SDL, OpenGL och grafiska client-serverapplikationer

Här ges en översikt av verktyg och tekniker som ni kan ha nytta av när ni skapar grafiska client-serverapplikationer.

Ett förtydligande dock, alla behöver inte förstå allt som presenteras här. Det som ges här ges inom ramen för en projektkurs, det finns alltså ingen tentamen att plugga in detta till. Vidare är de tekniker som presenteras här inte så avancerade, men de kan vara något komplicerade att förstå sig på. Speciellt används ganska så intrasslade datastrukturer, vi har, på ett ställe, en array som innehåller structar som innhåller structar, som i sin tur också innehåller structar. För att komma åt vår data måste vi då skriva

```
array[index].struct1.struct2.struct3.data
```

det här är egentligen inte avancerat, men det är lite rörigt. Det blir lite avancerat när vi också skickar detta som parametrar till funktioner i form av pekare och när vi också ska skicka adresser av sådana här konstruktioner till biblioteksfunktioner, och som `void *` till trådfunktioner, vi ser då konstruktioner av typerna

```
&array[index].struct1.struct2.struct3.data
```

och

```
struct1->struct2.struct3.data
```

men, återigen, det är egentligen inte så avancerat, det blir lättare att förstå när man väl greppat grundprinciperna..

Jag baserar denna presentation på flera källor som det är bra att studera om man tycker att innehållet här är bra:

> Beej's Guide to Network programming: det finns flera exempel där på hur man skickar data över både TCP och UDP förbindelser i ett client-server sammanhang.
>
> Information från webben om hur SDL fungerar över nätet, “SDL_Net”
>
> Information från webben om hur SDL och OpenGL fungerar lokalt.

Tonvikten ligger inte på att gå igenom exakta detaljer om hur man får igång SDL och OpenGL på en plattform och exakt vilka funktionsanrop man ska göra för att åstadkomma olika saker i SDL och OpenGL, sådana saker lämnas åt gruppernas egna faktainsamlingar. Vidare ges strategiskt konstruerade programexempel (som finns på bilda). Dock är dessa programexempel inte tänkta som programvara som ni ska använda i era projekt, de är bara tänkta att beskriva funktionssättet. Inga av programmmen är trådsäkra eller minnestäta och de har inte heller så god struktur. Alla dessa saker måste ni ta hand om: ni måste skriva trådsäkra program som inte läcker minne och som också är väl strukturerade. Den strukturerade varianten av programmet MouseControlledSquares är däremot ett exempel på ett program som är strukturerat, det, jämte Anders Lindströms exempel.

Tonvikten här kommer att ligga på att ge en översikt av hur man kan tänka när man utvecklar nätbaserade grafiska program.

1. Förra årets material

Förra årets material bestod i enkla introduktioner till SDL och OpenGL. Vi studerar början av det materialet i programmen

[Studera `0_staticball`, `1_eventsball`, `2_movingball`, `3_opengl`]

[Poäntera den strukturerade varianten av `3_opengl`.]

2. Mer av förra årets material som vi inte studerar.

Inför förra året satte jag ihop ett enkelt material där en klient kontaktar en server. Användaren klickar på en punkt på sin fönsteryta, dessa koordinater skickas till servern som skickar tillbaka 10 slumpvis valda koordinater i anslutning till de koordinater som angivits. Klientprogrammet ritar då ut kvadrater på dessa 10 punkter. Inte ett så upphetsande program kanske, men det intressanta här är att vi separerar databehandling på två olika värdar (klient och server), kvadraters positioner slumpas ut på servern men ritas ut på klienten. Detta program hade inte stöd för flera klienter samtidigt där de respektive klienterna såg varandras skärmar. Ni får själva provköra detta program.

3. Det nya för i år: mer detaljerad implementation av en grafisk client-server applikation

Inför detta år har jag skrivit ett mer komplett program bestående av en klient och en server. Varje klient har en kvadrat som rör sig över skärmen. Flera klienter kan ansluta och varje klients kvadrat syns på alla andra klienters skärmar. Programmet är av typen "quick and dirty" och är absolut inte så värst väl strukturerat – jag är inte stolt över det, men jag tror att ni kan ha användning av det. Ni ska alltså inte inspireras av detta program mer än att ha en fungerande konstellation av en grafisk client-server applikation.

Servern använder POSIX-trådar och BSD-sockets för IPC. Klienten använder SDL_net för IPC och SDL för eventhantering samt OpenGL för grafiken. Vi tittar på en demonstration.

[Demonstrera]

Musklick skickas med TCP och de nya positionsangivelserna skickas med UDP. Där finns då ett applikationsprotokoll som defineras av följande headerfil:

```
#ifndef PROTOCOL_H
#define PROTOCOL_H

#define MAX_SQUARES 3

#define RED 1
#define BLUE 2
#define YELLOW 3
```

```
//Client requesting entry.
//Sent via TCP.
struct csm_request_entry
{
  int requested_color;
};

//Client notifying server of a click.
//Sent via TCP
struct csm_click_at
{
  float x, y;
};


// Server granting entry to a user.
// Sent via TCP. (-1 indicates failure.)
struct scm_grant_entry
{
  int success;
};

// Server requesting client to draw a square at x, y
// Previous coordinates sent for the clients to remove
// old image during animation
// Sent via UDP.
struct scm_draw_square_at
{
  float x, y;
  float prevx, prevy;
  int square_color;
};

#endif
```

Ett meddelande som skickas i någon riktning beskrivs alltså av en struct och det finns fyra olika sorters meddelanden, alltså fyra olika sorters structar.

Structarnas innehåll omvandlas alltid till strängar och man kommunicerar över nätet i form av strängar. Detta är en säkerhetsåtgärd för att öka allmängilitgheten hos ett program. En sträng är en följd av characters, medan en struct kan hamna lite anpassat på olika sätt hos olika kompilatorer. (Den tekniska termen för detta är *alignment* och det är alltså någonting vi vill undvika.)

Vi skapar också structar här för att betona att en structs definition kan förändras vartefter vi utvecklar programmet. Det är ingen lätt sak att ändra i `protocol.h`, eftersom allting kommer att bero på den, men om vi gör structar från början har vi i alla fall förberett för förändring.

Programmet har följande uppbyggnad:

1. En global array (`very_ugly_array`) med uppgifter om hur man skickar saker per UDP till alla klienter. Denna har gjorts global på grund av tidsbrist (jag skrev klart detta program igårnatt). Den innehåller allt alla behöver för att kunna skicka UDP-paket till alla klienter. En tillhörande initieringsrutin finns med.

2. En deklaration av en struct som håller reda på data tillhörande en klients anslutning. Den heter `struct square_t` och kanske skulle byta namn till client eftersom den lagrar mer saker än bara uppgifter om en kvadrat. Det finns också en tillhörnade initieringsrutin
.
3. En funktion för att hitta en ledig plats för en klientanslutning, den går igenom alla klientanslutningar (square_t-arrayen) och tittar efter en ledigflagga (free) som är satt så platsen är ledig.

4. En funktion som heter `prepare_udp_stuff_in_very_ugly_array`, detta är programmets mest komplicerade funktion att förstå, den har ett fruktansvärt namn och ett fruktansvärt syfte, att skapa innehållet i den globala arrayen (`very_ugly_array`). Varje gång en klient ansluter måste andra klienter ges möjlighet att skicka UDP-meddelanden till den, därför sparas alla udp-uppgifter i den globala arrayen. Alla klient-trådar som skapas för att betjäna en klient går till denna array för att hitta UDP-uppgifter så att de kan skicka till alla andra klienter. Den här är baserad på UDP-programmet talker i Beej's Guide to Network programming, och här generaliseras mjukvaruarkitekturen från Beej så att vi kan kommunicera med flera parallella klienter samtidigt.

5. Funktionen `move_square` är den trådfunktion som sköter själva uppdaterandet av representationsformerna för att uppnå flyttandet av en kvadrat. Denna trådfunktion skickar sedan data till alla klienter (med hjälp av `ugly_array` och UDP) och så får varje klient uppdatera sina grafiska representationer av att alla kvadrater har flyttat sig.

6. Funktionen `do_square` är en annan trådfunktion som sköter själva uppfångandet av klickningar från klienterna. Observera då att varje enskild klient kör på olika maskiner och att funktionen `do_square` körs i dialog med en specifik klient. Det är en tråd per klient och den deskriptor som man använder för kommunikationen lagras i arrayen av `square_t`, den heter `csd` som är förkortning för *client socket descriptor*.

7. Sedan följer en funktion och en trådfunktion för debugging-utskrifter av aktuella data.

8. Sist följer huvudprogrammet som egentligen bara startar alla trådar.

Ni får experimentera så mycket ni vill med detta program, man kan också välja att programmera i SDL på både klient och serversidan, men jag tycker bättre om POSIX så jag valde SDL för klientsidan och POSIX för serversidan. Den konkreta anledningen till detta var att `accept` i SDL inte blockerade och jag visste inte hur jag skulle få den att blockera.

Som nämnt tidigare är programmet varken strukturerat, trådsäkert eller minnestätt.