von *TWinControl* ab. *TGraphicControl* und *TWinControl* wiederum sind direkte Nachfahren von *TControl*. Worin liegt nun der entscheidende Unterschied? Nachfahren von *TGraphicControl* besitzen selbst kein Window-*Handle*. Sie verwenden das Handle ihres übergeordneten Fensters (*Parent*) und damit auch dessen *Display Context*. Bei Nachfahren von *TGraphicControl* handelt es sich also um keine Fenster im Sinne des Windows API, sondern um eine delphi-eigene Technik. Aus diesem Grund ist es für von *TGraphicControl* abgeleitete Komponenten auch ein Leichtes, transparent zu sein: Sie übermalen ganz einfach den Hintergrund ihres übergeordneten Fensters nicht, denn sie verwenden denselben *Display Context*. Und der verhält sich wie ein Blatt Papier. Nur wenn man darauf zeichnet, wird zuvor Gemaltes verdeckt. Wird also der Hintergrund einer von *TGraphicControl* abgeleiteten Komponenten nicht übermalt, so ist sie transparent. Die von mir zu Anfangs erwähnten transparenten Komponenten *TLabel*, *TSpeedButton* und *TImage* sind – Sie ahnen es vermutlich schon – direkte Nachfahren von *TGraphicControl*.
Die anderen, nicht-transparenten Komponenten, wie *TButton* oder *TMemo*, sind nicht von *TGraphicControl* abgeleitet, sondern von *TWinControl*. Im Unterschied zu *TGraphicControl* handelt es sich bei *TWinControl* um echte, untergeordnete Fenster in Sinne des Windows API. Sie besitzen also ein eigenes *Handle* und einen eigenen *Display Context*. Und aus genau diesem Grund sind diese Komponenten nicht transparent. Wird das übergeordnete Fenster neu gezeichnet, so geschieht das im *Display Context* des übergeordneten Fensters. Auf den *Display Context* des untergeordneten *TWinControl* hat dies leider keine Auswirkungen. Unsere Aufgabenstellung hat sich damit also ein wenig modifiziert. Sie lautet nun: Wie kann man Komponenten, die von *TWinControl* abgeleitet sind, Transparenz verleihen?

Die triviale Lösung funktioniert nicht

Die triviale Lösung, die einem intuitiv vorschweben würde, ist natürlich, den *Display Context* des *TWinControl* durch eine Einstellung transparent zu machen. Die passende Option scheint sich im *Extended Style* eines Fensters zu befinden: *WS_EX_TRANSPARENT*. Doch leider führt das Setzen dieser Eigenschaft nicht zum erwünschten Ziel. Wird das übergeordnete Fenster neu gezeichnet, so wird der *Display Context* des „transparenten" *TWinControl* damit zwar überschrieben. Es erhält auf der einen Seite aber nicht die Aufforderung, sich selbst neu zu zeichnen (*WM_PAINT*), und auf der anderen Seite ist es auch nicht möglich, dass – wenn das „transparente" *TWinControl* neu gezeichnet werden möchte – die untergeordneten Fenster dabei transparent hindurchscheinen. Dieses Verhalten lässt sich auch der Windows API Hilfe (*Windows SDK*) entnehmen.
Wir müssen also auf die Hilfe von Windows verzichten und uns unsere Lösung selbst basteln, was sich als aufwändiger erweist, als man möglicherweise vermuten mag.

Die Lösung des Problems zerlegen

Um unser Problem zu lösen, wollen wir es zunächst in die verschiedenen Bestandteile zerlegen: Wir müssen es schaffen, den (transparent erscheinenden) Hintergrund einer Komponente zeichnen zu lassen, den Vordergrund der Komponente darüber maskiert zu zeichnen, und es mitzubekommen, wann sich der Hintergrund der Komponente ändert. Außerdem müssen wir davon ausgehen, dass die schon existierenden Komponenten, die wir transparent machen wollen, nicht kooperativ sein werden, das heißt: Wir müssen, um die Transparenz einer Komponente zu erhalten, mit ihren Eigenheiten zurecht kommen.
Aus diesem Grund macht es Sinn, die oben genannten drei Teilaufgaben zunächst für eine kooperative Komponente zu entwickeln, und sie sodann an nicht-kooperative Komponenten anzupassen. Als kooperative Komponente erzeugen wir uns ein *TCustomControl*-Derivat, welches den Hintergrund seines *Display Context* nicht übermalt, sondern beispielsweise nur einen Text ausgibt. *TCustomControl* ist ein direkter Nachfolger von *TWinControl*, welches lediglich den eigenen *Display Context* in Form eines *TCanvas* delphigerecht kapselt.
Da wir ja im Endeffekt schon bestehende Komponenten mit der Fähigkeit zur Transparenz nachrüsten wollen, müssen wir versuchen, den Anpassungsaufwand für die jeweilige Komponente zu minimieren. Da wir jedoch relativ viel Code schreiben werden und auch müssen, um an unser Ziel zu gelangen, und da ferner Mehrfachvererbung in Delphi nicht möglich ist, wollen wir eine sogenannte Subklasse erstellen, welche die Aufgabe der Transparenz übernimmt. Der Sinn dahinter ist, dass wir in derjenigen Komponente, die transparent werden soll, lediglich eine Instanz der Subklasse erzeugen, und die Subklasse dann die ganze Arbeit übernimmt.
Die Subklasse selbst wird jedoch bestimmte Fensternachrichten, sogenannte *Messages*, die an ihr *TWinControl* gesandt werden, abfangen müssen, um beispielsweise im rechten Augenblick die Arbeit des Zeichnens (*WM_PAINT*) übernehmen zu können.

Das Abfangen von Nachrichten

Um das Abfangen von Nachrichten eines *TControl*, auch *Hooken* genannt, zu verallgemeinern, habe ich eine Hilfsklasse namens *TControlHookHelperClass* verfasst. Sie fängt die Nachrichten dadurch ab, dass sie die *WindowProc* des *TControl* auf eine eigene Methode setzt, und somit die Nachrichten als Primärquelle empfangen kann, und natürlich außerdem die vorhergehende *WindowProc* speichert, sodass, nachdem die Nachrichten empfangen wurden, sie an die ursprüngliche Botschafts-Behandlungsroutine weitergegeben werden können. Die Hilfsklasse verfügt über zwei Ereignisse, einmal *OnWndProc*, welches ausgelöst wird, wenn eine Botschaft eintrifft, und *OnHandleChange*, welches ausgelöst wird, wenn sich die *Handle* des *TControl* ändert.
Trickreich an dieser Hilfsklasse ist, dass auch ohne Probleme mehrere Hilfsklassen die Nachrichten ein und desselben *TControl* abfangen können, ohne sich beim Überschreiben der *WindowProc* in die Quere zu kommen. Die Erläuterung dieses Algorithmus würde zu weit führen und den Rahmen dieses Artikels sprengen, kann jedoch am Quelltext und der Kommentierung nachvollzogen werden.

Die Subklasse erzeugen

Wie schon oben angedeutet, wollen wir nun eine Subklasse erzeugen, welche für die betreffende Komponente das Erzeugen von Transparenz übernehmen soll. Wir leiten sie von *TPersistent* ab, um die Eigenschaften, welche wir der Subklasse verpassen werden, von Delphis *Streaming-System* speichern zu lassen. Die Komponente also, welche die Fähigkeit zur

Transparenz erhalten soll, muss nur eine Instanz der Subklasse erzeugen, und sie dem *Objektinspektor* über das *Published*-Feld zu Verfügung zu stellen.
Da wir ja zunächst von einem kooperativen *TCustomControl*-Derivat ausgehen wollen, nennen wir unsere Basissubklasse dementsprechend *TCustomControlTransparency*.
Auf jeden Fall muss unsere Subklasse, wie schon erläutert, die Botschaften des *TWinControl* abfangen können. Aus diesem Grund verwenden wir in unserer Subklasse eine *TControlHookSubClass*-Instanz, und stellen die Botschaftsverarbeitung in Form der virtuellen Methode *WndProc* zu Verfügung. Unsere Aufgabe ist es nun, festzustellen, wann unser *TWinControl* neu gezeichnet werden soll. Das ist genau dann der Fall, wenn wir eine *WM_PAINT*-Nachricht erhalten und aus diesem Grund wird genau dann die Methode *PaintControl* unserer Subklasse aufgerufen, die zwei Aufgaben hat: Sie muss sowohl den Hintergrund des *TWinControl* zeichnen, als auch den Vordergrund.

Den Hintergrund zeichnen lassen

Den Hintergrund eines Controls zeichnen zu lassen bedeutet, so tautologisch es auch klingen mag, dass man alles zeichnen lässt, was sich hinter dem Control befindet.
Als erstes müssen wir das übergeordnete Fenster zeichnen lassen, also den *Parent* des Controls. Danach müssen wir all diejenigen dem *Parent* untergeordneten Controls zeichnen, die unterhalb unseres *TWinControl* liegen und somit durch unser *TWinControl* durchschimmern sollen.
Die Funktion, die diese Aufgabe übernehmen soll, habe ich *PaintControlBackground* genannt. Sie erwartet als Parameter das Control, dessen Hintergrund gezeichnet werden soll, einen *Display Context*, auf den gezeichnet werden soll, und optional, ob untergeordnete Controls überhaupt gezeichnet werden sollen.
Die Funktionsweise mag zwar kompliziert wirken, ist es aber eigentlich gar nicht. Sie verschiebt zuerst den Ursprung (*Origin*) des *Display Context* um die Position des Controls selbst. Das hat den erwünschten Effekt, dass am tatsächlichen Ursprung des *Display Context* genau derjenige Bereich des *Parent* erscheinen wird, der unterhalb des Controls liegt.
Als nächstes wird überprüft, ob der *Parent* unterhalb des Controls überhaupt sichtbar ist, oder durch seine eigenen, untergeordneten Controls komplett überdeckt ist. Wenn auch nur ein einziges Pixel zu sehen ist, wird er angewiesen, sich auf dem *Display Context* zu verewigen. Dies wird durch den Aufruf der Methode *PaintWindow* erreicht.
Nachdem der *Parent* gezeichnet wurde, müssen die einzelnen, untergeordneten Controls des *Parent*, die sich mit unserem *TWinControl* überschneiden, gezeichnet werden. Dies geschieht über eine Schleife, in welcher die jeweilige *PaintTo*-Methode des untergeordneten Controls aufgerufen wird, nachdem mittels der Funktion *IntersectControls* überprüft wurde, ob sich die beiden Controls überhaupt überschneiden.
Zurück zu unserer Subklasse. Sie verwendet die eben beschriebene Funktion, um den Hintergrund des *TWinControl* zeichnen zu lassen. Sie besitzt allerdings noch eine weitere, *Performance*-relevante Option. Wird das *TWinControl* oft gezeichnet, erscheint es natürlich nicht sinnvoll, jedes Mal wieder *PaintControlBackground* aufzurufen und den Hintergrund neu zeichnen zu lassen. Aus diesem Grund können wir über die Eigenschaft *BufferedBackground* einen Hintergrundpuffer, in Form einer *TBitmap*-Instanz, erzeugen lassen, welcher nur dann erneuert wird, wenn sich der Hintergrund auch wirklich geändert hat. Diese Option verbraucht natürlich mehr Arbeitsspeicher. Für *TWinControls* also, die nicht besonders oft neu gezeichnet werden, ist es also sinnvoller, diese Option zu deaktivieren. Wir bezeichnen die Methode, welche in unserer Subklasse die Aufgabe übernimmt, den Hintergrund auf einen *Display Context* zu zeichnen, mit *PaintBackground*. Diese Methode macht im Grunde nichts anderes, als die oben beschriebene Funktion *PaintControlBackground* aufzurufen oder gegebenenfalls aus dem gepufferten Hintergrund auf den *Display Context* zu zeichnen.

Der Zeichenvorgang des Controls

Das Zeichnen des Vordergrundes selbst erweist sich als nicht besonders kompliziert. Wir verwenden dazu die auch dafür vorgesehene Methode *PaintTo*, über die jedes *TControl* verfügt. Mithilfe dieser Methode ist es möglich, den Vordergrund auf einem angegebenen *Display Context* zeichnen zu lassen.
Wie schon erwähnt, soll unsere Subklasse über die Methode *PaintControl* verfügen, welche das *TWinControl* mit transparentem Hintergrund zeichnen soll. Dazu deaktiviert sie zuerst das *Caret*, sofern das *TWinControl* über eines

verfügt. Würde man dies nicht tun, können an der Stelle des *Caret* in der Folge unschöne Grafikfehler auftreten. Als nächstes bereitet man Windows mittels *BeginPaint* auf die sich anschließende Benutzung des *Display Context* vor. Die Zeichnung selbst ist dann recht simpel: Es wird über die vorher verfasste Methode *PaintBackground* der Hintergrund, und anschließend über *PaintTo* der Vordergrund gezeichnet.

In den meisten Fällen soll der Hintergrund natürlich direkt auf den *Display Context* unseres *TWinControl* gezeichnet werden soll. Allerdings wird unser *TWinControl* dadurch anfangen zu flimmern, während es gezeichnet wird. Da ein realer *Display Context* seine Änderungen sofort an den Bildschirm überträgt, und nicht erst, wenn die darauf stattfindenden Operationen abgeschlossen sind, hat sich Delphi für seine Controls die Option *DoubleBuffered* ausgedacht. Sie stellt sicher, dass das *TControl* auf einem virtuellen *Display Context* gezeichnet wird, und erst, wenn die Zeichnung komplett abgeschlossen ist, wird dieser virtuelle *Display Context* auf den realen übertragen.

Da allerdings wir jetzt die Aufgabe des Zeichnens durch das *Hooken* der *WindowProc* übernommen haben, müssen wir auch die Implementation der *DoubleBuffered*-Option übernehmen. Dazu wird eine *TBitmap*-Instanz erstellt, die, ähnlich wie der Hintergrundpuffer, einen virtuellen *Display Context* zu Verfügung stellt. Auf diesen wenden wir dann, wie oben beschrieben, *PaintBackground* und *PaintTo* an, und übertragen anschließend diesen virtuellen *Display Context* auf den realen unseres *TWinControl*.

Um den Zeichenvorgang abzuschließen, schließen wir den Vorgang mit *EndPaint* ab, und zeigen gegebenenfalls durch einen Aufruf von *ShowCaret* das zuvor verborgene *Caret* wieder an.

Den Vordergrund maskiert zeichnen

Da sich jedoch leider nicht alle *TWinControl*-Komponenten, wie ich schon zu Anfangs erwähnt habe, kooperativ verhalten, verläuft die Zeichnung des *TWinControl*, zumindest in manchen Fällen, etwas komplizierter. Wir sind bis jetzt stillschweigend davon ausgegangen, dass sich der Vordergrund, den wir mit *PaintTo* über den zuvor erstellten Hintergrund malen, zu diesem transparent verhält, das heißt: Selbst nicht versucht, einen Hintergrund beispielsweise über *FillRect*, zu zeichnen.

Es wird sich jedoch leider zeigen, dass genau dies in den meisten Fällen passiert. Die Folge ist, dass wir zwar unseren transparenten Hintergrund auf den *Display Context* übertragen haben, dieser jedoch komplett von *PaintTo* mit einer einheitlichen Farbe übermalt wird. Dies geschieht beispielsweise bei der *TButton*-Komponente.

Doch auch für dieses Problem gibt es eine Lösung, und zwar maskiertes Zeichnen. Es wird also eine bestimmte Farbe des Vordergrundes transparent gemacht. Diese Option wollen wir in unserer Subklasse über *MaskPaint* aktivieren lassen, und mittels *MaskColor* die Farbe spezifizieren lassen, die sich transparent verhalten soll. Bei einem *TButton* beispielsweise, würde es sich dabei um *clBtnFace* handeln.

Das maskierte Zeichnen läuft schließlich folgendermaßen ab: Es wird zunächst der Hintergrund auf den *Display Context* gezeichnet, und der Vordergrund auf einen temporären virtuellen *Context*. Anschließend wird die Maske für den Vordergrund erstellt: Dazu wird der temporäre Vordergrund kopiert, und mit der Farbe *MaskColor* maskiert. Das Ergebnis ist eine monochrome Maske, in der alle Pixel, die zuvor der Farbe *MaskPaint* waren nun weiß sind, und alle restlichen schwarz. Anschließend wird die Maske mittels des *AND*-Operators mit dem *Display Context*, auf den der Hintergrund gezeichnet wurde, verknüpft. Es sind damit nun all diejenigen Pixel, die vom Vordergrund übermalt werden sollen, schwarz geworden.

Jetzt wird die Maske invertiert: Alle Pixel die zuvor schwarz waren, werden weiß, und umgekehrt. Die invertierte Maske wird, ebenfalls mit dem *AND*-Operator, auf den temporär gezeichneten Vordergrund angewandt. Es sind analog zur vorherigen Zeichnung nun all diejenigen Pixel schwarz geworden, die zuvor der Farbe *MaskColor* entsprochen haben.

Der letzte Schritt ist jetzt, den Vordergrund mit Hintergrund zu verknüpfen. Dies erfolgt über den *OR*-Operator. Es werden dadurch alle durch die Maske geschwärzten Pixel des Hintergrundes durch die farbigen Pixel des Vordergrundes ersetzt. Die durch Maskierung transparent gewordenen Pixel des Vordergrundes werden nicht mehr übertragen.

Der Hintergrund im Wandel

Obwohl unser *TWinControl*, ob kooperativ oder nicht, jetzt schon wunderbar transparent gezeichnet wird, haben wir noch ein offenes Problem. Ändert sich etwas hinter unserem

TWinControl, beispielsweise ein dahinterliegendes *TLabel*, so bekommen wir davon leider nichts mit. Unser *Display Context* wird also nicht neu gezeichnet. Dieses Manko wird sofort sichtbar: Das Aussehen des *TLabel*, hat sich zwar verändert, jedoch scheint durch unser *TWinControl* immer noch das alte *TLabel* durch – denn unsere Transparenz ist ja keine echte Transparenz, sondern basiert auf dem Zeichnen des Hintergrundes. Und dieser wird von uns eben nicht neu gezeichnet. Wir müssen also mitbekommen, wann sich unser Hintergrund geändert hat, und wann wir somit unser *TWinControl* neu zeichnen müssen.

Dazu müssen wir erfahren, was sich hinter uns tut, und das bedeutet übersetzt, wir müssen alles, was sich hinter uns befindet, *hooken*. Da die Aufgabe des fensterweiten *Hooken* nicht nur singulär für das Transparenzproblem nützlich sein kann, schreiben wir uns eine weitere Hilfsklasse. Wir bezeichnen Sie mit *TFormMessageHook*. Diese funktioniert folgendermaßen: Sie erstellt für jedes *TControl*, dass sich auf dem Formular befindet, eine *TControlHookHelperClass*, sodass sämtliche Botschaften, die an die verschiedenen Controls versandt werden, von der *TFormMessageHook*-Hilfsklasse überwacht werden können.

Ferner verfügt diese Hilfsklasse über eine Ereignisliste, in die sich einzelne Klassen eintragen können. Es kann sich somit also unsere *TWinControl*-Subklasse über die Methode *RegisterWndProcHook* der Hilfsklasse *TFormMessageHook* registrieren lassen, und folgerichtig alle Nachrichten, die versandt werden, mithören. Treten dann Nachrichten, die auf eine Veränderung des Aussehens von Controls schließen lassen, die sich hinter unserem *TWinControl* befinden, auf, so muss unser Hintergrund neu gezeichnet werden.

Nachrichten dieser Art sind natürlich in erster Linie *WM_PAINT*, *WM_SIZE*, *WM_MOVE* und *WM_SETTEXT*, aber auch Nachrichten wie *WM_LBUTTONDOWN*. Unser *TWinControl* ist also jetzt in der Lage, transparent dargestellt zu werden, und mitzubekommen, wann sich sein Hintergrund zu ändern hat. Jetzt ist es unsere Aufgabe, auf die Eigenheiten im Speziellen einzugehen, also mit den einzelnen Problemen der unkooperativen *TWinControls* umzugehen.

Memo, Edit und Button

Exemplarisch möchte ich nun auf die Eigenheiten von *TMemo*, *TEdit* und *TButton* eingehen. Von der jeweiligen Komponente werden *TTransparent*-Versionen abgeleitet, also beispielsweise *TTransparentMemo*. Von unserer allgemeinen Subklasse werden ebenfalls angepasste Klassen gebildet, im Falle von *TMemo* bezeichnen wir die Subklasse mit *TEditTransparency*, weil sich das Memofeld bezüglich der Transparenzeigenschaften genau wie das Editfeld verhält. *TTransparentMemo* selbst erzeugt lediglich eine *TEditTransparency*-Instanz und veröffentlicht diese in einer *Published-Property*, sodass sie mithilfe des *Objektinspektors* bearbeitet werden kann.

Jetzt müssen wir uns in der neuen Subklasse *TEditTransparency* mit den Eigenheiten von Edit- bzw. Memofeldern befassen. Als erstes lässt sich bemerken, dass wir auf maskiertes Zeichnen verzichten können. An Edit- bzw. Memofelder werden nämlich die beiden Nachrichten *CN_CTLCOLOREDIT* und *CN_CTLCOLORSTATIC* versandt. Durch das *Hooken* und Verarbeiten dieser Nachrichten ist es uns möglich, den *Brush*, also den Pinsel, mit welchem der Hintergrund eines Edit- bzw. Memofeldes gemalt wird, zu bestimmen. Und wir setzen ihn an dieser Stelle natürlich auf *TRANSPARENT*, einem konstanten *Brush*, der eben gar nichts übermalt.

Damit sind wir fast fertig. Verloren geht uns unsere Transparenz allerdings teilweise, wenn sich der Inhalt des Edit- bzw. Memofeld ändert. Aus diesem Grunde fangen wir die *CN_COMMAND*-Nachricht ab, die versandt wird, wenn sich der Inhalt des Feldes ändert, und lassen es einfach durch unsere Methode *PaintControl* auf der Stelle neu zeichnen.

TButton jedoch verhält sich wesentlich unkooperativer. Es existiert zwar die Nachricht *CN_CTLCOLORBTN*, sie hat jedoch keine Auswirkungen auf normal erzeugte Buttons. Das bedeutet, wir können unseren *TButton* nicht dazu bringen, einen transparenten *Brush* zu verwenden. Es kommt also unsere *MaskPaint*-Eigenschaft zum Einsatz, wobei im Falle eines *TButton* die transparente Farbe, also *MaskColor*, immer *clBtnFace* entspricht. Doch damit leider nicht genug. Unser *TButton* wird jetzt zwar transparent gezeichnet, aber nur, wenn er neu gezeichnet werden soll, also durch ein Eintreffen der Nachricht *WM_PAINT*. Treffen Nachrichten wie *WM_LBUTTONUP*, *WM_LBUTTONDOWN* oder *WM_LBUTTONDBLCLK* ein, so zeichnet sich ein *TButton* automatisch selbst neu, ohne uns auf eine *WM_PAINT*-Nachricht reagieren zu lassen.

Um einen Workaround bemüht, lassen wir natürlich unsere Button-Subklasse auf diese oben

genannten Nachrichten reagieren, und den Button neu zeichnen. Nur entsteht dadurch ein kosmetisches Problem, nämlich ein Flimmern. Wie schon erwähnt, werden Veränderungen an einem realen *Display Context* sofort an den Bildschirm übertragen. Was also passiert, ist, dass sich beim Eintreffen einer der oben genannten Nachrichten der Button zunächst selbst zeichnet, und das natürlich nicht transparent. Erst anschließend wird er von uns wiederum transparent über unsere Methode *PaintControl* gezeichnet.
Um nun dieses Flimmern zu unterdrücken, bedienen wir uns eines unkonventionellen Tricks. Wir müssen es schaffen, den *Display Context* beim Eintreffen der oben genannten Nachrichten zu deaktivieren, bzw. die Darstellung auf dem Bildschirm zu unterbinden, um ihn danach wieder zu aktivieren, und selbst zu zeichnen. Diese Aktivierung und Deaktivierung erreichen wir dadurch, dass wir *Regionen* für unser *TWinControl* setzen lassen. Wir verwenden dazu die Funktion *SetWindowRgn*, die beispielsweise dafür zuständig ist, runde *TWinControls* zu erzeugen. Um also unseren *Display Context* zu deaktivieren, erzeugen wir für unser Fenster eine Region, die aus keinem einzigen Pixel besteht. Eine Region ohne Pixel besitzt zwar einen *Display Context*, er wird jedoch nicht auf dem Bildschirm dargestellt. Um den *Display Context* wieder zu aktivieren, stellen wir einfach die *Default-Region*, nämlich das Rechteck des *TWinControl*, wieder her.

Die Transparenz ist teuer erkauft

Nachdem nun ausführlich beschrieben wurde, wie Komponenten, auch wenn sie sich unkooperativ verhalten, Transparenz beigebracht werden kann, bleibt dennoch zu sagen, dass sich dadurch natürlich ein nicht unerheblicher Mehrverbrauch an Rechenzeit und Speicher, je nach Einstellung von *BufferedBackground*, ergibt. Im einen Fall wird bei jedem Zeichenvorgang eines *TWinControl* alles neu gezeichnet, was sich dahinter befindet, im anderen Fall muss zu dem *TWinControl* ein graphischer Puffer dauerhaft im Arbeitsspeicher gehalten werden.
Die machbar gewordene Transparenz ist zweifelsohne ein kosmetischer Fortschritt. Sie sollte jedoch aus den oben genannten Gründen so selten wie möglich eingesetzt werden. In vielen Fällen ist es wesentlich sinnvoller, die eigene Komponente entweder von *TGraphicControl* abzuleiten, oder den Hintergrund durch eine kleine Funktion graphisch dem echten Hintergrund nachzuempfinden.

Der Quelltext

Die in diesem Artikel beschriebene Technik findet sich natürlich auch in Form von Quellcode auf unserer Heft-CD. Die genannten Eigennamen und Bezüge finden sich im Quelltext wieder, sodass Sie die hier erläuterten Überlegungen gut nachvollziehen können. Exemplarisch implementiert habe ich die drei erwähnten Komponenten *TTransparentMemo*, *TTransparentEdit* und *TTransparentButton*. Um sie in Benutzung zu nehmen, kompilieren und registrieren Sie bitte das *Package „tbTransparent.dpk“*. Sie finden die drei Komponenten und den globalen *Formularhook*, *TFormMessageHook*, unter der Palettenseite *„Toolbox“*. Die Transparenz der drei Komponenten kann im *Objektinspektor* über die Subklassen-Eigenschaft *Transparency* aktiviert werden.