LEDGRID

Wim Leers
Universiteit Hasselt
Agoralaan - Gebouw D, 3590 Diepenbeek, Belgium
mail@wimleers.com

ABSTRACT

In deze paper wordt ons project voor het vak Mobile & Pervasive Computing toegelicht. We hebben gekozen om een 8×8 matrix van RGB leds — vandaar de naam: LEDGRID — te bouwen, met boven iedere led een matte, doorzichtige knop. Zo wordt directe interactie met het scherm mogelijk gemaakt. De bedoeling is dat hier makkelijk apps voor kunnen geschreven worden, opdat het een multifunctioneel apparaat is.
Daarnaast werd een LEDGRID emulator geschreven, opdat het bouwen van nieuwe apps eenvoudiger werd. Dit bleek ook noodzakelijk om tijdens het bouwen van de fysieke LEDGRID reeds aan apps te kunnen werken. Zo konden we ons ook beperken tot één fysieke LEDGRID (zie figuur 1) en tóch nog netwerkfunctionaliteit ondersteunen.

INTRODUCTIE

Ik heb dit project samen met Jens Bruggemans gemaakt, en allebei vonden we het nogal beperkend om een apparaat met één specifieke functie te maken. We wilden iets bouwen dat meerdere functies kon hebben — iets dat in meerdere situaties nuttig zou kunnen zijn. En als iets meerdere functies kan hebben, is het natuurlijk nog beter als de gebruiker gemakkelijk zelf nieuwe functionaliteit kan toevoegen.

Na een tijd brainstormen en zoeken, hebben we het Monome project [9] gevonden.

Wat is een Monome?

Een Monome (zie figuur 2) is een herconfigureerbare grid van ledknoppen: knoppen met daaronder een led. Er wordt gebruik gemaakt van *monochrome* leds, dus met 1 specifieke kleur: wit, rood, groen of blauw. De bouwkwaliteit is, zoals de foto in figuur 2 al doet vermoeden, uitstekend.

Er zijn Monomes in meerdere smaken [10]: met 256 knoppen (16x16), 128 (16x8) en 64 knoppen (8x8). Monomes worden in beperkte oplages geproduceerd, met telkens slechts één model dat op een gegeven moment kan besteld worden — het is immers een project dat wordt gerund door enkele enthousiastelingen en niet gebackt wordt door een groot bedrijf. Op het moment van schrijven wordt de *Monome grayscale one twenty eight* aangeboden, voor USD $800 — het mag dus duidelijk zijn dat aan de hoge kwaliteit van Monomes ook een stevig prijskaartje hangt.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UbiComp '10, Sep 26-Sep 29, 2010, Copenhagen, Denmark.
Copyright 2010 ACM 978-1-60558-843-8/10/09...$10.00.

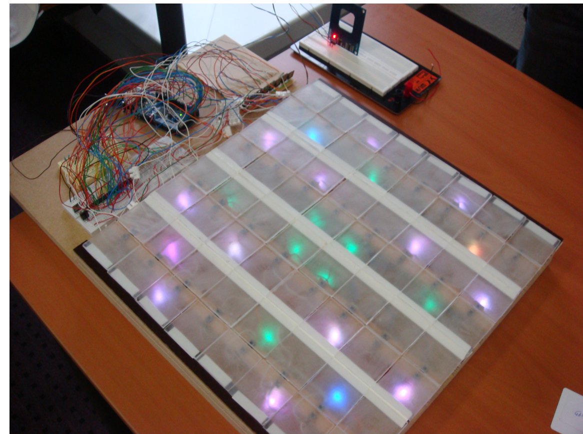

Figure 1. De voltooide LEDGRID.

Figure 2. Een *grayscale sixty four* Monome.

Interactie tussen de knoppen en de leds eronder wordt bepaald door de applicatie die op de computer uitgevoerd wordt. Met andere woorden: een Monome is eenvoudigweg een doos met leds en knoppen, waarbij je met een schone lei kan beginnen aan het ontwerpen van een ”interface” voor om het even welke functionaliteit. Zonder andere complexiteiten die in de weg kunnen komen te zitten: de interface is beperkt tot wat je ziet.

Een groot nadeel — en tegelijk misschien ook een sterkte — aan Monome is dat het niet onafhankelijk kan werken: het moet verbonden zijn met een computer via USB, waarop de applicaties dan kunnen uitgevoerd worden. Er is eenvoudige communicatie in twee richtingen: een knop induwen zendt een bericht naar de computer via USB; in omgekeerde richting kan een led ingeschakeld worden door de computer via een bericht over de USB kabel.

LEDGRID: MONOME KLOON, MAAR BETER

Er bestaat reeds een *Arduinome* [5] project (wat instructies bevat om een Monome kloon te maken op basis van een Arduino, maar met een lagerprijskaartje), maar uiteraard was het niet de bedoeling van dit vak om zonder zelf na te denken bestaande instructies te volgen. Bovendien geldt ook voor Arduinome dat het nog steeds vereist is om verbonden te zijn met een computer.

Ons doel was dus om net zoals bij Arduinome een Arduino bordje te gebruiken als basis, maar verder onze eigen weg te zoeken. Bovendien wilden we ervoor zorgen dat het eindresultaat onafhankelijk van een computer kan gebruikt worden, én wilden we niet met monochrome leds maar met RGB leds werken, opdat geavanceerdere informatieweergave mogelijk zou worden. Als naam vonden we *LEDGRID* wel toepasselijk.

De kosten moesten echter beperkt blijven, dus kozen we voor een 8x8 LEDGRID. Wetende dat een 8x8 Monome al USD $500 kost exclusief verzendingskosten, hadden we dus als nevendoelstelling om daar vér onder te blijven.

RFID Tag App Switching

Aangezien het voor dit project ook vereist om niet-conventionele (sensor-based) interactie te integreren, hebben we er voor geopteerd om het wisselen tussen applicaties te implementeren d.m.v. RFID tags: met iedere RFID tag is een bepaalde applicatie geassocieerd.

Multiplayer: Twee LEDGRIDs

Dan was ook een draadloze netwerkcomponent vereist. Als je al weet dat we als bedoeling hadden om onze LEDGRID compleet zelfstandig te laten werken, is het niet moeilijk meer om de link te leggen naar multiplayer games. En dus wilden we *twéé* LEDGRIDs bouwen, waarmee je dan draadloos spelletjes tegen elkaar zou kunnen spelen.

LEDGRID Emulator

Naarmate het project vorderde, bleek dat het niet realistisch zou zijn om binnen de beperkte tijd die we hadden twee LEDGRIDs te bouwen (hoewel we wel over de hardware ervoor beschikten). Ook bleek het "programmeer, compileer, upload naar Arduino, wacht tot die is opgestart, test, herhaal"-proces niet erg handig te zijn. Zeker als we meerdere applicaties zouden gaan ontwikkelen, leek dat al snel (té) frustrerend en vertragend te gaan worden.

Daarom besloten we om ook een LEDGRID emulator te ontwikkelen. Op die manier kunnen we dan een tweede LEDGRID emuleren, zonder deze effectief te moeten bouwen.

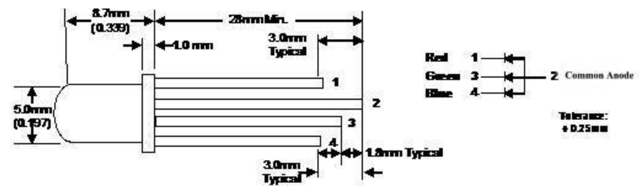

Figure 3. Het schema van onze RGB leds.

Ook werd het makkelijker om nieuwe applicaties te bouwen, want we voorzien exact dezelfde API in de emulator als op de fysieke Arduino bordjes beschikbaar is, maar in de emulator kan de debugger van een C++ IDE gebruikt worden.

DE HARDWARE

Onze eerste uitdaging was onderzoeken hoe we 64 RGB leds moesten aansturen. Want om één RGB led aan te sturen, zijn er 4 pinnen nodig: een gedeelde (*common*) anode (of cathode) en 3 cathodes (of anodes): één voor elk van de drie kleurcomponenten: rood, groen en blauw
De leds die wij kochten, werken met een common anode (zie figuur 3 voor het schema).
Voor iedere led zou er ook een knop moeten komen, maar dat bleek gelukkig eenvoudiger te integreren: er is slechts een enkele pin nodig per knop.

We bespraken ook alvast de form factor van de behuizing. We vonden het een leuk idee om plexiglas plaatjes (van 5 bij 5 cm) mat te maken en deze over een led te bevestigen. Zo zouden we met redelijke waarschijnlijkheid een even kleuren lichtopbrengst krijgen. De moeilijkheid zou vooral het contact van deze plexiglas plaatjes met de overeenkomstige knoppen worden: de plexiglas plaatjes zouden het drukoppervlak van de knoppen moeten worden.
Voor mij stopte het bij deze oppervlakkige bespreking, de verdere uitwerking van de behuizing is geheel het werk van Jens. Voor meer details verwijs ik daarom naar zijn paper.

Multiplexing

Indien we iedere led afzonderlijk zouden willen aansturen, zouden er op z'n minst $64 \times 4 = 256$ pinnen nodig zijn. Dat is natuurlijk absurd veel. Dit kan eenvoudig verminderd worden door gebruik te maken van multiplexing: er wordt dan telkens slechts één rij leds tegelijk ingeschakeld; er wordt dus één rij geactiveerd en de 8 kolommen RGB leds worden aangestuurd. We willen de 3 kleuren van ieder van deze RGB leds aansturen, en dus zijn daar $8 \times 3 = 24$ pinnen voor nodig.
Voor de RGB leds zijn er in totaal dus $8 + (8 \times 3) = 32$ pinnen nodig.

Voor de 64 knoppen zouden 64 pinnen kunnen nodig zijn, maar als we ook hier multiplexing op toepassen, kunnen dezelfde 8 pinnen die de rij van leds selecteren, ook gebruikt worden om de rij van knoppen te selecteren. Dan zijn er nog slechts 8 extra pinnen nodig.

Het eindtotaal is dus $8 + (8 \times 3) + 8 = 40$ pinnen (8 voor het activeren van iedere rij, 24 om de 8 RGB leds in de

kolommen aan te sturen en nog eens 8 voor de 8 knoppen per kolom).

Geen Arduino Duemilanove, maar een Arduino Mega

40 pinnen is dus het minimaal aantal benodigde pinnen in onze op multiplexing gebaseerde opstelling. Het basis Arduino bordje, wat op het moment van schrijven de Arduino Duemilanove is, heeft echter slechts 16 digitale I/O pinnen (en 6 analoge input pinnen, dus 20 pinnen in totaal) [1].

Ons opzet dwong ons dus om naar een geavanceerdere Arduino te kijken, wat ons bij de Arduino Mega [3] bracht, welke maar liefst 48 digitale I/O pinnen (en eveneens 6 analoge input pinnen) heeft.

PWM: Voldoende Pinnen?

De RGB-pinnen enkel aan en uitzetten (dus minimale en maximale intensiteit) geeft je enkel de mogelijkheid om de standaard combinaties met rood, groen en blauw te maken, zoals bijvoorbeeld paars, wat de combinatie van rood en blauw is.

Natuurlijk is het de bedoeling om ook andere kleuren te kunnen genereren — zoals bijvoorbeeld geel, wat met de bovenstaande beperkte setup niet mogelijk is! Daarvoor is het nodig om de intensiteit te kunnen variëren over meer stappen dan enkel minimaal en maximaal. Met andere woorden: het is nodig om iedere pin een fractie van de tijd aan- en uitte kunnen zetten. Dit wordt pulse width modulation (PWM) genoemd.
Ieder Arduino bordje heeft een aantal pinnen dat via de Arduino software makkelijk in PWM modus kan gezet worden, bij de Arduino Mega zijn dit er 14 van de 54 digitale I/O pinnen. We hebben 40 pinnen hiervan nodig, maar niet alle 40 moeten via PWM aangestuurd worden: de 8 pinnen om de rij te selecteren hebben dit niet nodig en de 8 pinnen om het induwen van knoppen te detecteren evenmin.
Dus $40 - 8 - 8 = (8 \times 3) = 24$ PWM pinnen zijn benodigd: één voor elk van de 3 kleuren van elk van de 8 gelijktijdig geactiveerde leds.

Zoals eerder vermeld, heeft de Arduino Mega echter slechts 14 PWM pinnen, terwijl er 24 nodig zijn. Daarom bleek het dus nodig te zijn om PWM te gaan simuleren in de software, door middel van getimede interrupts, zodat op regelmatige intervallen de leds kunnen aangestuurd worden.
Dit wordt verder uitgelegd in de *PWM Simulatie* sectie.

RFID tags lezen

Zoals eerder vermeld, was het de bedoeling om RFIDs te gebruiken om van applicatie te wisselen. Dit bleek érg triviaal te zijn om te implementeren: het was werkend op amper een kwartiertje. Het is gewoon een kwestie van de inkomende tag id te parsen en te checken of deze verschillend is van de vorige RFID. Voor deze zeldzame keer geldt *"it just works"*.

Details en Schema

In onze multiplexed LED setup maken we gebruik van 1 NPN transistor per kolom en 1 PNP transistor per kolom; wat dus resulteert in 8 NPN transistors en 8 PNP transistors.

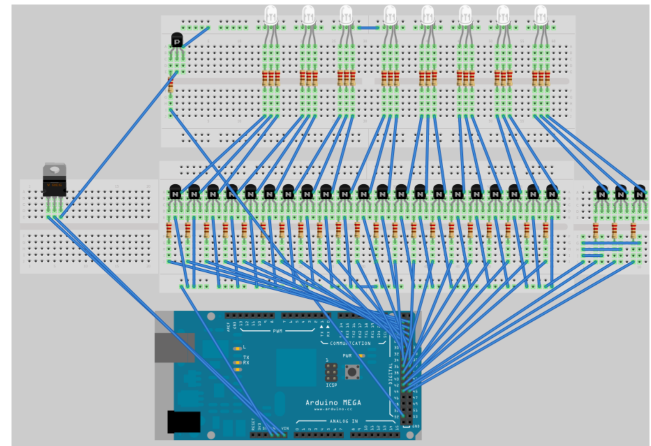

Figure 4. Het schema voor één rij van onze RGB leds.

Omdat meer stroomsterkte vereist is dan de Arduino Mega aankan, is een externe spanningsbron vereist. Om diens voltage naar 5V om te zetten, is een enkele spanningsregelaar nodig.
Immers, de ATmega1280 chip op de Arduino Mega kan slechts een maximum van 200mA aan [15] over alle pinnen samen (aangezien de DC Current VCC and GND pins maximaal 200 mA aankunnen). Wij hebben echter op ieder moment 8 RGB leds (1 rij) met elk 3 kleuren, dus 24 pinnen tegelijk aanstaan, wat $24 \times 20 = 480$ mA vereist.

Een schema van alle 8 rijen zou teveel plaats in beslag nemen in deze paper. Daarom staat in figuur 4 een schema voor één rij, waarbij voor de 8 rijen geen extra NPN transistors (voor de kolommen) moeten toegevoegd worden, maar enkel nog een extra PNP transistor moet toegevoegd worden voor iedere extra rij.

Deze beschrijving is nog steeds niet erg gedetailleerd. Dit is omdat ik me vooral op de software kant heb gericht — Jens heeft de hardware kant voor zijn rekening gehouden. Ik heb in het begin mee geredeneerd (voor voorbeelden daarvan, zie de secties over multiplexing en PWM), maar van zodra we één rij van onze RGB leds op een breadboard in de klas werkend hadden, heeft Jens de rest van de hardware kant op zich genomen. Zo konden we veel effectiever werken: de een alle hardware, de ander alle software. Beide aspecten hadden voldoende moeilijke uitdagingen om het op ongeveer hetzelfde moeilijkheidsniveau te brengen.
Voor een gedetailleerdere beschrijving van de hardware verwijs ik dus naar de paper van Jens.

SOFTWARE

Het algemene opzet is om van LEDGRID een heel eenvoudig (en natuurlijk beperkt) development platform te maken: zolang de developer zich beperkt tot de API die het voorziet, *"it all just works"* (of dat is toch het doel). Met als achterliggende gedachte natuurlijk dat als het eenvoudig genoeg is, snel nieuwe apps kunnen toegevoegd worden.

In het `configuration.h` bestand kan de configuratie voor

de gebruikte LEDGRID ingesteld worden. Het is mogelijk om er debug mode in te schakelen (waarbij dan extra informatie geprint via `Serial`), om het aantal rijen en kolommen in te stellen, het aantal keer dat iedere kleur van een led aan en uit kan gezet worden per refresh (de `INTENSITIES` define) en wat die refresh rate is (de `REFRESH_RATE` define). Ook kan er ingesteld worden welke pins welke functie krijgen.

PWM Simulatie

Het aantal interrupts per seconde wordt automatisch ingesteld a.d.h.v. de configuratie: NUM_ROWS × INTENSITIES × REFRESH_RATE $= 8 \times 6 \times 60 = 2880$. De betekenis van deze defines wordt verderop in de tekst uitgelegd.

MsTimer2

Jammer genoeg bleek het niet bepaald eenvoudig om een interrupt 2880 keer per seconde, oftewel om de $1.0/2880 = 0.00034722$ seconden[1]. Eerst probeerde ik op basis van het "Arduino Interrupts" artikel [16] interrupts aan het werken te krijgen. Die code was echter zodanig low-level en doorspekt van globale variabelen en defines waarover nergens uitleg te vinden was, dat het me na dagen geworstel tot complete wanhoop dreef.

Met hernieuwde moed consulteerde ik daarna de officiële lijst met Arduino libraries [2] en leek de `MsTimer2`[2] library [11] de beste keuze om mee van start te gaan — met de grote beperking dat deze slechts 1000 maal per seconde een interrupt kan afvuren. Met `MsTimer2` kreeg ik interrupts metéén werkend.

FlexiTimer2

Daarna stortte ik me op het flexibeler maken van `MsTimer2`: het zou mogelijk moeten zijn om de gewenste resolutie van de timer interrupts in te stellen. Daarom heb ik een fork van `MsTimer2` gemaakt, met de naam `FlexiTimer2`[3]. Daarbij is het dus mogelijk om een willekeurige resolutie in te stellen.

Aangezien de code van `MsTimer2` onder de LGPL is vrijgegeven, was de meest logische keuze om deze nieuwe library onder dezelfde licentie te publiceren. De code is gehost op GitHub [7]. Ik heb ook een aanvraag gedaan om dit aan de officiële lijst met Arduino libraries [2] toe te voegen [8]. Op het moment van schrijven wordt dit nog geëvalueerd, maar `FlexiTimer2` is wel al te vinden in de *Arduino Playground*.

Kleurbereik

Nu het dus mogelijk was om een willekeurig aantal interrupts in te stellen, was het tijd geworden om te bepalen bij welke van die interrupts iedere led aan of uit zou moeten staan, om zo dus een redelijk kleurbereik te behalen. Zoals eerder vermeld, kan het aantal keer dat iedere kleur van een led aan en uit kan gezet worden per refresh (de `INTENSITIES` define) en wat de refresh rate is (de `REFRESH_RATE` define) ingesteld worden. De `INTENSITIES` define is ingesteld op 6, wat betekent dat 6 keer per refresh de kleur van de led aan of uit kan gezet worden. Dit levert 7 mogelijke combinaties op, want de kleur van de led kan dus ingesteld worden om 0, 1, 2, 3, 4, 5 of 6 keer aan te staan. Een RGB led heeft 3 kleuren, dus zijn er $7^3 = 343$ verschillende kleuren mogelijk.

Om flikkeren tegen te gaan, is het belangrijk om een zo gelijkmatig aan/uit patroon te gebruiken: als de led 50% ($\frac{3}{6}$) van de tijd moet aanstaan, kan er gekozen worden voor het eenvoudige patroon 111000 of voor het complexere patroon 101010. Het laatste gaat uiteraard een gelijkmatiger resultaat geven. Een ander voorbeeld is $\frac{4}{6}$, het beste patroon hiervoor is 1011011: wanneer dit herhaald wordt, resulteert dat in een perfect gelijkmatig patroon.

Deze patronen worden gegenereerd in de `generateModulos()` functie — sommige patronen kunnen automatisch gegenereerd worden, anderen zijn handmatig gedefinieerd.

Gevolg van het Gebruik van (Timer) Interrupts

Er is een belangrijk gevolg van het gebruik van interrupts: zolang de code van de ISR[4] wordt uitgevoerd, is het niet mogelijk om in de ISR's van andere interrupts te gaan. Concreet betekent dit dat seriële communicatie, welke ook gebruik maakt van interrupts (telkens wanneer een byte aan data ontvangen is), *mogelijk* niet langer als betrouwbaar kan beschouwd worden.

In deze paragraaf wordt ruwweg uitgelegd hoe een bit verloren kan gaan. Bijvoorbeeld bij een Baud rate van 9600, worden 9600 bits oftewel 1200 bytes per seconde doorgestuurd. Bij een hogere Baud rate, wat toelaat om tijd-kritische data sneller door te sturen (in ons geval bijvoorbeeld de kleuren van alle leds in de LEDGRID), volgen de bytes elkdaar dus ook sneller op.

Wij voeren echter 2880 keer per seconde onze `updateLeds` ISR uit die naar de volgende rij springt en de pins van die rij updatet — en iedere 20ste keer wordt bovendien de toestand van iedere knop uitgelezen (want iedere knop $2880/8 = 360$ keer per seconde checken is zinloos: bij iedere 20ste update is dat $2880/8/20 = 18$ keer per seconde, wat betekent dat enkel indien de knop minder dan 55 ms ingeduwd wordt, het mogelijk is dat het niet gedetecteerd wordt). Omwille van de uitvoeringstijd van `updateLeds` (die dus variabel is) kan het zijn dat, gedurende de hele tijd dat een byte beschikbaar was via seriële communicatie, `updateLeds` werd uitgevoerd. Met andere woorden: het kan zijn dat sommige bytes niet verwerkt konden worden en dus "overgeslagen" werden.

Uit tests bleek dat zelfs bij een Baud rate van 9600 er occasioneel fouten konden optreden. Dit kan enkel opgelost worden door te garanderen dat de `updateLeds` ISR minder tijd in beslag neemt dan de tijd gedurende dewelke een byte van de seriële communicatie beschikbaar is.

Het is mogelijk dat dit doel haalbaar is door *direct port manipulation* [4] te gebruiken i.p.v. de `digitalWrite()` en

[1] Dit wordt ook wel een *resolutie* van 0.00034722 seconden genoemd.

[2] De '2' in `MsTimer2` verwijst naar het feit dat timer 2 van de Arduino chip wordt gebruikt om een interrupt te triggeren. De 'ms' in de naam verwijst naar 'milliseconde'.

[3] Alluderend naar het feit dat de resolutie van deze library flexibel is, in tegenstelling tot een vaste resolutie van één milliseconde.

[4] Interrupt Service Routine

digitalRead(), welke een grootte orde trager zijn: ze bestaan immers elk uit ongeveer een dozijn regels code versus een enkele machine instructie.
Uiteindelijk ben ik gestoten op de DigitalWriteFast library [6], welke direct port manipulation toestaat op zowel de Arduino Duemilanove als de Mega, maar dan met een high-level syntax. Deze bestaat uit een set van macro's[5] die "functieaanroepen" omzetten naar direct port manipulation commando's. Het nadeel is echter dat geen variabelen kunnen gebruikt worden in aanroepen naar DigitalWriteFast. Om dit dus effectief te gebruiken, zou de code hardcoded moeten afgestemd worden op het gebruik van een 8x8 LEDGRID én zouden er in de updateLeds ISR enorm veel if-statements moeten komen: tweemaal het aantal pinnen (iedere pin moet aan en uit kunnen gezet worden). Dit is natuurlijk verre van elegant en impliceert nog steeds behoorlijk wat overhead.
Dus bleef enkel "échte" direct port manipulation over als optie. Omwille van het verlies van portability (de te gebruiken poorten zijn afhankelijk van de processor op het Arduino bordje), het complexer maken van de code van de ISR (er moeten telkens dynamisch bitmasks gegenereerd worden, welke toegekend moeten worden aan de poort registers) en het pas laat ontdekken dat dit corruptie van de data via de seriële lijn veroorzaakt, hebben we ervoor gekozen om direct port manipulation niet meer te implementeren.

API
Om de leesbaarheid van de code te verhogen, zijn er enkele defines en een structure voorzien (de Arduino 0018 IDE bleek niet om te kunnen gaan met typedefs): ColorValue, RGB, Row, Col, struct ColorValues {ColorValue red; ColorValue green; ColorValue blue; }.

De API zelf is zoals gezegd zeer eenvoudig en beperkt:

- `ColorValue getColor(Row row, Col col, RGB rgb);`
- `struct ColorValues getColors(Row row, Col col, RGB rgb);`
- `void setColor(Row row, Col col, RGB rgb, ColorValue color);`
- `void setColors(Row row, Col col, struct ColorValues colors);`
- `void resetColors(Row row, Col col);`

Om ingeduwde knoppen te detecteren moet er naar de overeenkomstige waarde in de buttons[NUM_ROWS][NUM_COLS]; variabele gekeken worden — uiteraard betekent een true waarde dat de knop ingeduwd was tijdens de laatste check.

Apps
Iedere app heeft een functie die tijdens iedere aanroep van Arduino's loop() functie wordt aangeroepen. Een app functie moet dus inkomende data verwerken, ingeduwde knoppen detecteren, zijn logica uitvoeren, het scherm updaten, zijn toestand bijhouden en vervolgens beëeindigen.

[5]Anders zou er immers nog steeds onnodige overhead zijn o.w.v. function calls.

Deze aanpak maakte het makkelijk om het dynamisch wisselen van applicaties te ondersteunen.

Hier vind je de volledige lijst van apps, telkens met een zéér beknopte beschrijving. Ze werden door mij geschreven zijn tenzij anders vermeld:

- *None* — doet helemaal niets
- *Remote Control* — verzendt de "schermdata"
- *Remotely Controlled Display* — toont de "schermdata"
- *Bomberman* — alomgekend, tot 4 spelers tegelijk (Jens)
- *Game of Life* — ingeduwde knop is "nieuw leven" (Jens)
- *HSV color wheel* — itereert over het hele kleurengamma
- *Numbers* — duw op een knop en zijn getal wordt getoond
- *Paint* — zeer eenvoudige tekenapplicatie
- *Pong* — een kleurrijke implementatie van een klassieker

Remote Control/Remotely Controlled Display
Deze apps (de ene is een zender, de andere een ontvanger) verdienen extra uitleg.
RC (Remote Control) krijgt een speciale behandeling: deze kan "gelijktijdig" met een andere app worden uitgevoerd, d.w.z.: telkens na een aanroep van de functie van de actieve app wordt RC aangeroepen en verzendt deze de kleuren van de leds wiens kleuren veranderd zijn. Per veranderde led is 2 bytes (16 bits) aan data nodig, telkens 3 bytes voor: de rij, de kolom, de intensiteit van iedere kleur (waarde in het interval $[0, 7]$). Dat geeft 15 bits in totaal, met dus nog 1 irrelevante bit die genegeerd wordt.
De RC Display app werkt wel als een gewone applicatie en werkt dus als een "view-only VNC sessie" naar de andere LEDGRID. Telkens wanneer klaar is het met het tonen van het huidige "scherm", vraagt hij aan de RC app op de andere LEDGRID naar de data voor het volgende "scherm".

Pong
Bij het opstarten van de Pong app worden een grote "1" en "2" getoond. Voor "1" kiezen, betekent dat Pong op één LEDGRID zal gespeeld worden. En voor "2" kiezen, betekent dus dat Pong op twéé LEDGRIDs gespeeld zal worden (dus eigenlijk worden dan de fysieke en een geëmuleerde LEDGRID gebruikt). De speler die met de linker *paddle* zal gaan spelen wordt de master, de andere de slave. De master berekent de positie van de bal en stuurt de data door van zijn paddle (de linker paddle) en de positie van de bal. De slave stuurt tevens de data door van zijn paddle (de rechter) en doet geen berekeningen, maar toont enkel de positie van de bal die het krijgt van de master. Wegens tijdsgebrek hebben we dit echter nog niet honderd procent werkende gekregen.

LEDGRID EMULATOR
Apps zouden zonder enige aanpassing meteen in de LEDGRID emulator moeten kunnen worden uitgevoerd. Aangezien de apps geschreven zijn voor Arduino en dus in C++, kon de emulator best ook in C++ geschreven moeten worden.

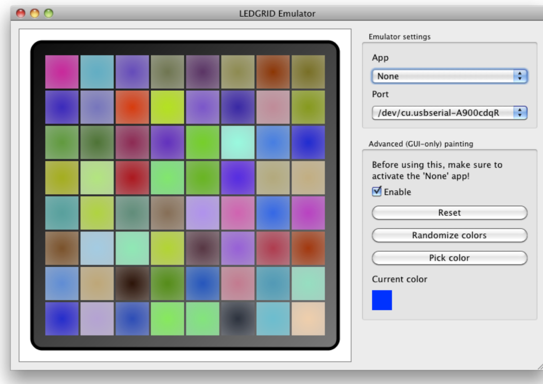

Figure 5. LEDGRID emulator.

Daarom heb ik gekozen voor C++ i.c.m. Qt. Voor een screenshot van het resultaat, zie figuur 5. De emulator is getest in zowel Mac OS X 10.6 als in Windows XP.

Eerst werkte ik aan het grafische gedeelte: een visueel aangename voorstelling van LEDGRID, die de realiteit zou moeten benaderen. Daarom wordt iedere ledknop met een gradiënt gerenderd, die het ietwat heldere centrum van de knop ook helderder rendert. Er wordt gebruik gemaakt van anti-aliasing om het geheel er naadloos te laten uitzien.
Vervolgens bouwde ik de emulator, waarbij de door ons voorziene API om de LEDGRID aan te sturen geëmuleerd werd.
Ten slotte moest het ook mogelijk zijn om van een geëmuleerd equivalent aan Arduino's `Serial` library gebruik te maken, dus voegde ik ook daarvoor ondersteuning toe. Om dit te implementeren werd gebruik gemaakt van de `QextSerialPort` [13] library. Ook van deze library (met BSD licentie) heb ik een fork gemaakt die op GitHub beschikbaar is [14], dit keer niet om nieuwe functionaliteit te voorzien, maar om het buildproces te verbeteren en de library automatisch te kunnen installeren (d.m.v. `make`).

De GUI heeft ook de mogelijkheid tot "geavanceerd tekenen": tekenen met het gehele kleurenpalet en de mogelijkheid om random kleuren te genereren voor alle leds.

Een van de grootste moeilijkheden was het includen van zoveel mogelijk code die al voor de "echte" Arduino geschreven was, en het omgaan met includes en de globale variabelen die gebruikt worden door die code. Een andere grote moeilijkheid was het aan de praat krijgen van de `QextSerialPort` library en daarna nog eens om deze ook na deployment werkende te krijgen (vandaar de fork).

De zeer beknopte inhoud en beperkte lengte van deze sectie staan niet in verhouding tot de hoeveelheid werk die er is ingestoken: dit heeft minstens evenveel tijd gekost als de pure Arduino code!
Wanneer ik de `Ohcount` [12] source code line counter op de broncode los liet, bleek dat de pure Arduino code zo'n 1400 regels code bevat (exclusief white space of comments). De Arduino Emulator bleek eveneens uit ruim 1400 regels code te bestaan.

CONCLUSIE

We zijn er in geslaagd om een functioneel apparaat te maken dat voldoet aan onze vooropgestelde eisen. We hadden graag nog meer applicaties geschreven om het potentieel nog beter in de verf te zetten, maar wegens tijdsgebrek moesten we daar van afzien.
Met meer tijd hadden we ook de `updateLeds` ISR beter kunnen optimaliseren, wat mogelijk de seriële communicatie vrij van corruptie had kunnen maken.
Over het algemeen zijn we zeer tevreden met het resultaat, niet in het minst omdat de complexiteit veel hoger bleek te zijn dan we bij aanvang verwachtten.

REFERENCES

1. Arduino duemilanove. `http://arduino.cc/en/Main/ArduinoBoardDuemilanove`.
2. Arduino libraries. `http://arduino.cc/en/Reference/Libraries`.
3. Arduino mega. `http://arduino.cc/en/Main/ArduinoBoardMega`.
4. Arduino port manipulation. `http://www.arduino.cc/en/Reference/PortManipulation`.
5. Arduinome. `http://flipmu.com/work/arduinome/`.
6. Digitalwritefast. `http://code.google.com/p/digitalwritefast/`.
7. Flexitimer2 — project page. `http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1273777987/0`.
8. Flexitimer2 — request to include on library references page. `http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1273777987/0`.
9. Monome. `http://monome.org/`.
10. Monome modellen. `http://monome.org/devices`.
11. Mstimer2 — project page. `http://www.arduino.cc/playground/Main/MsTimer2`.
12. Ohcount. `http://www.ohloh.net/p/ohcount`.
13. Qextserialport. `http://code.google.com/p/qextserialport/`.
14. Qextserialport fork. `http://github.com/wimleers/qextserialport`.
15. *ATmega1280 data sheet, p 370*. Atmel, 2325 Orchard Parkway, San Jose, CA 95131, United States of America, 2001.
16. D. Fowler. Arduino interrupts, 11 2007. `http://www.uchobby.com/index.php/2007/11/24/arduino-interrupts/`.