sehr schnelle, einfache und ressourcenschonende Grafik

Seit Jahren leben die meisten von uns mit dem Problem, das Rechentechnik viel zu schnell entsorgt wird, da die Programm einfach zu langsam laufen. Wie kann man einen betagten Rechner eigentlich besser recyceln als ihn mit vernünftiger Software weiter sinnvoll zu nutzen ?

Es ist immer wieder erstaunlich wie genial einige Programmierer es verstehen eine Mücke zum Elefanten aufzublasen um den Prozessor so richtig arbeiten zulassen. Sicher kein schlechtes Geschäft, wenn man als Programmierer das Programm so aufbläht, das man gleich noch einen neuen Rechner mit verkaufen kann. Wo würde die Wirtschaft auch hinkommen, wenn nicht ständig sinnlos neu gekauft werden würde.
Ich bin mir deshalb auch bewusst, das mancher Programmierer dies als Kriegserklärung auffassen wird. Ich kann gut damit leben, wenn jemand seine Programme lieber weiter mit angezogener Handbremse laufen lassen möchte.

Wer aber bei Grafikanwendungen, auch mit schmalbrüstiger oder betagter Technik beachtenswerte Geschwindigkeiten erreichen möchte, dem biete ich hier die Möglichkeit, richtig den Turbo anzusetzen. Je älter und langsamer die Hardware ist, um so deutlicher wird man die Unterschiede sehen.

Grafikprogrammierung auf dem PC könnte so einfach sein - wenn man denn nur wollte.

Ich habe die letzten Jahrzehnte meines Berufslebens als Softwareentwickler gearbeitet. Gelieferte Komponenten habe ich nie als gottgegeben und perfekt betrachtet. Vielmehr habe ich die Quelltexte ausgewertet und je nach dem, wie schlecht sie waren, ausgemistet, erweitert oder gleich komplett neu geschrieben. Das war bei Delphi so und ist in Lazarus nicht anders!

Vor Jahren stand ich vor der Aufgabe für einen Raspberry eine Anwendung zu entwickeln, mit vielen grafischen Elementen, die sich ganz schnell verändern mussten. Außerdem musste extrem schnell auf Kommunikationsanfragen reagiert werden. Mit den damals vorhandenen Komponenten war das nicht annähernd realisierbar. Durch einen ganz neuen Ansatz in der Grafik konnte die Geschwindigkeit deutlich verbessert und Prozessorauslastung so drastisch gesenkt werden, das nicht nur die Aufgabe gelöst wurde, sondern sogar der Dauerbetrieb in einem geschlossenes Gehäuse, ohne Lüftungslöcher und ohne Kühlkörper möglich wurde! Super schnell aber eben leider nur für den Raspberry verwendbar.

Zur Zeit arbeite ich nur auf einem PC unter LINUX 64 BIT (Deepin 20.2). Deshalb entstand zunächst eine überarbeitete Variante für diese Hard– und Software. Eine optimierte Version für den Raspberry könnte folgen.
Theoretisch sollte die Software auf allen Betriebssystemen laufen, auf denen auch Lazarus lauffähig ist. Für Systeme, bei denen der Assemblercode nicht passt, gibt es die gleichen Funktionen noch einmal als Lazarus - Quelltext. Diese müssen nur ausgetauscht werden.
Gegebenenfalls müssen die Funktionen für das Lesen und Schreiben der Datei angepasst und meine effizienten Funktionen für Zeichenketten durch Strings ersetzt werden.
Nach dem WINDOWS10 beim letzten Update, aus einer Bierlaune heraus, einfach mal mein komplettes Benutzerverzeichnis unwiederbringlich gelöscht hat, arbeite ich nur noch unter Linux, was ich noch keinen Tag bereut habe. Damit kann ich aber weder das ganze unter WINDOWS testen noch die eventuell notwendige Anpassungen einarbeiten. Mit dem unwiderruflichen Ende von WINDOWS gibt es für mich leider auch kein Delphi mehr. Mit geringen Anpassungen sollte das Prinzip aber auch unter Delphi nutzbar sein. Erweiterungen für WINDOWS oder Delphi werden ausdrücklich erlaubt mit der Bitte um Rückmeldung!

Mindestbestandteile für Computergrafik:

> Man benötigt zwei Variablen für die Breite und Höhe der Grafik und einen Speicherbereich für die Pixel, der über einen Pointer erreichbar ist. Zusätzliche Hilfsvariablen, wie für die Gesamtanzahl der Pixel, der Größe des Puffers usw. sind nicht unbedingt nötig, aber sinnvoll, da diese feststehenden Werte nicht jedes mal neu berechnet werden müssen.
>
> und sonst absolut gar nichts! keine Handles, Gerätecontexte, keinen Canvas und was man sich sonst alles als überflüssigen Ballast hat einfallen lassen!

Notwendige Software:

> Getmem() und Freemem() um den Speicherbereich zu erzeugen und wieder freizugeben.
> Eine Schnittstelle zu Betriebssystem um die Grafik auszugeben.
> verschiedene Grafikfunktionen um Linien, Quadrate oder Kreise zu zeichnen, Bereiche zu kopieren oder einzelne Pixel zu setzen. Egal was auch immer getan werden muss, es wird ausschließlich durch Kopieren oder Setzen der Pixel in dem Speicherbereich erreicht.

Nützliche Tools:

Jeweils eine Funktion die Grafikdateien einliest und als Grafikdatei speichert

Zum Speicherbereich:

Es gibt viele verschiedene Varianten um die Pixel zu speichern. Es ist aber völlig ausreichend wenn man nur eine Variante benutzt. Alle Betriebssysteme die ich kenne, benutzen inzwischen für den Framebuffer 32 Bit Datenbreite. Also wähle ich auch nur dieses eine Format. Andere Formate werden bei der Verwendung einfach in dieses Bitformat gewandelt.
Als Pointer wähle ich PColorSatz um immer auf ein komplettes Pixel zuzugreifen. Bytepointer erfordern mehr unnötige Zugriffe und moderne Prozessoren können mit 32 Bit Werten viel effizienter arbeiten als mit einzelnen Bytes.

Betriebssystem bedingte Unterschiede:

Viele begründen den aufgeblasenen Quellkode von vorhandener Grafiksoftware damit, das damit angeblich der gleiche Quelltext für alle Betriebssysteme verwendet werden kann, was natürlich Unfug ist. Die betriebssystemabhängigen Unterschiede sind viel kleiner als man vermuten würde und sind ganz einfach zu beheben. Für die Grafik selbst und die Bearbeitung bestehen überhaupt keine zwingenden Unterschiede.

Windows und Linux belegen die Farbanteile der Pixel im Speicher unterschiedlich.
Zwei verschiedene Varianten von ColorSatz, könnten durch Kompileroptionen einfach ausgetauscht, werden und damit ist dieses Problem für ein und alle mal erledigt. Für die Verwendung von Farben als Parameter sollte man gleich Farbkonstanten mit richtiger Farbdarstellung entwenden, was auch einfach durch Kompileroptionen erreicht werden kann. Nur wenn eine Anpassung wirklich nötig ist, kann man durch Kompileroptionen eine Konvertierungsfunktion einblenden.

Für jedes Betriebssystem kann man außerdem eine Funktion mit gleichen Namen und gleichen Parametern, schreiben die die jeweilige Schnittstelle des Betriebssystems versorgt. Durch Kompileroptionen wird dann jeweils nur die richtige Funktion eingeblendet, womit auch die Nutzung der Direktausgabe für den Raspberry möglich wird. Problem erledigt und Kapitel abgeschlossen.

Grafikdateiformate:
Es gibt unzählige Formate in denen man Grafiken auf Datenträger ablegen kann. Leider enthalten diese Dateien viele Informationen, die ich nicht benötige und das Einlesen unnötig behindern. Auch sind die Daten teilweise in einer ungünstiger Reihenfolge abgelegt oder enthalten Auffüllbytes am Zeilenende, was die Adressenberechnung unnötig belastet.
JPG - Dateien sind komprimiert und gut, wenn man viele Urlaubsbilder auf einem Stick speichern möchte, aber für unsere Anwendung sehr ungünstig. Je nach eingestellter Komprimierung kann man viel Speicherplatz sparen, womit natürlich auch die Qualität verloren geht. Die kürzeren Dateien benötigen zwar weniger Zeit zum einlesen. Diese Zeit setzt man aber mehrfach zu, da die Daten jedes mal ja erst dekomprimiert werden müssen. Unnötiger Aufwand, den man sich eigentlich gar nicht antun möchte.

Deshalb habe ich ein optimiertes Speicherformat entwickelt, das nur die notwendigen Informationen in einer günstigen Reihenfolge enthält und sich damit schneller einlesen oder speichern lässt. Grafik-Dateien die nur einmal benötigt werden, lassen sich natürlich im Originalformat einlesen und werden automatisch konvertiert. Farbinformationen werden unabhängig vom Betriebssystem richtig übernommen. Dateien, die bei jedem Programmstart benötigt werden, sollten einmalig im Originalformat eingelesen und für die Zukunft dann im optimierten Format gespeichert werden. Das beschleunigt jeden weiteren Programmlauf deutlich.

Grafikdateien auf Bildschirmformate skalieren:
Als ich meine Arbeit begann gab es eigentlich nur ein Bildschirmformat. 640 x 480 Pixel und schwarz weiß. Und dann kamen immer mehr neue Formate dazu. Ganz wild wurde es dann als die LCD-Bildschirme aufkamen. Damit war es nicht mehr sinnvoll für jedes Format angepasste Grafikdateien zu verwenden. Die Grafik musste auf das Bildschirmformat skaliert werden, was rechenintensiv war und Zeit benötigte. Aber diese Zeit scheint vorbei zu sein. Zur Zeit werden neue Monitore fast nur noch im Format 1920 x 1080 Pixel hergestellt. Deshalb sollte die Grafik gleich in der Größe gespeichert werden, in der sie später benötigt wird. Werden wirklich mehrere Ausgabeformate benötigt, sollte für jedes Format ein Bilderverzeichnis erzeugt werden. Über das Setzen des richtigen Pfades wird dann jeweils die Datei in der richtigen Größe eingelesen.

Der Standartprogrammierer zieht z.B. ein TEdit auf das Formular, ein Klickereignis dazu und fertig – Geld verdient. Dagegen ist erst mal nichts einzuwenden. Aber wirklich schön ist etwas anderes. Wenn ich schöne Bildschirmansichten haben möchte, dann suche ich mir schöne Buttons aus, speichere diese in einer Datei,

in meinem optimierten Format und kopiere die Grafik, dann zur Laufzeit des Programms, einfach an die gewünschte Stelle meines Hintergrundbildes. Zum Schluss wird mein Button mit der gewünschte Funktion in der ausgewählten Sprache beschriftet. Das klingt erst einmal umständlicher. Damit hat man aber auch alle künstlerischen Freiheiten und erhält einen Bildschirm, der eher an ein Spiel erinnert als an den typischen Windowsstiehl. Ein schöner Nebeneffekt, es erfordert viel, viel weniger Rechenzeit, meinen schönen Button in das Hintergrundbild zu kopieren, als den Standartbutton vom Betriebssystem neu erzeugen und neu zeichnen zu lassen. Auch die Mouseereignisse laufen schneller ab, da nicht alle Komponenten des Formulars der Reihe nach abgefragt werden müssen, ob die Koordinaten auf der Komponente liegen. Auf Grund des x- oder y– Wertes kann man garantiert einige Bereiche von einer genauen Prüfung ausschließen.

Auch über den Sinn der Pixel in einer Grafikdatei können wir uns Gedanken machen. In einem Bild, das ich als Hintergrundbild verwenden möchte, wird natürlich jedes Pixel benötigt.
Wenn ich aber einen Button in das Hintergrundbild kopiere, sieht das anders aus. Wenn wir das Bild eines Buttons (ohne Schrift) genau anschauen, stellen wir fest, das nützliche Informationen eigentlich nur in den 4 Ecken stecken. Alle Lücken haben Farbwerte, die man durch einfaches Kopieren der Spalten- oder Zeilenwerte füllen kann. Folglich könnte man die 4 Ecken mit einem Grafikprogramm zusammenschieben und damit die Arbeitsfläche und die Datei so weit verkleinern, das nur noch sinnvolle Information enthalten ist. Beim Zeichnen auf den Bildschirm werden dann zunächst nur die 4 Ecken an die richtige Stelle kopiert. Dann werden die Lücken durch kopieren gefüllt. Mit ein und der selben Datei lassen sich so unterschiedlich große Ziele ohne Skalierung erreichen. Es entstehen eben nur unterschiedlich große Lücken, die gefüllt werden müssen. Die Bitmaps benötigen deutlich weniger Platz auf der Platte, was sich sich positiv bei der Installation bemerkbar macht. Die kürzeren Quelldateien benötigen kürzere Downloadzeiten. Wenn diese Bitmaps im Arbeitsspeicher gehalten werden, wird dieser weniger belastet und die eine oder andere Auslagerung des Arbeitsspeichers wird vermieden, was wiederum Rechenzeit einspart.
Die Funktion „CopyEckenToTAroDib“ erledigt diese Aufgabe automatisch. Sie müssen nur die Position und die gewünschte Größe angeben.

Schriften in einer Grafik ausgeben.
Wenn man über ein Bild einen Text setz ist das einfach nachvollziehbar. Aber was viele nicht bedenken. Bei eine Textausgabe in einen Label, die Beschriftung eines Buttons oder die Eingabe in ein Editfeld wird Text in die Zeichenfläche des Formulars „gemalt“
Und dieses Malen der Zeichen in Schönschrift ist extrem aufwendig und benötigt viel Rechenzeit. Es muss die entsprechende Trueschriftdatei geöffnet werden. Die entsprechende Position des Zeichens muss gefunden werden. Jedes Zeichen hat mehrere Daten, nach denen mit einer mathematischen Funktion das eigentliche Zeichen dann gezeichnet wird. Vorher muss aber noch auf die gewünschte Zeichengröße skaliert werden. Auch fette Schrift und italic müssen eingearbeitet werden.
So wie die Kurve verläuft müssen dann die Pixel gesetzt werden. Da die Kurve fast nie nur genau den Mittelpunkt des Pixels trifft, müssen die angrenzenden Pixel farblich angepasst werden. Und dann muss auch noch der Abstand zum nächsten Zeichen berechnet werden. Schönheit hat eben ihren Preis.

Aber auch dafür habe ich eine Lösung gefunden, wie dies in gleicher Qualität viel einfacher und schneller umgesetzt werden kann. Dazu verwende ich ein eigenes Format für meine Fontdateien. Wenn ein Font in einer bestimmten Größe das allererste mal benötigt wird, dann besteht die Datei natürlich noch nicht, und wird automatisch erzeugt und gespeichert. Und das geschieht meistens schon während des Programmierens. Bei jedem weiteren Programmstart ist die Datei dann vorhanden und damit sofort verfügbar. Die Dateien benutzen den Bereich von 128 bis 255 für Sonderzeichen. Viel benötigte Zeichen aus anderen Schriften, UTF-8 Umlaute, aber auch eigene Zeichen lassen sich so unkompliziert und vor allem als 8-Bit Zeichen erreichen.
Soll Text ausgegeben werden, wird als erstes geprüft ob die richtige Fontdatei bereits geöffnet wurde.
Wenn noch nicht wird die Datei gesucht und geöffnet.
Wenn sie noch nicht existiert, wird die Datei vorher einmalig erzeugt.
Einmal geöffnete Fontdateien werden erst am Programmende geschlossen und stehen ständig zur Verfügung.

Aus der meiner Fontdatei wird für jedes Zeichen ein Record mit allen notwendigen Informationen für das gewünschte Zeichen eingelesen. Danach werden die Alphawerte für die beteiligten Pixel aus der Datei gelesen und mit diesen, die Textfarbe mit dem Alphawert eingeblendet. Abschließend mit dem Zeichenabstand die Anfangsposition für das nächste Zeichen gesetzt und fertig. Und genau so einfach wie das hier klingt wurde es wirklich realisiert!

BGRABitmap:

Da ich zielorientiert arbeite, habe ich natürlich nur die Grafikfunktionen programmiert, die ich bisher auch wirklich für meine Programme benötigt habe. Mir ist vollkommen klar, das andere User auch Funktionen benötigen, die ich noch nicht benötigt und deshalb noch nicht vorgesehen habe. Und was gerade nötig ist hängt vor allem von der Aufgabenstellung ab. Die Lösung bietet BGRABitmap.

BGRABitmap ist relativ neu und wesentlich besser als TBitmap, beinhaltet aber immer noch viele Dinge, die es unnötig ausbremsen. Deshalb können einige Aufgaben mit meiner Grafik auch wesentlich schneller erledigt werden.
Das soll aber keinesfalls eine Kritik BGRABitmap werden. BGRABitmap kann viele Dinge die ich nicht oder noch nicht vorgesehen habe! Aber durch die Kombination mit BGRABitmap kann mit meiner Grafik der volle Funktionsumfang erreicht werden.

BGRABitmap hat erfreulich viele Gemeinsamkeiten mit meiner Lösung. Es verfügt, ebenfalls über einen Pointer, über den der Grafikbereich zugänglich ist. Der Pointer verändert sich nicht ständig durch sinnloses kopieren wie in TBitmap und der Grafikbereich ist gleich organisiert wie bei mir. Außerdem verfügt BGRABitmap über sehr viele Grafikfunktionen, die kaum Wünsche offen lassen.

Über eine Schnittstelle zu BGRABitmap ist ein unkomplizierter und schneller Datenaustausch möglich. Damit ist es möglich Funktionen, die ich bisher nicht benötigt habe, durch BGRABitmap zeichnen zu lassen. Das ist viel vernünftiger, als das Fahrrad unbedingt noch einmal neu zu erfinden.

Dazu erzeugrt eine Funktion von mir, ein BGRABitmap mit der gleichen Größe und gibt das BGRABitmap als Result zurück.
Meine Grafik kann dann einfach in den Bereiche des BGRABitmap kopiert werden.
Auf dem BGRABitmap kann man dann alle Funktionen nutzen, die BGRABitmap bietet.
Die bearbeitete Grafik kann man einfach wieder zurückholen.
Ein guter Kompromiss von Aufwand und Nutzen, um seltener gebrauchte Grafikfunktionen einzubinden.

Um bmp-Dateie schnell einzulesen habe ich eine eigene Funktion geschrieben. BGRABitmap bietet aber die Möglichkeiten auch weitere Grafikformate zu öffnen. Also biete ich auch eine Funktion über BGRABitmap, diese Formate einzulesen, statt diese Funktionen neu zu schreiben.

Zur Schnittstelle zum Betriebssystem:

> Jedes Betriebssystem erwartet, das die Grafikdaten für eine Ausgabe auf dem Bildschirm, in eine bestimmte Schnittstelle übergeben werden. Die Schnittstellen sind natürlich vollkommen unterschiedlich aufgebaut und erwarten auch andere Parameter. Da es mir noch nicht gelungen ist in LINUX 64 Bit die Direktausgabe wie beim Raspberry zu realisieren verwende ich (noch) die gdk2 – Schnittstelle.

Verbesserungsmöglichkeiten: Es gibt immer die Möglichkeit, die Geschwindigkeit weiter zu steigern. Meine Grafik wurde auch im laufe der Jahre immer weiter optimiert. Assemblerkode ist viel effizienter als der Maschinencode des Compilers. Damit entfällt aber die Möglichkeit das Programm universell zu nutzen. Deshalb sind meine Assemblerfunktionen auf das Wesentliche beschränkt. Für diese Funktionen gibt es jeweils noch einmal die gleiche Funktion in Lazarus als Kommentar. Wenn Assemblercode nicht passt, kann man die Ersatzfunktionen einblenden und den Assemblercode als Kommentar ausblenden. Für Intel 32 Bit dürfte die Umstellung von den 64 Bit Registern auf die gleichen 32 Bit Register nicht zu schwierig sein. Für den Raspberry wird es komplizierter, da dort ganz andere Register vorhanden sind.
Wenn Interesse besteht und ich die Zeit finde, könnte ich meinen Raspberry wieder aktivieren. Wenn mit den aktuellen Versionen die Direktausgabe noch funktioniert, könnte ich eine optimierte Version für den Raspberry bereitstellen.

Ein gesondertes Dokument informiert, informiert über Eigenheiten meiner Programmierung: