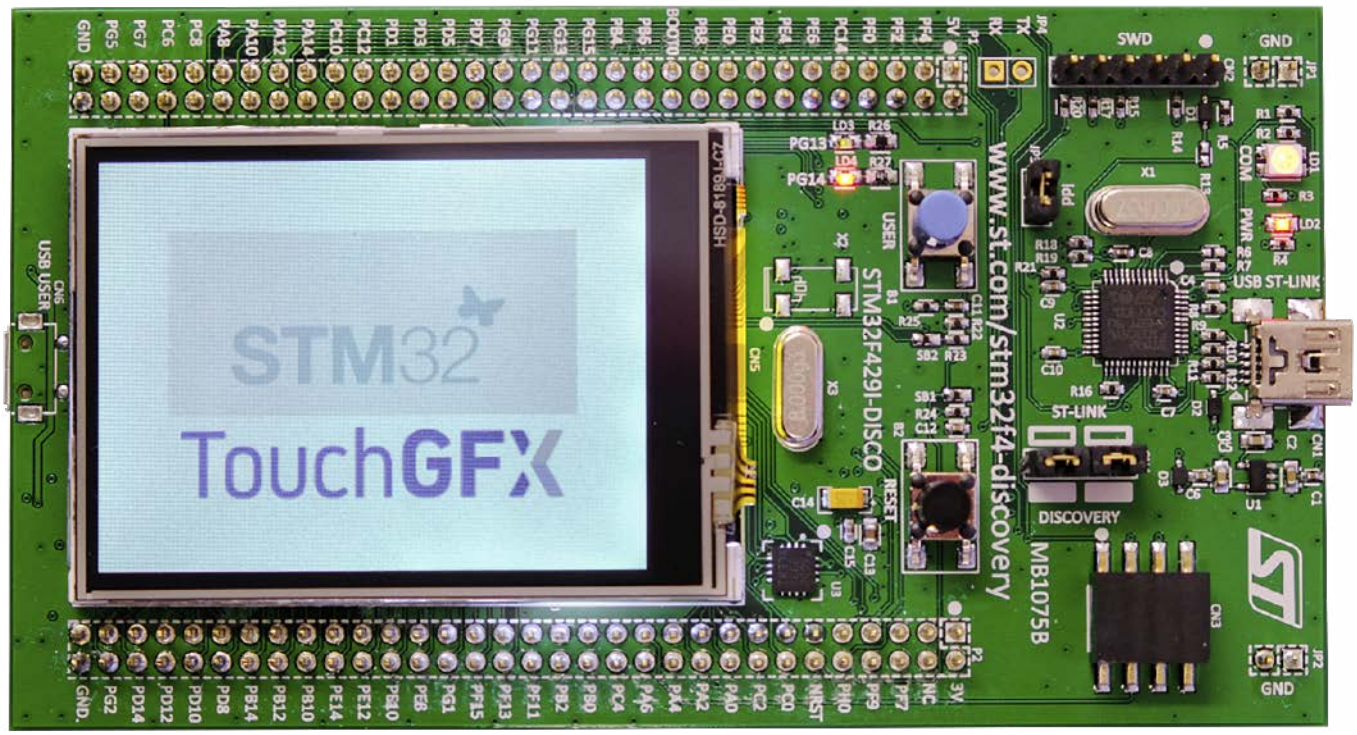


FTP Wszystkie poprzednie odcinki kursu są dostępne na stronie www.media.avt.pl

Projektowanie interfejsów graficznych z użyciem TouchGFX (3)

W poprzedniej części udowodniliśmy, że jeżeli mamy moduł ewaluacyjny całkowicie wspierany przez TouchGFX Designer, to w relatywnie krótkim czasie jesteśmy w stanie zaprojektować i wykonać prosty interfejs z ekranem dotykowym. Chociaż TouchGFX wykonuje za nas olbrzymią pracę, to programista musi uzupełniać wygenerowany kod o swoje procedury. Zaprezentujemy narzędzia ułatwiające tę pracę.

Pokazany wcześniej przykład jest świetną zachętą do dalszych eksperymentów. Jednak trzeba sobie zdawać sprawę, że zastosowany moduł ma zbyt małe zasoby, głównie pamięć Flash i pamięć RAM do poważniejszych zastosowań. Ponadto przyjęty sposób edycji plików w zewnętrznym edytorze nadaje się do bardzo prostych testów. Chociaż TouhGFX znaczną pracę wykonuje za nas to komplet narzędzi powinien zawierać:

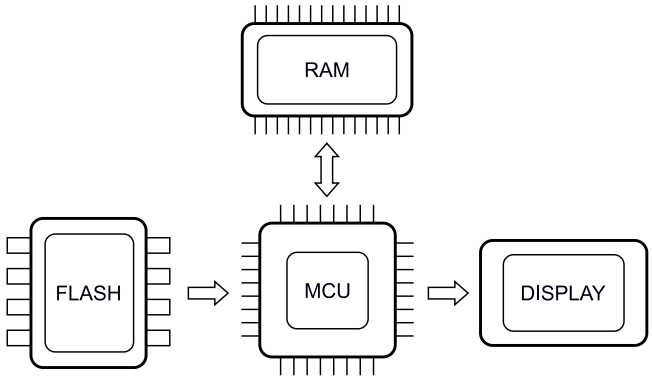
- TouchGFX – do kreowania graficznego interfejsu i projektowania podstawowych interakcji wyzwalanych przez elementy graficzne ekranów interfejsów. Przykładową interakcją może być naciskanie przycisków i modyfikacja licznika opisywana w przykładzie powyżej. Kod wynikowy TouchGFX ma postać projektu dla IDE EWARM firmy IAR, MDK-ARM firmy Keil i STM32CubeIDE;
- Środowisko programistyczne IDE – w którym otworzymy wygenerowany projekt i będziemy mogli go modyfikować, kompilować i programować pamięci modułu ewaluacyjnego. Ja używałem firmowego, bezpłatnego STM32CubeIDE.
- Konfigurator STM32CubeMX – służy do konfigurowania układów peryferyjnych i middleware (na przykład FreeRTOS).

Kolejną kwestią wartą rozważenia jest podział zadań między usługami wykonywanymi przez panel operatorski HMI, a właściwymi usługami wykonywanymi przez układ wbudowany. Moduły ewaluacyjne przeznaczone do roli HMI mają stosunkowo duże zasoby: szybki mikrokontroler z dużą pamięcią wbudowaną Flash i RAM, oraz zewnętrzną pamięć Flash i RAM połączoną przez interfejs równoległy lub szybki interfejs szeregowy, z mikrokontrolerem. Wydawałoby się, że takie konfiguracje sprzętowe bez problemu poradzą sobie z większością różnych zadań. O tym czy tak jest będą decydowały z jednej strony wymagania algorytmów HMI, a z drugiej strony wymagania algorytmów układu wbudowanego.

Realizacja atrakcyjnego HMI ma duże wymagania sprzętowe. Wiadomo o tym chociażby ze świata komputerów PC. Jeżeli chcemy na nich uruchamiać skomplikowane zadania graficzne to niezbędny jest bardzo wydajny specjalizowany układ nazywany potocznie kartą graficzną. Karta graficzna musi mieć bardzo szybki wielordzeniowy procesor współpracujący z bardzo szybką, dużą pamięcią RAM. Szczególnie wymagające gry komputerowe z grafiką 3D nie będą działały płynnie bez odpowiedniej karty graficznej, ale też szybkiego wielordzeniowego procesora komputera. Taka sama sytuacja jest w smartfonach. Bardzo szybkie wielordzeniowe procesory są tam wspomagane

przez specjalizowane układy graficzne. Dzięki temu, przynajmniej w telefonach z wyższej półki, działanie interfejsów graficznych HMI jest płynne mimo zaawansowanych funkcji graficznych.

Podobnie, w odpowiedniej skali będzie w naszym przypadku. Elementy animacji, zmiany dużej ilości elementów graficznych w krótkim czasie powodują, że program będzie zajmował większość czasu procesora, a w skrajnych przypadkach zajmie go całkowicie. Musimy pamiętać, że mikrokontrolery z rdzeniami Cortex lub podobnymi używane do układów wbudowanych mimo ciągłego postępu i związanej z tym coraz większej wydajności nie mogą się równać pod tym względem do mikroprocesorów wspieranych przez układy graficzne stosowanych w smartfonach. Z drugiej strony graficzne interfejsy HMI oparte o coraz bardziej rozbudowane elementy graficzne stają się powoli standardem w wielu aplikacjach wbudowanych. Dlatego w mikrokontrolery wyposażone w szybkie rdzenie i duże pamięci zaczęto wbudowywać sprzętowe akceleratory grafiki będące specjalizowanymi kanałami DMA wspomagającymi szybki transfer danych z pamięci do wyświetlacza bez zajmowania czasu procesora.



Rysunek 1. Główne elementy grafiki HMI

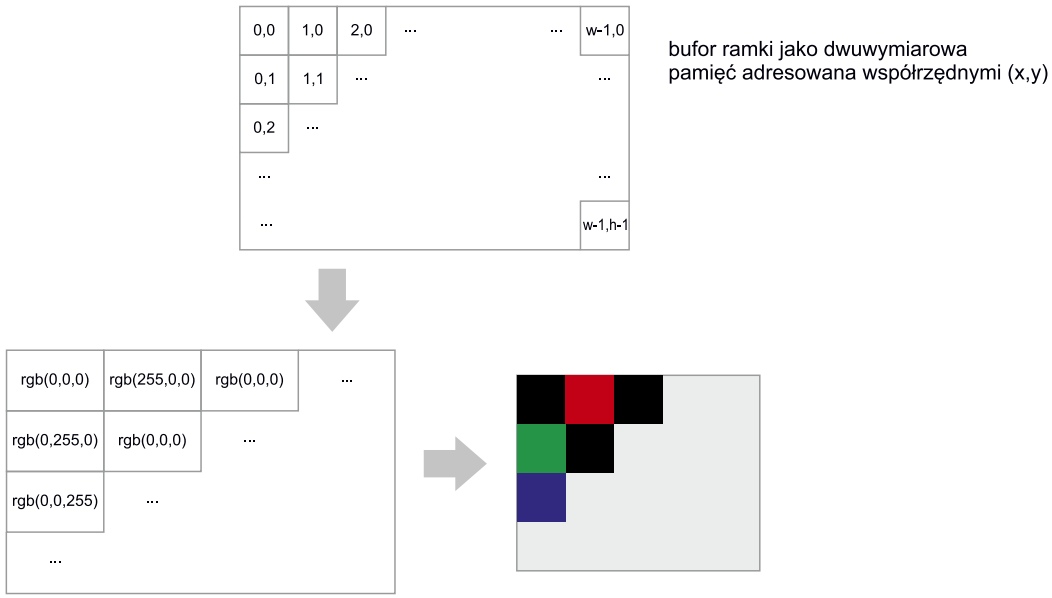
Silnik TouchGFX

Żeby można było określić jakie obciążenie procesora stanowią operacje graficzne interfejsu HMI i w konsekwencji mieć podstawę do jego prawidłowego zaprojektowania trzeba wiedzieć, przynajmniej ogólnie, jak działa silnik grafiki TouchGFX. Na **rysunku 1** zostały pokazane główne elementy potrzebne do zaimplementowania graficznego interfejsu HMI. Jednostka MCU spełnia podstawową rolę: odczytuje obrazy zapisane w pamięci Flash i po koniecznych modyfikacjach przesyła je do pamięci RAM. Musi wykonywać wiele czasochłonnych obliczeń na przykład potrzebne do wyliczania wynikowego koloru w łączeniu półprzeźroczystym dwu kolorów, czy renderowania pikseli. MCU wykonuje tez transfer danych z pamięci RAM do wyświetlacza, o ile nie ma wbudowanego akceleratora grafiki. W mikrokontrolery rodziny STM32 mogą być wbudowane układy peryferyjne znacząco wspomagające implementację interfejsów graficznych:

- LTDC – układ peryferyjny pozwalający na bezpośrednie sterownie wyświetlaczy wysokiej rozdzielczości bez udziału procesora. Może uzyskiwać autonomiczny niezależny dostęp do pamięci wewnętrznej lub zewnętrznej w celu uzyskiwania danych opisujących piksele wyświetlacza;
- Chrom ART – wyspecjalizowany układ DMA przeznaczony do obróbki obrazu. Realizuje kilka prostych, ale czasochłonnych operacji graficznych: wypełnianie bufora kolorem, kopiowanie i mieszanie obrazów, konwersja formatu pikseli itp.;
- Chrom GRC – graficzny moduł zarządzania pamięcią w celu optymalizacji użycia pamięci zgodnie z wyświetlanym kształtem. Zmniejsza użycie pamięci, bo powoduje, że są w niej przechowywane tylko widoczne piksele (w widocznej warstwie).

Pamięć RAM przechowuje bufor ramki (jeden lub więcej) zawierający wyliczony wynikowy obraz do wyświetlenia. W cyklach odświeżania bufor ramki jest przesyłany do wyświetlacza i ponownie wyliczany i zapisywany. Bufor ramki może być zależnie od potrzeb i możliwości zapisany w wewnętrznej lub zewnętrznej pamięci RAM. Zewnętrzne pamięci RAM wyposaża się w interfejsy równoległe FMC/FSMC lub szeregowe: SPI, QuadSPI, Dual QuadSPI, OctoSPI/Hyper Bus itp.

W pamięci Flash umieszczane są obrazy (bitmapy), czcionki teksty itp. Te elementy zajmują sporo miejsca i dlatego w konfiguracjach sprzętowych przeznaczonych dla aplikacji graficznych stosuje się zewnętrzne pamięci Flash. Żeby znacząco przyspieszyć działanie mapuje się pamięć Flash w pamięci RAM. Jeżeli nie jest to możliwe, to silnik grafiki może przenieść potrzebne elementy statyczne do pamięci RAM i stamtąd je odczytać.

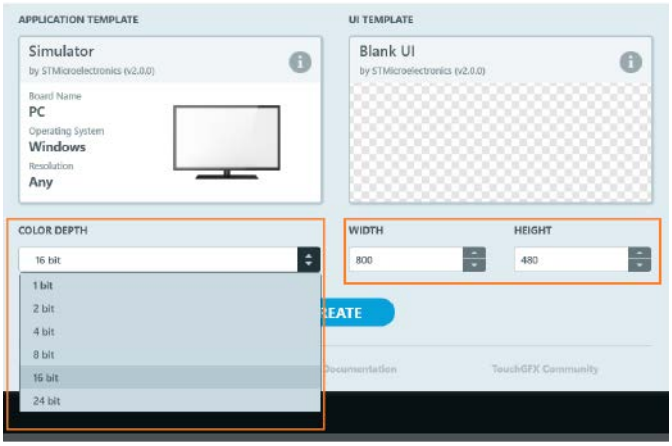


Rysunek 2. Idea bufora ramki

Bufor ramki

Bufor ramki to część pamięci zawierająca kolejny obraz, który ma być pokazany na wyświetlaczu. Jego zawartość jest aktualizowana przez silnik graficzny. Zawartość bufora ramki jest po przesłaniu ostatecznie widoczna na ekranie wyświetlacza. Dlatego wygodnie jest jego zawartość adresować współrzędnymi (x, y). Na **rysunku 2** został pokazany bufor ramki adresowany współrzędnymi x, y.

Rozmiar bufora jest najczęściej identyczny z rozmiarem wyświetlacza podawanym w pikselach. Każdy z pikseli na rysunku 2 jest opisywany 24 bitową liczbą (głębią kolorów): każda składowa koloru RGB jest zakodowana na 8 bitach. W systemach graficznych istnieje ograniczona liczba możliwych kolorów, które mogą być reprezentowane, używane i wyświetlane. Dotyczy to również aplikacji TouchGFX. Liczba możliwych kolorów pikseli aplikacji ma wpływ na wyglądu



Rysunek 3. Wybór głębi bitowej i rozmiaru wyświetlacza

tego, co widać na wyświetlaczu, ale też na zużycie pamięci narzucone przez bufor ramki i ogólną wydajność interfejsu graficznego. Im większa głębia koloru tym lepiej wygląda to co wyświetlamy na ekranie pod warunkiem, że wyświetlacz potrafi to wyświetlić. Jednak większa głębia to z drugiej strony wymagana większa pojemność bufora ramki i większe obciążenie procesora w czasie obliczeń wykonywanych przez silnik grafiki.

TouchGFX ma wbudowaną obsługę następujących głębi kolorów (podawanych w bpp czyli bitów na piksel):

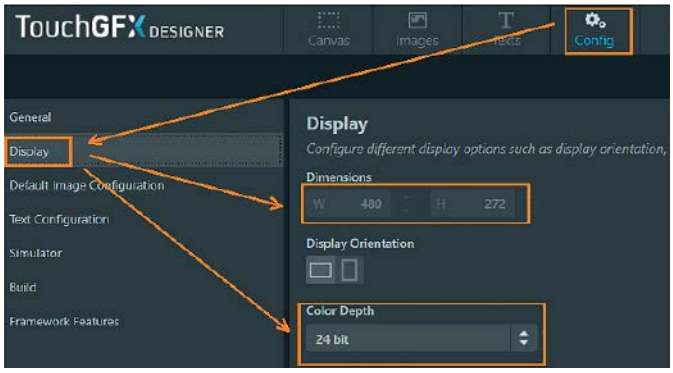
- 32 bpp – 16777216 kolorów i odpowiadające im wartości krycia
- 24 bpp – 16777216 kolorów
- 16 bpp – 65536 kolorów
- 6 bpp – 32 kolory
- 4 bpp – 16 kolorów w skali szarości
- 2 bpp – 4 kolory w skali szarości
- 1 bpp – 2 kolory w skali szarości

Głębia 32 bpp ma dodatkowy 8 bitowy składnik alfa opisujący krycie koloru. Kolory z kryciem nazywane są kolorami RGBA. Całkowicie nieprzezroczysty czarny kolor to (0,0,0,255), nieco przezroczysty czerwony to (255,0,0,128) i tak dalej.

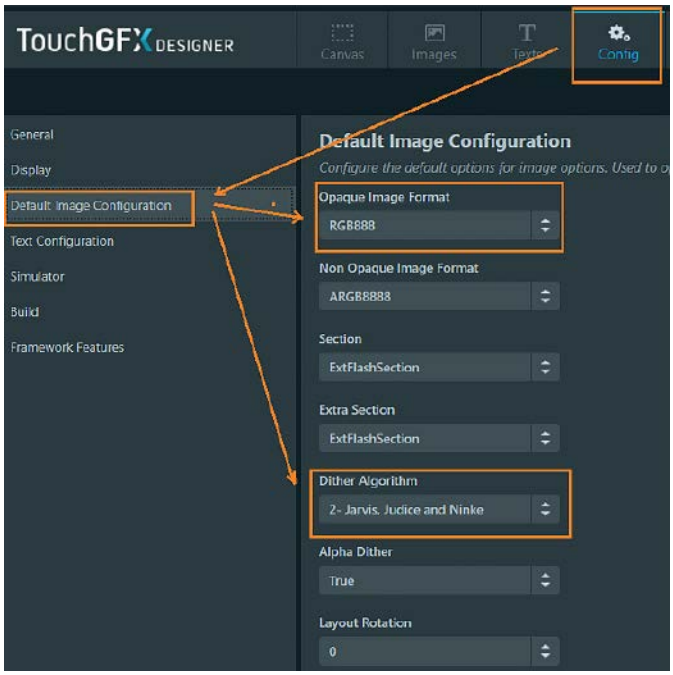
Formalnie zamiast głębi kolorów posługujemy się formatami kolorów. Format koloru oprócz całkowitej ilości bitów opisujących piksel określa też, ile bitów przypada na każdą ze składowych koloru. W TouchGFX kolor o głębi 24 bpp będzie miał format RGB888. Oznacza to, że dla każdego składnika koloru czerwonego, zielonego i niebieskiego używanych jest 8 bitów. W przypadku kolorów 16 bpp TouchGFX używa formatu kolorów RGB565. To znaczy 5 bitów dla czerwonego, 6 bitów dla zielonego, 5 bitów dla niebieskiego. Wybierając głębie koloru powinniśmy się kierować zachowaniem równowagi pomiędzy jakością wyświetlanych informacji a zużyciem pamięci. Najlepsze efekty osiągniemy przy 24 bpp (RGB888), ale być może stosując 16 bpp (RGB565) różnica w wyglądzie będzie niezauważalna a zajętość pamięci dużo mniejsza.

TouchGFX wykorzystuje tzw. *Dithering*, który jest dobrze znaną techniką powodującą, że obrazy wydają się mieć więcej kolorów niż to, co jest w rzeczywistości. Odbywa się to poprzez dodanie odrobiny szumu do kolorów obrazu. Na przykład w czasie konwersji obrazu RGB888 na obraz RGB565, zamiast usuwać najmłodsze bity każdego składnika koloru, proces konwersji dodaje trochę szumu do każdego z wynikowych kolorów. Taka operacja powoduje, że przekonwertowany obraz wygląda jak bogatszy w kolory i podobny do oryginału RGB888.

Zobaczmy teraz jak będzie wyglądało zajęcie pamięci przez bufor ramki wyświetlacza o wymiarach 480×272 pikseli przy głębi bitowej 24 bpp. Każdy piksel jest zapisany w trzech bajtach (RGB888), czyli mamy 480·272·3=391680 bajtów. Dla głębi bitowej 16 bpp, kiedy każdy piksel jest zapisany w bajtach (RGB565) będzie to 480·272·2=261120 bajtów. W bardzo wielu systemach w tym



Rysunek 4. Zakładka Display panelu Config TouchGFX Designer



Rysunek 5. Kodowanie koloru i algorytm dihteringu

w TouchGFX stosuje się 2 ramki i wtedy zajętość pamięci RAM jest dwukrotnie większa.

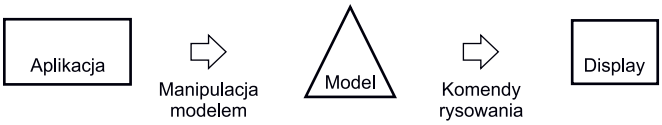
Na **rysunku 3** pokazano wybór głębi bitowej i rozmiaru wyświetlacza w trakcie tworzenia szablonu projektu w programie TouchGFX Designer, natomiast na **rysunku 4** została pokazana zakładka Display z wybranymi ustawieniami rozmiaru wyświetlacza i głębi koloru. Kodowanie koloru i algorytm dihteringu można ustawić w zakładce Default Image Configuration panelu Config projektu (**rysunek 5**).

Silnik grafiki TouchGFX

Podstawowym zadaniem silnika grafiki TouchGFX jest rysowanie elementów graficznych na wyświetlaczu urządzenia wbudowanego. Silniki grafiki można podzielić na dwie główne kategorie:

- Silniki graficzne trybu natychmiastowego (*Immediate mode graphics engines*) zapewniają interfejs API, który umożliwia aplikacji bezpośrednie rysowanie elementów na wyświetlaczu. Obowiązkiem aplikacji jest zapewnienie wywołania prawidłowych operacji rysowania we właściwym czasie;
- Silniki graficzne w trybie zachowanym nazywanym też opóźnionym (*Retained mode graphics engines*) pozwalają użytkownikowi manipulować abstrakcyjnym modelem wyświetlanych komponentów. Silnik dba o przełożenie tego modelu komponentu na prawidłowe operacje rysowania grafiki we właściwych momentach.

Silnik grafiki TouchGFX działa w trybie zachowanym (**rysunek 6**). W skrócie oznacza to, że TouchGFX dostarcza model, którym użytkownik może manipulować. Wywołania klienta nie powodują bezpośrednio rzeczywistego renderowania, ale zamiast tego aktualizują



Rysunek 6. Ogólna koncepcja działania silnika grafiki w trybie zachowanym

wspomniany abstrakcyjny model wewnętrzny (zazwyczaj listę obiektów). Silnik grafiki TouchGFX następnie przejmuje translację z tego modelu na zoptymalizowany zestaw wywołań metod (komend) renderowania.

Działanie na abstrakcyjnym modelu zwalnia programistę z myślenia w kategoriach czasowo – sprzętowych i może się on wyłącznie skupić na implementacji obiektów graficznych. Zalet korzystania z trybu zachowanego jest wiele:

- Łatwość użytkowania: silnik graficzny jest łatwy w użyciu. Użytkownik adresuje konfigurację komponentów na ekranie, wywołując metody na modelu wewnętrznym i nie myśli w kategoriach rzeczywistych operacji rysowania;
- Wydajność: TouchGFX analizuje model sceny i optymalizuje wywołania rysowania potrzebne do zrealizowania modelu na ekranie. Obejmuje to celowe nie rysowanie ukrytych komponentów, rysowanie i przenoszenie tylko zmienionych części komponentów, zarządzanie buforami ramki i wiele więcej;
- Zarządzanie stanami: TouchGFX śledzi, która część modelu sceny jest aktywna. To z kolei ułatwia użytkownikowi optymalizację zawartości modelu sceny.

Wadą tego rozwiązania jest zużycie pamięci większe niż w trybie natychmiastowym.

Manipulacja modelem

Model składa się z komponentów. Każdy z komponentów w modelu ma dokładnie jeden powiązany komponent macierzysty. Sam komponent macierzysty jest również częścią modelu. Tego typu model jest powszechnie nazywany drzewem (**rysunek 7**). Komponent w tym drzewie jest nazywany jako komponent interfejsu użytkownika lub widżet. Z punktu widzenia aplikacji wyświetlane grafiki są aktualizowane poprzez konfigurowanie i manipulowanie widżetami w modelu scen.

Na **listingu 1** pokazano przykład manipulacji widżetem przycisku myButton dodawanego do modelu sceny. W trakcie tych czynności nie określamy, kiedy i w jaki sposób przycisk ma być wyświetlany. Podajemy tylko położenie i identyfikację bitmap skojarzonych ze stanem przyciśnięcia i stanem zwolnienia przycisku. Do rysowania na ekranie silnik TouchGFX wykorzystuje zestaw swoich „rysunkowych" komend API zawierający metody rysowania prymitywów graficznych, takich jak ramki, obrazy, teksty, linie, wielokąty, trójkąty teksturowane itp. Do fizycznego rysowania bitmap i prymitywów wykorzystuje się akceleratory grafiki, na przykład ChromART (jeżeli jest dostępny w mikrokontrolerze). Implementacja metod API rysowania jest specyficzna dla platformy i wysoce zoptymalizowana dla każdego konkretnego MCU.

Silnik grafiki TouchGFX – pętla główna

Działanie wielu silników gier, silników graficznych, w tym również TouchGFX, można traktować jako nieskończoną pętlę wykonującą cykle składające się z trzech podstawowych czynności:

- zbieranie zdarzeń – zbiera zdarzenia z ekranu dotykowego, naciśnięcia fizycznych przycisków, wiadomości/sygnały z systemów podrzędnych na przykład z interfejsów szeregowych,
- aktualizowanie modelu sceny – reaguje na zebrane zdarzenia, aktualizuje pozycje, animacje, kolory, obrazy,
- renderowanie modelu sceny – przerysowuje części modelu, który został zaktualizowany i wyświetla je na ekranie.

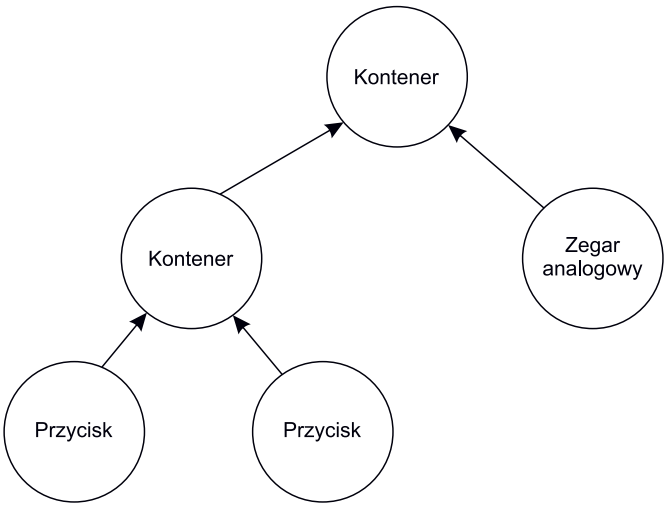
Na **rysunku 8** pokazano ogólny schemat działania silnika grafiki TouchGFX. Każda z czynności silnika jest obsługiwana przez osobną warstwę:

- Zbiór wydarzeń jest obsługiwany przez dedykowaną warstwę abstrakcji TouchGFX AL, którą musimy w potrzebnym zakresie zmodyfikować do własnych potrzeb;
- Aktualizacja modelu zależy całkowicie od warstwy aplikacji tworzonej przez programistę;
- Renderowanie grafiki do bufora ramki jest obsługiwane przez TouchGFX i generalnie nie trzeba go dostosowywać.

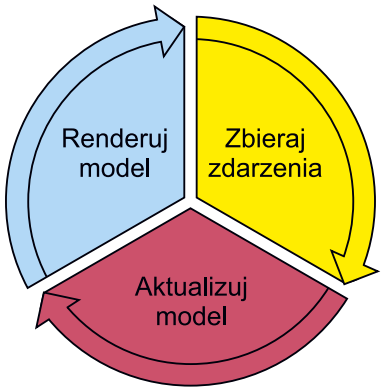
Po renderowaniu ramki zawartość bufora ramki jest przesyłana do wyświetlacza. Transfer danych musi być zsynchronizowany z układem wyświetlacza, po to by uniknąć wyświetlania zakłóceń na ekranie. Zależnie od rozwiązania wyświetlacza transfery danych muszą być wykonywane sekwencyjnie co określony minimalny interwał lub alternatywnie po otrzymaniu z układu wyświetlacza sygnału wyzwalającego możliwość transferu danych. Silniki TouchGFX czeka na taki sygnał przesyłany z warstwy abstrakcji sprzętu.

Działanie silnika TouchGFX omówimy dokładniej w kolejnej części tego cyklu.

Tomasz Jabłoński, EP



Rysunek 7. Drzewo komponentów sceny



Rysunek 8. Schemat działania silnika TouchGFX

Listing 1. Przykład manipulacji widżetem

```
myButton . setXY ( 100 , 50 );
myButton . setBitmaps ( Bitmap ( BITMAP_BUTTON_RELEASED_ID ) , Bitmap ( BITMAP_BUTTON_PRESSED_ID ) );
add ( myButton );
```