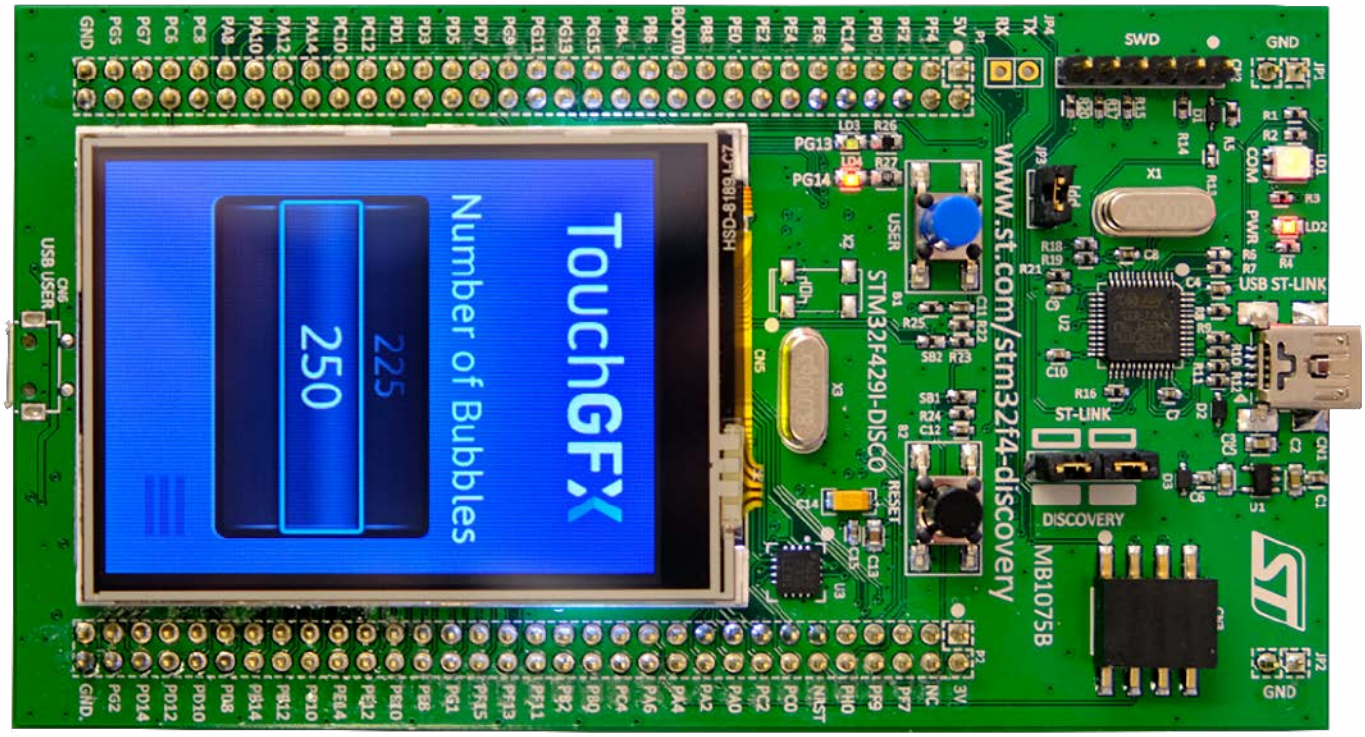


FTP Wszystkie poprzednie odcinki kursu są dostępne na stronie www.media.avt.pl

Projektowanie interfejsów graficznych z użyciem TouchGFX (4)

W poprzedniej części ukończyliśmy prosty interfejs użytkownika i rozpoczęliśmy analizę działania silnika grafiki TouchGFX. W ostatniej części tego cyklu dokończymy analizę, ponieważ pozwoli to zrozumieć, jak jest zorganizowany transfer danych do sterownika wyświetlacza i jak uniknąć wyświetlania zakłóceń na ekranie.

Silnik TouchGFX zobrazowaliśmy 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.

Zbieranie zdarzeń

W tej fazie silnik graficzny zbiera zdarzenia ze środowiska zewnętrznego. Te zdarzenia to zazwyczaj zdarzenia związane z ekranem dotykowym i przyciskaniem przycisków zewnętrznych. TouchGFX próbkuje te elementy i przesyła wykryte zdarzenia do warstwy aplikacji. W przypadku ekranów dotykowych TouchGFX wykrywa następujące zdarzenia:

- kliknięcie – użytkownik nacisnął lub zdjął palec z wyświetlacza,
- przeciągnięcie – użytkownik przesunął palcem po wyświetlaczu (dotykając wyświetlacza),
- gesty – użytkownik szybko przesunął palec w wybranym kierunku, a następnie puścił. Nazywa się to machnięciem i jest rozpoznawane przez silnik graficzny.

Po wykryciu i zidentyfikowaniu są one przekazywane do aktualnie aktywnych elementów interfejsu użytkownika (np. widżetów).

Wyróżnionym zdarzeniem jest zdarzenie tick. Tick reprezentuje nową ramkę (lub krok w czasie) i jest zawsze wysyłane, także wtedy, gdy nie było innego zdarzenia. Jest ono używane przez aplikacje do uruchamiania animacji lub innych działań opartych na czasie, takich jak przejście do ekranu pauzy po upływie określonego czasu.

Aktualizowanie modelu

Jednym z zadań silnika graficznego jest współpraca z warstwą aplikacji mająca na celu aktualizację interfejsu użytkownika. Ta aktualizacja odzwierciedla zebrane zdarzenia. Silnik graficzny wie, który ekran (*screen*) jest aktualnie aktywny i przekazuje zdarzenia do obiektów tego ekranu.

Inaczej mówiąc, silnik informuje aplikację o zdarzeniach. W odpowiedzi aplikacja żąda przerysowania określonych elementów na wyświetlaczu. Aplikacja w modelu zachowanym (*retained*) nie rysuje bezpośrednio w odpowiedzi na zdarzenia, ale zmienia właściwości widżetów i żąda przerysowania przez warstwę Renderuj modelu silnika graficznego.

Jeśli na przykład wystąpi zdarzenie Click, silnik graficzny przeszukuje model sceny obiektu Screen, aby znaleźć widżet, który powinien odebrać zdarzenie. Niektóre widżety, takie jak Image i TextArea, nie obsługują zdarzeń Click i mają pustą procedurę obsługi zdarzeń, więc wtedy nic się nie dzieje. Inne widżety, takie jak przycisk, reagują na zdarzenie Click (naciśnięte lub zwolnione). Widżet Przycisk (*button*) zmienia swój stan, aby wyświetlić inny obraz po naciśnięciu i znów zmienia stan po ponownym zwolnieniu dotknięcia. Dlatego musi być ponownie „narysowany" w buforze ramki wyświetlacza. Co znamienne, silnik grafiki sam nie inicjuje tego przerysowania widżetów na podstawie zebranych zdarzeń. Widżety śledzą swój własny stan wewnętrzny i instruują silnik, żeby przerysował część ekranu (na przykład prostokąt) obrazujący widżet.

Sama aplikacja może również reagować na zdarzenia na jeden z dwu sposobów:

- Przez konfigurację interakcji dla widżetu w TouchGFX Designer. Na przykład, możemy skonfigurować interakcję, aby kolejny widżet był widoczny po naciśnięciu przycisku. Ta interakcja jest wykonywana po zmianie stanu przycisku i zażądaniu ponownego rysowania od silnika graficznego. Jeśli używamy interakcji, aby wyświetlić inny (niewidoczny) widżet, aplikacja powinna również zażądać przerysowania z silnika graficznego;
- Reagowanie na zdarzenia na ekranie. Możliwe jest również reagowanie na zdarzenia na samym ekranie. Program obsługi zdarzeń to funkcje wirtualne w klasie Screen. Funkcje te można ponownie zaimplementować na ekranach w aplikacji. Można to na przykład wykorzystać do wykonania akcji za każdym razem, gdy użytkownik dotknie ekranu, bez względu na to, który widżet jest dotykany.

Listing 1. Zależności czasowe

```
void handleTickEvent ( ) {
tickCounter + = 1 ;
    if ( tickCounter == 600 ) {
        // Przejście do 0 = niewidoczne w 20 //klatkach
        myWidget . startFadeAnimation ( 0 , 20 ) ;
    }
}
```

Listing 2. Odpytywanie GPIO w klasie *Model*

```
void Model :: tick ( ){
    // Przykładowe odpytywanie GPIO
    bool b = sampleGPIO_Input1 ( ) ;
    if ( b ){
        . . .
    }
}
```

Listing 3. Przerysowanie widżetu box1

```
void handleTickEvent() {
    tickCounter += 1;
    if (tickCounter == 10) {
        // ustawienie koloru czerwonego
        box1.setColor(Color::getColorFrom24BitRGB(0xFF, 0x00, 0x00));
        // zezwolenie na przerysowanie
        box1.invalidate();
    }
}
```

Aktualizacje czasowe

Procedura obsługi zdarzenia *handleTickEvent* jest wywoływana na początku każdej ramki. Dzięki temu aplikacja może przeprowadzać aktualizacje interfejsu użytkownika w oparciu na czasie. Przykładem może być zniknięcie widżetu po 10 sekundach. Zakładając, że mamy 60 klatek na sekundę, kod mógłby wyglądać tak, jak na **listingu 1**. Silnik graficzny wywołuje również procedurę obsługi zdarzeń w klasie *Model*. Ten program obsługi zdarzeń jest zwykle używany do wykonywania powtarzających się czynności, takich jak sprawdzanie kolejek komunikatów lub próbkowanie GPIO (**listing 2**).

Jak omówiliśmy powyżej w przykładzie Button, widżety są odpowiedzialne za żądanie przerysowania, gdy zmienia się ich stan. Taki mechanizm nazywany jest unieważnionym obszarem (*invalidated area*). Kiedy przycisk zmienia stan (np. ze zwolnionego na wciśnięty) i wymaga przerysowania, obszar objęty widżetem przycisku staje się obszarem unieważnionym. Silnik graficzny przechowuje listę tych unieważnionych obszarów żądanych dla ramki. Wszystkie zebrane zdarzenia (dotknięcie, przycisk, tik) mogą skutkować jednym lub kilkoma unieważnionymi obszarami, więc w każdej klatce może być wiele unieważnionych obszarów.

Procedury obsługi zdarzeń w klasie Screen mogą również zażądać odświeżenia obszaru. Tutaj zmieniamy kolor widżetu Box, box1, w ramce 10 i żądamy przerysowania, wywołując metodę unieważnienia (*invalidate*) (**listing 3**). W pokazanym przykładzie silnik graficzny będzie wywoływał procedurę obsługi zdarzenia *handleTickEvent* w każdej ramce. W ramce 10 kod aplikacji żąda przerysowania obszaru objętego widżetem box1. W odpowiedzi na to silnik graficzny przerysuje ten obszar w buforze ramki, używając koloru zapisanego w argumencie *setColor*.

W kolejnym przykładzie interfejsu użytkownika pokazanym poniżej mamy widżety Button i Box. Jeśli wstawimy interakcję zmieniającą kolor ramki po kliknięciu przycisku, to otrzymamy dwa unieważnione obszary (zaznaczone na czerwono). Jeden dotyczy przerysowania przycisku po jego dotknięciu, a drugi przerysowania koloru obszaru box (**rysunek 1**).

Renderowanie

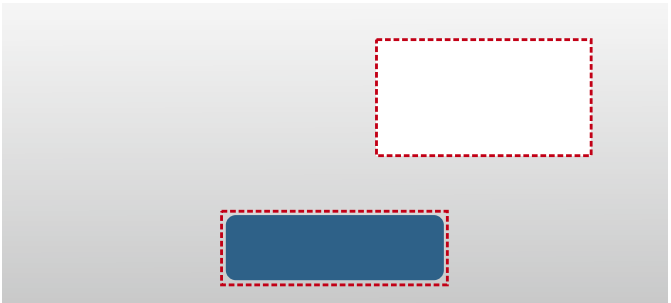
Tak jak to omówiliśmy, wynikiem fazy aktualizacji jest lista obszarów do przerysowania, nazwanymi obszarami unieważnionymi. Zadaniem fazy renderowania modelu jest przejrzenie tej listy i narysowanie widżetów pokrywających te obszary w buforze ramki. Faza renderowania modelu jest obsługiwana automatycznie przez silnik graficzny. Aplikacja ma za zadanie zdefiniowanie modelu sceny (widżety w interfejsie użytkownika) i unieważnienie określonych obszarów. Silnik graficzny obsługuje unieważnione obszary jeden po drugim. Dla każdego obszaru silnik skanuje model sceny i gromadzi listę widżetów objętych tym obszarem (częściowo lub w całości). Mając tę listę widżetów, silnik wywołuje metodę rysowania na widżetach, zaczynając od widżetu w tle, a kończąc na głównym widżecie.

Metody rysowania widżetu używają parametrów jego stanu np. koloru, podczas rysowania do bufora ramki. Wszelkie informacje potrzebne do narysowania widgetu muszą zostać zapisane w widżecie podczas fazy aktualizacji. W przeciwnym razie te informacje nie są dostępne w fazie renderowania.

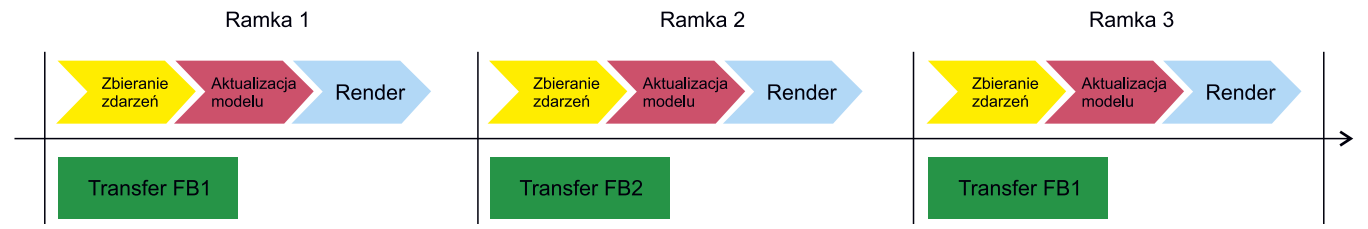
Stan wstrzymania *Wait*

Silnik graficzny TouchGFX czeka na sygnał gotowości przed aktualizacją i renderowaniem następnej klatki. Istnieją dwa powody, dla których należy czekać między klatkami zamiast ciągłego renderowania ramek tak szybko, jak to możliwe:

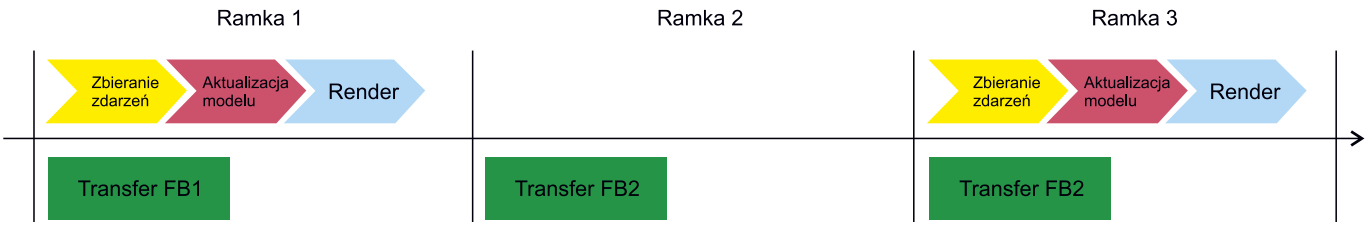
- Renderowanie jest zsynchronizowane z wyświetlaczem. Niektóre wyświetlacze wymagają wielokrotnego przesyłania bufora ramki. Podczas transmisji nie zaleca się arbitralnego renderowania



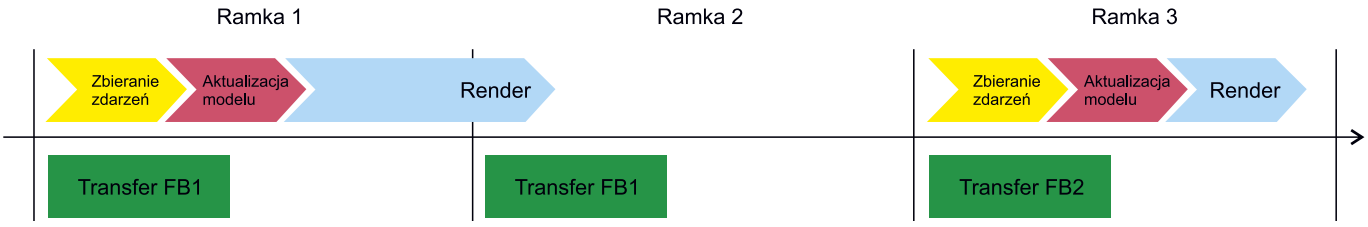
Rysunek 1. Przykład dwu obszarów unieważnionych interfejsu użytkownika



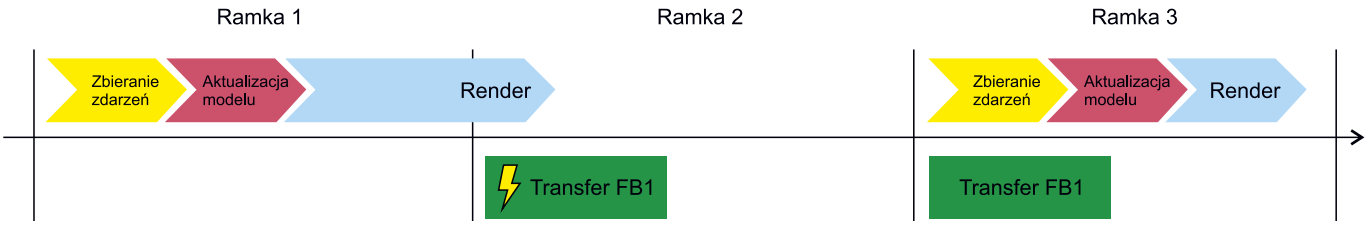
Rysunek 2. Podwójny bufor ramki



Rysunek 3. Ramka 2 nie jest aktualizowana



Rysunek 4. Dłuższe renderowanie ramki 1



Rysunek 5. Jeden bufor ramki i długie renderowanie

do bufora ramki. Dlatego silnik graficzny czeka przez krótki czas po uruchomieniu transmisji przed rozpoczęciem renderowania. Inne wyświetlacze wysyłają sygnał do mikrokontrolera, kiedy należy przesłać bufor ramki. Silnik graficzny czeka na ten sygnał;

- Ramki są renderowane ze stałą szybkością. Dla aplikacji często korzystne jest renderowanie klatek ze stałą szybkością, ponieważ ułatwia to tworzenie animacji trwających przez określony czas. Na przykład, jeśli masz wyświetlacz 60 Hz, dwusekundowa animacja powinna zostać zaprogramowana tak, aby kończyła się w 120 klatkach.

Czas oczekiwania silnika graficznego jest zwykle używany przez inne procesy o niższym priorytecie w aplikacji. W takich przypadkach czas nie jest marnowany, ponieważ procesy o niższym priorytecie i tak powinny być uruchomione w pewnym momencie.

Obsługa buforów ramki

Jak już wiemy, silnik graficzny synchronizuje się z wyświetlaczem przed aktualizacją bufora ramki. Po wyrenderowaniu do bufora ramki silnik musi również upewnić się, że wyświetlacz pokazuje zaktualizowany bufor ramki.

Dwa bufory ramki

W najprostszej konfiguracji dostępne są dwa bufory ramki. Silnik graficzny pracuje naprzemiennie między dwoma buforami ramki. Podczas rysowania ramki do bufora ramki drugi bufor ramki jest przesyłany i pokazywany na wyświetlaczu (**rysunek 2**). Zakładamy, że wyświetlacz ma równoległy interfejs RGB podłączony do kontrolera LTDC. Oznacza to, że bufor ramki musi być przesyłany do wyświetlacza w każdej ramce. Ponieważ mamy dwa bufory ramki, silnik graficzny może rysować do jednego bufora ramki, podczas gdy drugi jest przesyłany do wyświetlacza. Ten schemat działa bardzo dobrze i jest preferowany, jeśli to możliwe.

Często zdarzają się ramki, w których aplikacja niczego nie aktualizuje. Oznacza to, że nic nie jest renderowane. Dlatego ten sam bufor ramki jest przesyłany ponownie w następnej ramce. Na **rysunku 3** aplikacja nie rysuje niczego w ramce 2, więc silnik graficzny retransmituje bufor ramki 2 ponownie w ramce 3.

Typowy równoległy wyświetlacz RGB ma częstotliwość odświeżania około 60 Hz. Ta częstotliwość aktualizacji musi być utrzymywana przez mikrokontroler. Częstotliwość odświeżania 60 Hz oznacza, że mamy ok. 16 ms na renderowanie nowej ramki przed ponownym rozpoczęciem transmisji. W niektórych przypadkach ten czas jest dłuższy niż 16 ms, tak jak to pokazano na **rysunku 4**. W tym przypadku silnik graficzny po prostu ponownie przesyła tę samą ramkę. Renderowanie klatki 1 trwa dłużej niż 16 ms, więc ramka 0 wcześniej wyrenderowana do bufora ramki 1 jest ponownie przesyłana. Nowa ramka w buforze ramki 2 jest przesyłana w ramce 3. Gdy dostępne są dwa bufory ramki, czas renderowania może być bardzo długi. Poprzednia ramka jest retransmitowana do momentu udostępnienia nowej ramki.

Jeden bufor ramki

W niektórych systemach dostępna jest tylko pamięć dla jednego bufora ramki. Jeśli mamy równoległy wyświetlacz RGB, jesteśmy zmuszeni transmitować bufor ramki 1 w każdej ramce. Może to być problematyczne, ponieważ silnik graficzny jest zmuszony do rysowania do tego samego bufora ramki, który jednocześnie przesyłamy na wyświetlacz. Jeśli zostanie to zrobione bez zachowania pewnych

zasad, istnieje duże ryzyko, że wyświetlacz pokaże klatkę, która jest połączeniem poprzedniej i nowej klatki.

Jednym z rozwiązań jest wstrzymanie rysowania do zakończenia transferu i rysowanie tylko w przedziale czasowym przed ponownym rozpoczęciem transferu. Daje to mało czasu na narysowanie, ponieważ transfer zajmuje znaczną część całkowitego czasu ramki. Może się wtedy zdarzyć, że będą wysyłane niekompletne ramki, bo silnik nie zdążył z renderowaniem przed startem transferu.

Rozwiązaniem eliminującym potencjalnie tę wadę jest śledzenie, jaka część bufora ramki jest już wytransmitowana, a następnie ograniczenie renderowania do tej odpowiedniej części bufora ramki. W miarę postępu transferu coraz więcej bufora ramki jest dostępne dla algorytmów renderowania. Silnik graficzny zawiera algorytmy, które pomagają programiście upewnić się, że rysunek jest wykonywany poprawnie.

Sytuacja, w której czas renderowania jest dłuższy niż 16 ms, renderowanie nie zakończy się, gdy retransmisja rozpocznie się ponownie, jest pokazana na **rysunku 5**. Wtedy silnik graficzny musi upewnić się, że przesyłana część jest całkowicie renderowana. W przeciwnym razie wyświetlacz pokaże niedokończony bufor ramki.

Wydajność systemu graficznego

Przy omawianiu sytuacji pokazanej na rysunku 4 ustaliliśmy, że przy podwójnym buforze ramki czas renderowania może być dowolnie długi, bo do jego zakończenia jest ciągle wysyłana poprzednia ramka. To jest prawdziwe w przypadku, kiedy wyświetlane są obrazy statyczne. Sytuacja się zmienia, kiedy w interfejsie użytkownika chcemy zastosować animacje. Do płynnego wyświetlania animacji niezbędna jest możliwość wyświetlania określonej liczby ramek (klatek) na sekundę. Do tego ta liczba klatek nie powinna się zmieniać w czasie, bo powstaje wtedy wrażenie przyspieszenia lub opóźniania animacji, co jest odbierane jako bardzo niekorzystne wrażenie wizualne. Żeby wyświetlać zaawansowane animacje, niezbędna jest określona wysoka wydajność układu interfejsu użytkownika. Wysoka wydajność jest tutaj definiowana jako uzyskiwanie dużej liczby klatek na sekundę przy jednoczesnym uzyskiwaniu pożądanych efektów graficznych i animacji.

Przypomnijmy sobie, jak główna pętla wpływa na liczbę klatek interfejsu użytkownika. Załóżmy ponownie, że do LTDC jest podłączony równoległy wyświetlacz RGB i dwa bufory ramki (rysunek 2). Zakładając, że wyświetlacz jest odświeżany 60 razy na sekundę, między każdym odświeżeniem jest około 16 ms: 1 s/60=0,01667 s=16,67 ms.

TouchGFX rozpoczyna rysowanie ramki do bufora ramki 2 w momencie rozpoczęcia przesyłania bufora ramki 1. Jeśli renderowanie ramki 1 zostanie zakończone przed rozpoczęciem następnego przesyłania, możemy przesłać bufor ramki 2. Jeśli nie zakończy się w ciągu 16,67 ms, bufor ramki 1 zostanie ponownie przesłany i wyświetlacz będzie wyglądał niezmieniony tak jak na rysunku 3. W takim przypadku utracimy jedną klatkę animacji. Czas dla faz zbierania zdarzeń i aktualizacji modelu jest zwykle niewielki, np. krótszy niż 1 ms. Dlatego w dalszej części rozważań dla uproszczenia przyjmiemy, że czas renderowania jako najdłuższy zawiera też fazy zbierania danych i aktualizacji modelu.

Jeśli czas renderowania w wielu klatkach przekroczy limit wynoszący 16,67 ms, liczba klatek na sekundę na wyświetlaczu realnie wyniesie 30 klatek na sekundę (fps). Jeśli renderowanie jest generalnie krótsze niż 16,67 ms, ale w niektórych klatkach dłuższych niż 16,67 ms, średnia liczba klatek na sekundę może być bliska 60 fps, ale animacja może nie wyglądać poprawnie. W zależności od aplikacji może wyglądać, że część animacji działa szybko (poprawnie), a część zwalnia. To nie jest pożądane zjawisko. Jeżeli czas renderowania jest jeszcze dłuższy, na przykład nieco powyżej 33 ms, realna liczba klatek na sekundę spadnie do 20 fps, ponieważ mamy tylko nową ramkę gotową na co trzeci transfer. Zostało to pokazane na **rysunku 6**.

Zaawansowany użytkownik może mierzyć czas pomiędzy kolejnymi transferami. Silnik graficzny wywołuje funkcję w klasie GPIO, gdy rozpoczyna się faza zbierania zdarzeń i wykonuje kolejne wywołanie po zakończeniu fazy renderowania. Aplikacja definiuje te funkcje. Mogą to być na przykład zmiany stanów linii GPIO. Potem za pomocą oscyloskopu można zmierzyć czas renderowania kolejnych ramek. Alternatywną metodą jest zliczanie utraconych klatek. Silnik graficzny zlicza liczbę transferów, które miały miejsce podczas ostatniej fazy zbierania-aktualizowania-renderowania. Aplikacja może łatwo sprawdzić tę wartość, aby zobaczyć, czy klatka została utracona, a tym samym zmniejszyła się liczba klatek na sekundę.

FPS	Maksymalny czas renderowania
60	16,67 ms
30	33,34 ms
20	50,00 ms
15	66,67 ms

Rysunek 6. Maksymalny czas renderowania w połączeniu z liczbą klatek na sekundę (FPS)

Kiedy czas renderowania jest za długi i klatki się gubią, a co za tym idzie, zmniejsza się liczba klatek na sekundę w jednej z naszych animacji, możemy to do pewnego stopnia skompensować poprzez jedno z działań:

- poczekaj – pozwól animacji trwać, co spowoduje wydłużenie czasu trwania animacji i prawdopodobnie płynną animację,
- pomiń niektóre klatki – pomijając klatki, upewnij się, że ogólna animacja nie trwa dłużej, niż zamierzano.

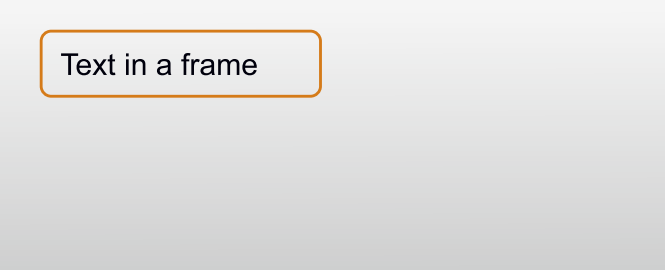
TouchGFX może zostać poinstruowany, aby automatycznie pomijał niektóre klatki, gdy zostaną utracone. Można to osiągnąć, zaznaczając animacje więcej niż raz na rzeczywistą klatkę. Może to pomóc w zwiększeniu płynności animacji, gdy czas renderowania jest nierówny.

Na czas renderowania wpływa wiele różnych czynników: rozmiar zaktualizowanych obszarów, użycie warstw, złożoność widżetów oraz dostępna sprzętowa obsługa przerysowania. Czas renderowania jest zwykle proporcjonalny do liczby pikseli, które należy zaktualizować. Jeśli więc przerysowanie animacji zajmuje zbyt dużo czasu, możliwym rozwiązaniem jest zmniejszenie obszaru animacji. Na przykład, jeśli masz obracający się obraz, a wydajność systemu nie jest wystarczająco dobra, można ją poprawić, zmniejszając rozmiar obrazu. Należy pamiętać, że silnik graficzny przerysowuje obszary, które aplikacja unieważniła. Dlatego ważne jest, aby unieważnić tylko te obszary, które faktycznie wymagają odświeżenia.

W typowej aplikacji grafika będzie składać się z różnych elementów, które są ułożone jeden na drugim w warstwach. Jeśli jeden z elementów jest aktualizowany, wszystkie elementy warstw muszą być zwykle przerysowane.

Typowy przykład został pokazany na **rysunku 7**. Ten interfejs użytkownika jest tworzony przez umieszczenie widżetu TextArea na widżecie Obraz wyświetlającym przezroczystą ramkę. To rozwiązanie jest bardzo często stosowane w aplikacji, bo jest bardzo proste i zapewnia dużą elastyczność. Można np. zmieniać ramkę w czasie wykonywania lub przesuwać ramkę i tekst w tle.

Jeśli tekst jest aktualizowany w czasie wykonywania i wymaga ponownego narysowania, silnik graficzny również musi przerysować tło i ramkę a następnie nowy tekst. Znacznie wydłuża to czas renderowania tekstu. Im więcej warstw do przerysowania, tym dłuższy czas jest potrzebny na przerysowanie.



Rysunek 7. Typowym przykładem jest obraz tła, ramka i tekst

We wszystkich typach renderowania silnik graficzny musi zapisać wynikowy piksel do bufora ramki. Okazuje się, że nakład mocy obliczeniowej na wyliczenie wartości reprezentującej piksel w buforze nie jest jednakowy w każdym przypadku. Na przykład w widżecie Box wszystkie piksele mają ten sam kolor i nie ma potrzeby wyliczania ich wartości za każdym razem, kiedy jest na przykład przesuwany czy obracany. W interfejsie zbudowanym z wielu takich elementów nie jest potrzebna duża moc obliczeniowa, ale z drugiej strony jakość i atrakcyjność takiego interfejsu jest problematyczna.

Podobnie jest w przypadku obrazków, ponieważ piksele są przechowywane w mapach bitowych w formacie gotowym do użycia. Obliczenie piksela zapisanego w buforze ramki polega na załadowaniu wartości koloru z odpowiedniego miejsca w mapie bitowej.

Wyświetlanie tekstu można traktować jak wyświetlanie małych bitmap. Jednak tu jest potrzebna trochę większa wydajność na przykład do obliczania pozycji kolejnych liter. Ładnie wyglądający tekst będzie wymagał wyświetlania z przezroczystością tak jak w przykładzie z rysunku 7. Przezroczystość zwiększa zapotrzebowanie na obliczenia niezbędne do narysowania elementu, ponieważ silnik graficzny musi najpierw narysować element za przezroczystym elementem (tak jak w przykładzie z rysunku 7). W kolejnym kroku silnik graficzny musi połączyć piksel tła z pikselem elementu przezroczystego i zapisać wynik do bufora ramki. Te obliczenia zajmują znacznie więcej czasu niż samo zapisanie obliczonego piksela.

Obracanie i operacje przeskalowania obrazków wymagają również dużej mocy obliczeniowej. Co prawda piksele są pobierane z mapy bitowej, ale do ich wyświetlania należy przeliczyć nową pozycję, uwzględniając rotacje i/lub skalowanie. Rysowanie elementów geometrycznych jest jedną z najbardziej obciążających operacji. Nie możemy załadować koloru piksela z mapy bitowej, ale musimy obliczyć zarówno kształt figury (na przykład koła), jak i kolor poszczególnych pikseli.

Przy projektowaniu interfejsu trzeba pamiętać o kilku ważnych zasadach pozwalających na uniknięcie niepotrzebnego obciążania systemu (mikrokontrolera):

- Nie zlecaj silnikowi grafiki przerysowania niezmienionych elementów grafiki. Upewnij się, że przypadkowo nie unieważnisz (*invalidated area*) niepotrzebnych części wyświetlacza. Zwiększa to obciążenie systemu bez żadnych korzyści;
- Znajdź równowagę między jakością interfejsu a szybkością. Zmniejszenie złożoności elementów może poprawić szybkość i jakość działania. Często kluczem jest odpowiednia równowaga między tymi elementami;
- Wykorzystaj możliwości sprzętowe Możliwości graficzne mikrokontrolera z akceleracją sprzętową (Chrom-ART) są często większe niż bez niej. Rozważ użycie mikrokontrolera z Chrom-ART;
- Zastąp konieczność obliczania elementów grafiki obrazami (bitmapami). Wyliczanie narysowania okręgu jest wolniejsze niż wyświetlenie obrazu koła. Obrazy mogą zastąpić wiele statycznych elementów;
- Dostosuj częstotliwość odświeżania ekranu. Jak to omówiliśmy, częstotliwość odświeżania jest sztywnym ograniczeniem czasu renderowania. Jeśli czas renderowania przekracza częstotliwość odświeżania, liczba klatek spada. Jeśli czas renderowania jest tylko trochę dłuższy od częstotliwości odświeżania, możliwe jest obniżenie częstotliwości odświeżania wyświetlacza do np. 55 Hz (co odpowiada 18,2 ms) i utrzymanie wysokiej częstotliwości odświeżania.

Do tej pory skupiliśmy się na zapewnieniu interfejsowi użytkownika (HMI) poprawnego, płynnego działania. Jak wiemy, sprzęt potrzebny do tego celu: mikrokontroler, najlepiej z akceleratorem grafiki i pozostałe zasoby, powinien być dość wydajny i rozbudowany jak na systemy wbudowane (*embeded*). HMI w systemach wbudowanych to tylko (lub aż) interfejs zapewniający interakcję z właściwym systemem sterowania lub nadzoru. Należy sobie postawić pytanie, czy dość zaawansowany system obsługujący HMI można wykorzystać do tych innych zadań. Z poprzednich rozważań wiemy, że silnik grafiki TouchGFX może w pewnych przypadkach potrzebować zająć cały czas procesora. Przy skomplikowanych animacjach nawet ten czas może nie być wystarczający. Trudno sobie wyobrazić, żeby interfejs graficzny miał wyższe priorytety i zwalniał podstawowe procesy sterowania i nadzoru systemu wbudowanego.

W prostym urządzeniu z graficznym interfejsem użytkownika i tylko kilkoma prostymi zadaniami pomocniczymi, takimi jak prosty timer, pomiar temperatury czy funkcja zegara RTC, możliwe jest zorganizowanie całej aplikacji wokół kodu interfejsu użytkownika. Aplikacja robi bardzo niewiele poza regularnymi aktualizacjami interfejsu użytkownika, więc wykonywanie innych zadań może z dużym powodzeniem zostać osadzone w kodzie interfejsu użytkownika. Ale kiedy urządzenie zawiera bardziej zaawansowaną funkcjonalność, która „działa w tle" z oddzielnymi wymaganiami czasowymi, na przykład takimi jak sterowanie silnika BLDC, szybko staje się trudne zintegrowanie tych dwóch zadań w jedno przy jednoczesnym spełnieniu wymagań stawianym aplikacji.

Jak już wiemy, silnik graficzny musi rysować nowe ramki, aby obsługiwać płynny interfejs użytkownika. Jeśli wstrzymamy to rysowanie podczas wykonywania innych zadań, to liczba klatek na sekundę spadnie. Podobnie, jeśli inne zadania działają tylko między klatkami, w czasie bezczynności silnika grafiki, wtedy inne zadania będą wstrzymywane, gdy interfejs użytkownika renderuje złożone sceny. Te zależności utrudniają lub nawet uniemożliwiają przeplatanie zadania interfejsu użytkownika innymi złożonymi zadaniami sterowania systemem wbudowaneym.

TouchGFX działa pod kontrolą systemu FreeRTOS. Funkcje systemowe pozwalają na uporządkowanie, kontrolę i współpracę pomiędzy uruchomionymi zadaniami. FreeRTOS zapewnia też standardowe mechanizmy komunikacji pomiędzy zadaniami, nadawanie im priorytetów i obsługę przerwań. Znacznie ułatwia to tworzenie złożonych aplikacji, ale w przypadku niedoborów wydajności nie likwiduje opisywanych wyżej problemów. Często, jeżeli chcemy mieć pewny i wydajny system sterowania i atrakcyjny interfejs HMI, trzeba obie te części podzielić na osobne systemy i zapewnić współpracę przez standaryzowany protokół, na przykład MODBUS.

Tomasz Jabłoński, EP