Desenvolupament d'un mòdul interactiu de visualització d'àudio en temps real

Ignasi Casasnovas Català

Projecte Fi de Carrera
Director: Sergi Jordà Puig
Desembre 2004
Enginyeria en Informàtica
Universitat Pompeu Fabra

A la meva família per haver-me recolzat en tot moment, sense ells no hauria arribat fins aquí
A Sergi Jordà per haver-me dirigit en aquest projecte i haver confiat en mi
A Martin Kaltenbrunner i a Günter Geiger pel seu inestimable ajut, i també al MTG
A tots els bons amics que he pogut conèixer a la universitat i també als que ja tenia

A tots vosaltres, gràcies

Resum

El projecte consisteix principalment en el disseny del mòdul de visualització d'un instrument musical electrònic, anomenat reacTable*. El projecte reacTable* s'està desenvolupant al Grup de Tecnologia Musical (MTG) de l'Institut Universitari de l'Audiovisual de la Universitat Pompeu Fabra. En la resta del projecte es farà un estudi de les visualitzacions d'àudio.

La reacTable*, què està sent desenvolupada pel grup de sistemes interactius dirigit per Sergi Jordà, és un instrument musical electrònic amb interfície tangible i feedback sonor i visual. Aquest projecte es centra en el que seria el feedback visual de l'instrument. S'haurà de crear un mòdul que generi imatges en temps real que reflecteixin l'activitat del sistema a la vegada que contribueixin a la seva interactivitat. Per tant, es dissenyarà el que es pot anomenar una visualització d'àudio interactiva.

Tot això pel que fa a la implementació, però, evidentment, hi haurà una labor prèvia de documentació i investigació, on farem un repàs històric de les representacions gràfiques del so, tant en el domini computacional, com en l'era prèvia als computadors, i analitzarem la relació imatge/so; fins arribar a l'estat d'aquesta línea d'investigació avui dia, explorant la situació actual tant pel que fa als visualitzadors interactius com pels que no ho són.

Índex

1. Introducció

1.1 Context tecnològic

L'evolució constant de la tecnologia digital ens ha conduït a un punt on la potència dels ordinadors personals és tan gran que permet la gestió de continguts audiovisuals a temps real, fins al punt que actualment un ordinador personal és capaç de llegir un fitxer d'àudio comprimit en mp3, descomprimir-lo, reproduir-lo, i generar un seguit d'imatges en concordança amb el so del fitxer tant sols utilitzant menys d'un 20% dels recursos del seu processador.

La gestió de l'àudio en els ordinadors ha experimentat un canvi enorme en els últims anys, actualment és possible realitzar síntesi de sons d'alta complexitat en ordinadors personals a temps real. Però, evidentment, l'àudio no ha estat l'únic camp de la informàtica en que s'ha progressat. El tractament de la imatge, segurament, ha millorat tant o més que el tractament d'àudio, fins el punt que es poden crear imatges amb una semblança increïble a la realitat en temps real. Fet que és pot apreciar simplement observant els gràfics de qualsevol dels videojocs d'última generació.

Existeix una gran quantitat de software dedicat a la síntesi i/o a la manipulació del so, així com a la visualització del so, es a dir, generació d'imatges a partir del so o fins i tot en alguns casos el contrari. Cada cop aquests programes generen sons més complexes, més reals o més curiosos, però, a la vegada, les seves interfícies es van tornant més complicades i més inaccessibles per molts dels possibles usuaris. El repte es troba, doncs, en la creació d'un sistema d'àudio que aprofiti tots els avanços en el tractament d'àudio, però que a més ho faci mitjançant una interfície intuïtiva, fàcil de manejar i que al mateix temps permeti accedir a tots aquests avanços.

1.2 El projecte reacTable*

Es tracta d'un projecte que esta sent desenvolupat pel grup de Sistemes Interactius Sònics del Grup de Tecnologia Musical (MTG) de la Universitat Pompeu Fabra. La idea principal del qual és la creació d'un instrument musical, electrònic i tangible aprofitant tota l'experiència que el MTG ha anat acumulant al llarg dels anys i tenint com a precedent al FMOL [1], del que ja parlarem mes endavant. L'instrument ha de permetre expressar-se musicalment tant als musics professionals com a persones sense coneixements de música sense les limitacions marcades per les interfícies típiques. És a dir, ha de ser a la vegada complex, intuïtiu i ha d'oferir un ampli espectre de possibilitats.

Com podem deduir a partir del seu nom, la reacTable* és un instrument basat en una taula. Aquest instrument es fa sonar manipulant un conjunt d'objectes que tindrem distribuïts a sobre d'aquesta taula, cadascun d'aquests objectes té una funció assignada. Tenim, doncs, objectes generadors del so (oscil·ladors, generadors de soroll blanc, *samplers*...), modificadors del so (filtres, efectes, envolupants...) o objectes de control (*LFOs*, *triggers*...). Així, els objectes varien les seves propietats (freqüència, amplitud...) en funció de la seva posició a la taula, de la seva orientació, etc. evidentment, els objectes es poden connectar uns amb els altres creant així sons més complexes. Com a resultat de la nostra interacció amb la taula, obtenim òbviament un feedback sonor, que és el so produït per l'instrument, però, a més, la reacTable* proporciona un feedback visual projectant sobre la taula els fluxos sonors i de dades de control, així com aures que envolten els objectes actius reflectint tota l'activitat del sistema.

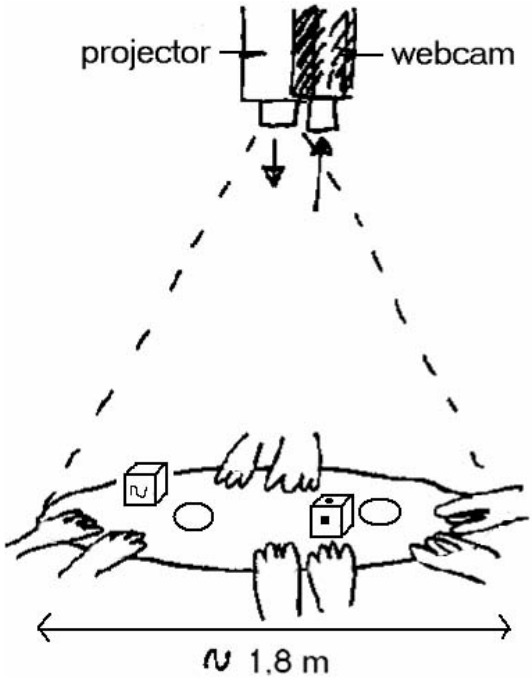


Figura 1. Primer esbós de la reacTable.

1.2.1 Objectius de la reacTable*

Els objectius de la reacTable* són bastant clars. El sistema ha de ser totalment intuïtiu i fàcil d'aprendre, no tindrem cap tipus d'instruccions, ajudes o displays alfanumèrics. Ha de ser interessant i competitiu, és a dir, l'usuari ha de ser capaç de crear nous sons i experimentar amb ells, a més aquest usuari pot ser tant un músic professional com una persona normal. La reacTable* ha de respondre bé tant en instal·lacions públiques com en concerts.

La interacció entre el músic i l'instrument ha de ser natural en tot moment, no hi ha d'haver botons, ni cables, ni el músic ha de portar cap dispositiu. La interacció es realitzarà sempre mitjançant la manipulació dels objectes que hi ha sobre la taula. Tampoc hi ha d'haver *presets* ocults o comportaments predefinits.

L'instrument permetrà que múltiples usuaris emprin la mateixa taula al mateix temps, per tant, serà col·laboratiu. Com a objectius més llunyans tenim també l'ús de telepresència i la interconnexió entre diferents taules.

1.2.2 Arquitectura bàsica

A grans trets podem dividir la reacTable* en un conjunt de components, on cada un té una missió específica. Així, per començar, direm que l'instrument es serveix de dos sintetitzadors: un sintetitzador de so, que s'encarrega de generar l'experiència musical actual i un sintetitzador gràfic, que és l'encarregat de generar el feedback visual basat en el so generat i en l'*input* de l'usuari.

D'altra banda, tenim també un component encarregat d'extreure i processar la informació dels sensors mitjançant tècniques de visió per ordinador i que proporciona al sistema la posició dels objectes, la orientació...

Tenim també el component lògic del sistema, que processa les entrades rebudes pel gestor dels sensors, gestiona els dos sintetitzadors, genera els *patches* [4] dinàmicament... en definitiva s'encarrega de tota la lògica interna del sistema. Finalment, hauríem de considerar com un component del sistema la interfície tangible, és a dir, la taula, el conjunt d'objectes així com el hardware utilitzat.

1.2.3 Equip de desenvolupament

Com s'ha esmentat anteriorment, el projecte reacTable* s'està desenvolupant al grup de Sistemes Interactius Sònics del Grup de Tecnologia Musical (MTG) de la Universitat Pompeu Fabra. El cap d'aquest grup és Sergi Jordà i és ell qui dirigeix aquest projecte del qual n'és l'ideòleg. L'equip queda constituït per les següents persones:

- Sergi Jordà: Cap del grup, ideòleg.
- Martin Kaltenbrunner: Encarregat de la HCI (interacció humà - ordinador), de la visió i de la integració del sistema.
- Günter Geiger: Encarregat del motor de síntesi de so.
- Ignasi Casasnovas: Encarregat del motor de síntesi de gràfics.
- Col·laboradors: Ruben Hinojosa, Álvaro Barbosa, Gerda Strobl.

1.3 Motivació del projecte

A simple vista es pot veure que la reacTable* és un projecte força ambiciós que pretén d'alguna manera oferir als seus usuaris la possibilitat de crear música servint-se dels avenços de la música electrònica, sense caure en la utilització d'una GUI tradicional, millorant la interacció amb el sistema fent-la més natural, intuïtiva i amb menys limitacions.

Al tractar-se d'un instrument, el feedback és un punt molt important, ja que és imprescindible per un músic que el seu instrument li doni una resposta a les seves accions a temps real. Aquí tenim per una banda el feedback sonor sense el qual no es tractaria d'un instrument musical, i el feedback visual, que en el cas de la reacTable* n'és un component fonamental. De fet, en les primeres proves amb l'actual prototip s'ha comprovat que el feedback visual és crucial a l'hora de tocar l'instrument.

Així doncs, el meu projecte té com a motivació principal la de proporcionar un feedback visual adequat a la reacTable*, és a dir, desenvolupar un mòdul de visualització d'àudio ajustat als requeriments de la reacTable*. Es tractarà de dissenyar una aplicació que projecti sobre la taula imatges que mostrin el fluix existent entre els diferents objectes, aures que envoltin els objectes actius... en definitiva, es tracta de representar l'activitat del sistema.

Per tant l'objectiu primari del projecte serà el disseny d'aquest mòdul de visualització, però el projecte té també altres objectius: intentarà oferir un anàlisi de les visualitzacions d'àudio existents, tant interactives com no, i farà un petit repàs a la historia dels sistemes que tracten amb la relació imatge/so.

2. Context històric

2.1 Sistemes visuals musicals anteriors als computadors

La relació entre la imatge i el so, contràriament al que molta gent pensa, ha estat estudiada des de fa segles. De fet, la sincronia entre el so i les imatges abstractes, coneguda entre d'altres maneres com música visual, té una història que s'expandeix al llarg de varis segles. Així i tot, la majoria de les persones no en són en absolut conscients, cosa que ja passava als inicis del segle passat. Klein va escriure a l'any 1927: "És estrany que quasi cada persona que desenvolupa un orgue de colors ho fa pensant equivocadament que és el primer mortal que ho intenta fer."[5], això ens demostra que fa molt temps que es tracta la relació imatge/so, i és que, de fet, són molts els que hi han treballat.

Pel que s'ha pogut esbrinar, el primer dispositiu que va permetre fer música visual fou l'*Ocular Harpsicord*, que va ser construït el 1734. Es tracta d'un instrument desenvolupat per Louis-Bertrand Castel (1688-1757), que ajuntava l'acció d'un clavicèmbal tradicional amb un display de cintes de paper transparent. Els colors d'aquestes cintes, segons Castel, tenien una relació directa amb cadascuna de les notes de l'escala musical occidental. El seu objectiu era fer visible el so per poder oferir als ulls els plaers que la música dona a les orelles. Així el disseny consistia en una pantalla que es posava a sobre d'un clavicèmbal normal, on cada una de les seves tecles activava, per mitjà d'un sistema de politges, un sistema que feia que la llum d'una candela filtrada per cintes de colors es projectés en aquesta pantalla. Així i tot la idea de Castel no va convèncer als seus contemporanis, que la trobaven realment estranya, però va servir d'inspiració en els segles posteriors. La major part d'aquesta informació i de la relatada a continuació la podem trobar en [6], [7] i [8].

Altres inventors varen seguir els passos de Castel. Un, més d'un segle després, al 1859, Frederic Kastner va dissenyar el *Pyrophone*, que utilitzava gas inflamable i tubs de cristall per generar tant imatge com so. El va seguir Bainbridge Bishop, al 1877, que tocava l'orgue mentre produïa llum amb un arc elèctric d'alt voltatge.

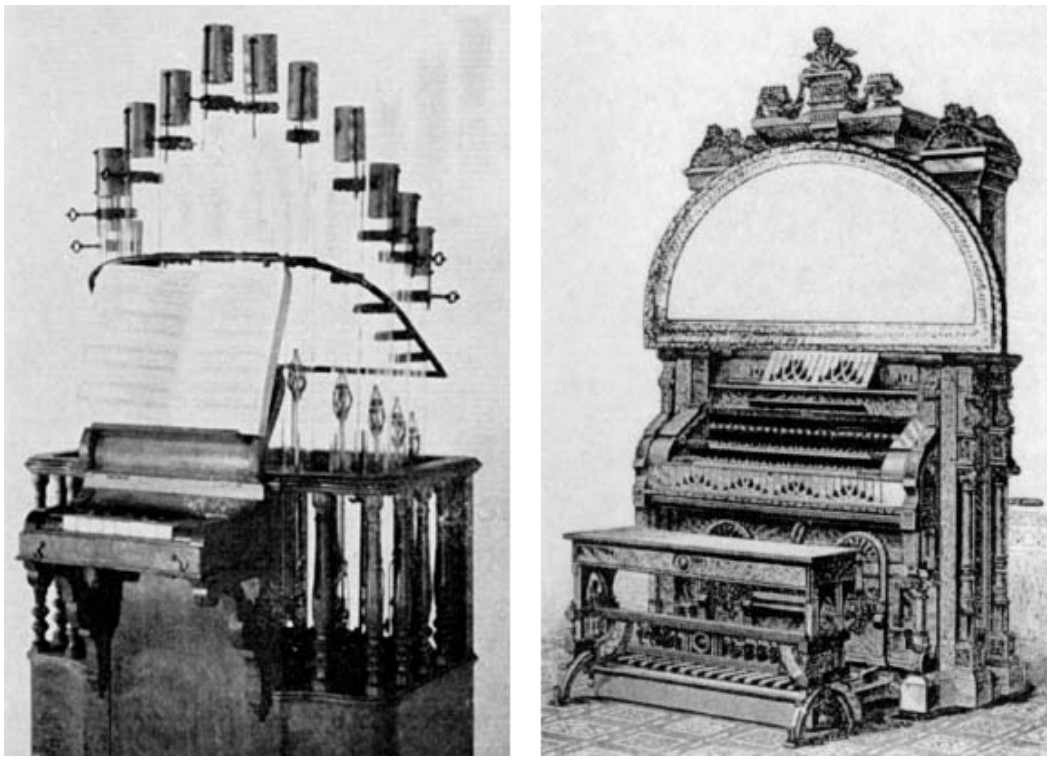


Figura 2. A l'esquerra el *Pyrophone* de Kastner, a la dreta l'orgue de colors de Bishop.

Però el boom del desenvolupament de sistemes visuals musicals abstractes no va arribar fins als inicis del segle vint. Fou quan la unió dels avenços en la tecnologia elèctrica i de l'òptica, juntament amb la invenció del cinema i l'apogeu de l'abstracció en l'art visual va propiciar el desenvolupament de molts nous sistemes en poques dècades. De tots aquests instruments caldria destacar especialment el *Clavilux* d'en Thomas Wilfred, el *Lumigraph* d'Oscar Fischinger i el *MobilColor Projector* de Charles Dockum.

Com molts altres, Thomas Wilfred (el creador del *Clavilux*), pensava que existia un *mapping* directe entre el so i el color, però després d'estudiar el treball dels seus predecessors i veient totes les errades i divergències existents, va arribar a la conclusió que no hi havia una correspondència absoluta entre el color i el so. Així, tenint com a objectiu l'art de la llum, on el so i la música eren admesos només com a accessoris, Wilfred va desenvolupar el 1919 el primer prototipus del seu orgue de colors, el *Clavilux*. El *Clavilux* consistia en un gran teclat amb cinc files de tecles lliscants i stops que amb l'ajut d'uns quants projectors (sis en el primer model) i un cert nombre de reflectors auxiliars obtenia els colors desitjats. Aquest instrument va tenir un gran èxit i Wilfred va fer concerts i gires per tot occident. A les següents figures podem observar els resultats visuals d'aquest sistema.

Figura 3. Algunes projeccions representatives del *Clavilux*.

Així com Thomas Wilfred va refusar la possibilitat de crear relacions entre el so i la imatge, donat que un mapeig individual era qüestionable, Oskar Fischinger pensava tot el contrari i va crear el seu *Lumigraph*, fent *mappings* arbitraris i personals entre imatge i so. Es tracta doncs, d'un instrument que permet a qualsevol crear imatges a partir de qualsevol música de manera molt simple. El *Lumigraph* està format bàsicament per un marc que conté una fina fulla de làtex que amb l'ajut de gels de colors, llums de neó... fa que si estem en una habitació fosca, puguem tocar l'instrument simplement interaccionant amb la fulla de làtex, ja sigui amb la mà o amb qualsevol objecte. Fischinger va obtenir la patent del seu instrument al 1950. Actualment el *Lumigraph* està exposat al Deutsches Filmmuseum de Frankfurt, on és tocat amb regularitat.

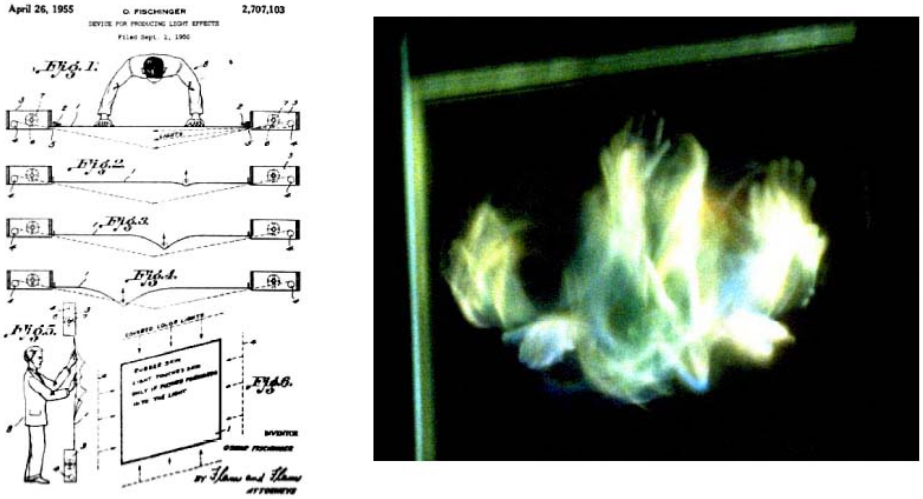


Figura 4. A l'esquerra un esquema de la patent del *Lumigraph*, a la dreta una imatge del *Lumigraph* en funcionament.

S'ha de mencionar el treball de Charles Dockum. Dockum que va desenvolupar un gran sistema de projecció anomenat *MobilColor*, que permetia al seu usuari crear patrons temporals de moviments de formes de colors. La dinàmica d'aquests patrons era fixada mitjançant la utilització d'un sistema mecànic de programació, utilitzant lleves de diverses formes. Encara que el *MobilColor* podia produir tant imatges amb formes afilades com suaus, es va emprar sobretot amb imatges ja preparades.

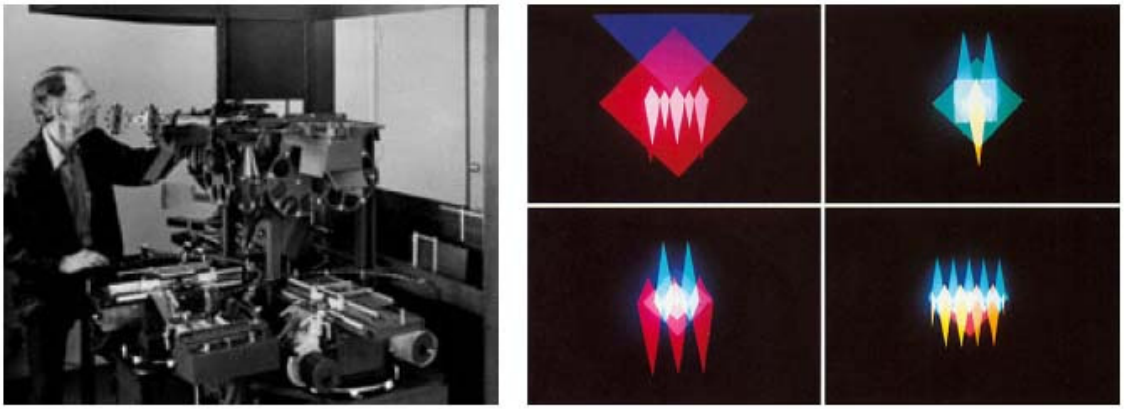


Figura 5. A l'esquerra Charles Dockum amb el seu *MobilColor Projector*, a la dreta alguns exemples de projeccions de l'instrument.

2.2 Sistemes visuals musicals en el domini computacional

La tecnologia computacional va fer possible que els dissenyadors de música visual poguessin superar les limitacions que els marcava la física, la mecànica o l'òptica. Per exemple, quan algú volia mostrar un triangle, necessitava un dispositiu en forma de triangle a traves del qual havia de projectar llum, per tal d'aconseguir, finalment, projectar la imatge del triangle sobre una pantalla. Aquesta i moltes més complicacions s'acaben amb l'arribada dels ordinadors.

Els pioners en el camp dels gràfics per ordinador, foren per una banda Ivan Sutherland al MIT que al 1963 va desenvolupar el primer programa de dibuix en el marc de la seva tesis doctoral, el *Sketchpad*, i per l'altre Myron Krueger que va centrar els seus esforços en la connexió entre la interacció i els gràfics per ordinador, i que podem veure en el seu *VideoPalace* on empra informació obtinguda mitjançant visió artificial per dirigir animacions de formes abstractes.

Pel que fa a la síntesis de so, aquesta es va veure afavorida primerament per l'aparició del transistor, i després, evidentment, per l'aparició de l'ordinador [9]. Els primers experiments sonors amb computadors els va dur a terme Max V. Matthews el 1957, i des d'aleshores han aparegut dotzenes de tècniques diferents de síntesi de so i s'han creat milers de programes d'ordinador per compondre, controlar o interpretar aquests so sintetitzat.

En el següent subapartat es descriuran les estratègies de representació d'àudio que s'han vingut utilitzant en els computadors en aquests últims anys i en el pròxim veurem alguns dels sistemes històricament més representatius en aquest camp.

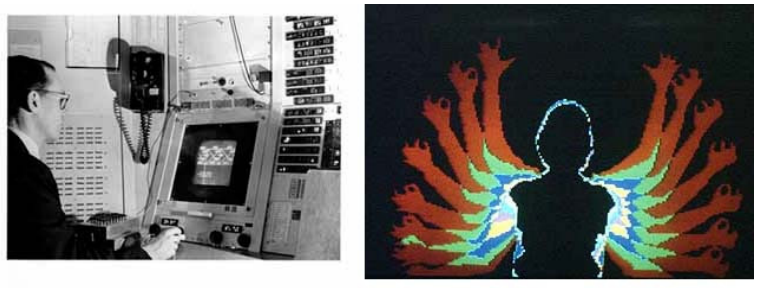


Figura 6. A l'esquerra el *Sketchpad*, a la dreta el *VideoPalace*.

2.2.1 Estratègies de representació del so en els ordinadors

La majoria de les interfícies visuals per els control i la representació del so han estat transposicions de les solucions gràfiques convencionals a l'espai de la pantalla de l'ordinador [10]. En concret, són tres les principals metàfores escollides per representar la relació imatge/so en el camp de la música visual en el context computacional: els displays basats en partitures, els basats en panells de control i els basats en objectes virtuals interactius.

Displays basats en partitures

Les partitures, que foren creades i utilitzades inicialment pels monjos medievals com a mètode per escriure el to de les melodies, s'han convertit en una notació emprada gairebé en tots els àmbits de la música. Les partitures són bàsicament diagrames bidimensionals, que relacionen la dimensió del temps amb qualsevol altra dimensió del so. És a dir, és una gràfica on un eix representa el temps i l'altre una dimensió del so, com pot ser el to o l'amplitud. Així, dins aquesta definició hi entren tant les partitures basades en pentagrames, com els displays de forma d'ona, els piano rolls, o els espectrogrames.

Malgrat que per la seva utilització és necessari haver aprés prèviament a interpretar adequadament aquests llenguatges de notació musical, els sistemes basats en representació de la música mitjançant l'ús de partitures són actualment els més utilitzats per un gran numero de músics. La majoria d'aquests sistemes de representació basats en partitures no ofereixen un control en temps real del so, encara que alguns d'aquests ja permeten editar les dades en temps de reproducció.

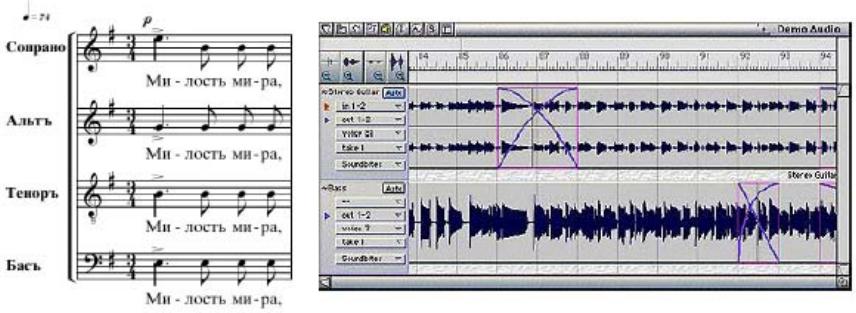


Figura 7. Exemples de displays basats en partitures.

Displays basats en panells de control

Aquests displays estan inspirats totalment en el que serien els controls de so dels mítics sintetitzadors analògics de la dècada dels setanta. Es tracta doncs, d'interfícies formades per un panell on hi trobem una llarga llista de botons, rodetes i barres desplaçables, a l'estil dels sintetitzadors d'aquella època.

Els sistemes que utilitzen aquest tipus d'interfície pateixen els mateixos problemes que tenien els seus "predecessors", és a dir, tenen la pantalla massa plena de coses, són bastant confosos i no solen tenir un *mapping* obvi entre els controls i els paràmetres sonors. Per acabar, cal dir que simplement observant algunes de les interfícies basades en aquesta estratègia, ja podem dir que en aquest paradigma la imatge i el so estan totalment desconnectats, és a dir no existeix una relació directe entre la imatge i el so.

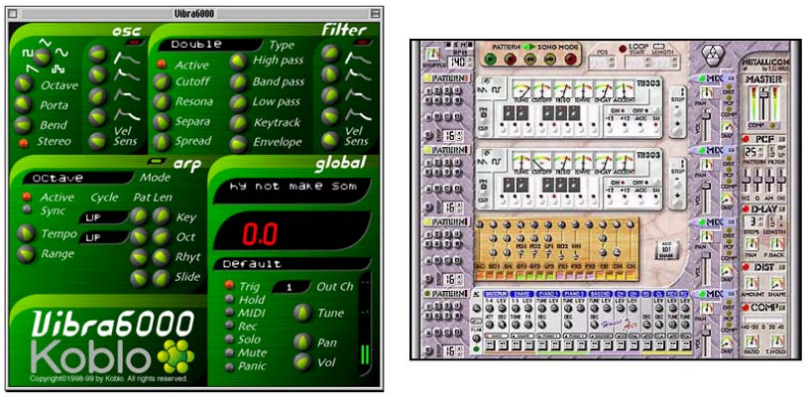


Figura 8. Exemples d'interfícies basades en panells de control: A l'esquerra el *Vibra6000*, a la dreta el *ReBirth RB-338*.

Displays basats en objectes virtuals interactius

La tercera estratègia per representar la música en l'ordinador està basada en un sistema en que un grup d'objectes virtuals poden ser manipulats per l'usuari amb la intenció de compondre música. Així, per exemple, tenim una finestra on hi apareixen una sèrie d'objectes que es poden moure amb l'ajuda del ratolí, se'ls pot canviar la forma, se'ls pot assignar un timbre, o se'ls pot fer xocar, ja sigui entre ells o amb els límits de la finestra. Aquest exemple comentat prové del programa *Sounder* desenvolupat per Jack Freudenheim, però hi ha molts altres programes que posseeixen una forma d'interactuar similar amb petites variacions sobre el mateix concepte, com poden ser el *Transducer* de Reed Kram, o el *Stretchable Music* de Pete Rice, ambdós del MIT.

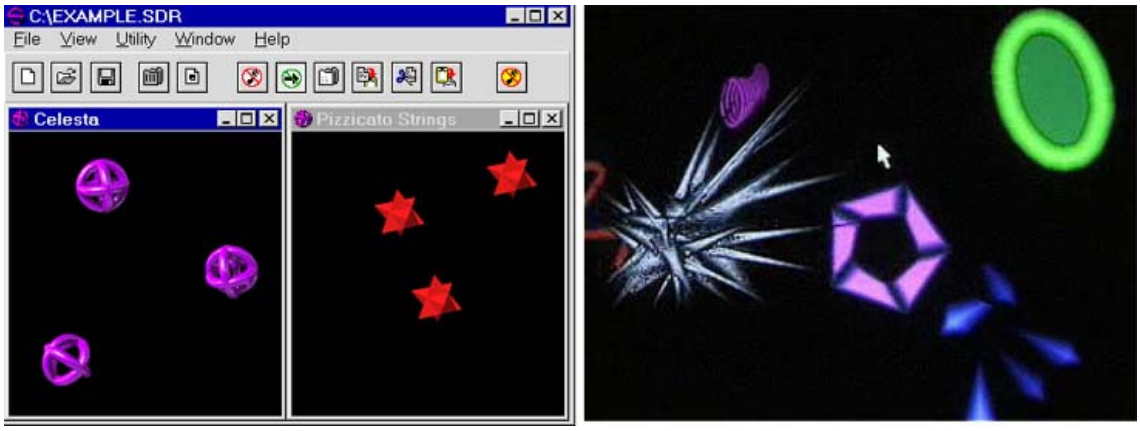


Figura 9. A l'esquerra el *Sounder*, a la dreta el *Stretchable Music*.

Aquests sistemes són perfectament aptes per a la composició a temps real, ja que estan pensats per ser-ho, i ho fan utilitzant una forma d'interacció bastant simple i intuïtiva. El problema que solen tenir la majoria d'aquests sistemes és que acaben produint experiències predictibles, ja que la majoria d'ells restringeixen a l'usuari a fer servir elements que estan massa predefinits.

2.2.2 Sistemes per a la interpretació visual en els ordinadors

Apart dels sistemes que han sortit intercalats en el subapartat anterior, hi han hagut molts altres sistemes de música visual per ordinador. Ara parlarem d'alguns dels més representatius.

El primer sistema a comentar és un sistema estrictament visual, és a dir, no té en principi res a veure amb la música, però ens interessa per la seva posterior contribució en la música visual. Es tracta del *DynaDraw*, que fou dissenyat al 1989 per Paul Haelberli [11]. El *DynaDraw* és simplement un programa de dibuix on els moviments del pintor són augmentats mitjançant una simulació física d'elasticitat. És a dir, el pinzell i la tinta són modelats com a objectes físics amb

massa, velocitat i fricció. Així doncs, la principal contribució d'aquest sistema a la música visual és la idea que els moviments d'interacció dels usuaris poden ser augmentats mitjançant una simulació física, dotant així d'un major dinamisme tant a les imatges creades a partir d'aquesta interacció com a la música mateixa.

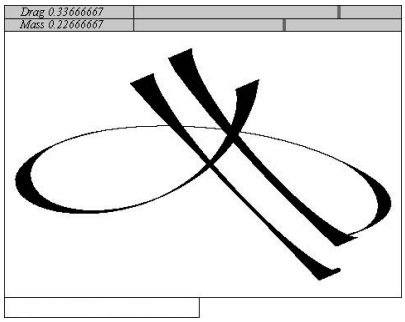


Figura 10. Captura de pantalla del *DynaDraw*.

Cal destacar també el treball de John Maeda al MIT. A principis dels noranta, Maeda va desenvolupar una sèrie de sistemes interactius per tal d'estudiar de quina manera els programes de dibuix es podien utilitzar per a interpretar i mostrar imatges dinàmiques. Va dissenyar tres sistemes, el *Timepaint*, el *A-Paint* i el *Process Color Dance* [12]. En el primer, Maeda ens presenta un programa de dibuix en forma d'experiència visual, segons diu ell, en dues dimensions i mitja on aquesta mitja es suposa que és el temps. El resultats del *Timepaint* són imatges creades per punts, que es mouen dirigits pel mouse, i les seves esteles. En canvi, els resultats dels altres dos, el *A-Paint* i el *Process Color Dance*, són més complexes i ofereixen un vocabulari visual més extens. El *A-Paint* basa el seu funcionament en la tinta intel·ligent, i permet al dissenyador definir com reaccionarà la tinta a les condicions del temps, de l'espai i del mateix pintor, per això necessita dos modes de funcionament, un per a la definició i un altre per a la interacció. En el *Process Color Dance*, Maeda va seguir investigant en aquesta línia però aquesta vegada utilitzant regions rectangulars de diferents colors per crear composicions de color reactives.

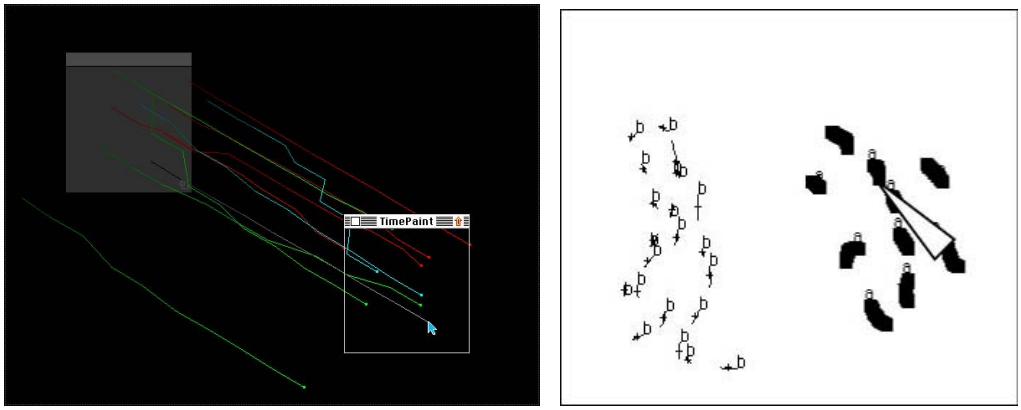


Figura 11. A l'esquerra una captura del *Timepaint*, a la dreta una del *A-Paint*.

Un altre enginyer que ha fet desenvolupaments importants en el camp de les aplicacions interactives per generar visualitzacions abstractes és Scott Snibbe. Entre el 1991 i el 1995, Snibbe va desenvolupar el *Motion Phone* [13], que és una aplicació per a la creació d'animacions dinàmiques a partir dels moviments introduïts pels usuaris. Posteriorment, Snibbe va realitzar tres treballs, que va ajuntar sota el nom de *Dynamic System Series*, en els quals es va centrar en com el fet d'augmentar els moviments d'interacció de l'usuari pot servir per la realització d'animacions abstractes. En el *Bubbleharp*, Snibbe creava diagrames de Voronoid a partir de l'entrada de l'usuari, en el *Lazy Line* s'augmentava l'entrada de l'usuari mitjançant l'ús de filtres i en el *Gravilux* s'augmentava mitjançant una simulació física.

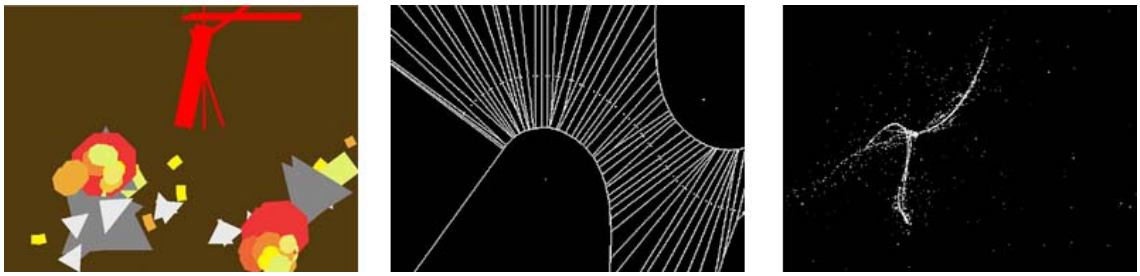


Figura 12. D'esquerra a dreta, el *Motion Phone*, el *Bubbleharp* i el *Gravilux*.

Acabarem aquest apartat parlant del *Music Insects* de Toshio Iwai [14]. Aquest sistema creat al 1991, és un híbrid entre un programa de dibuix convencional i un seqüenciador interactiu. L'usuari del sistema situa quadrats de varis colors sobre la pantalla els quals formaran la partitura que serà interpretada per uns quants insectes animats que s'arrosseguen per la superfície del marc on es pinten aquests quadrats. Així, cada vegada que un bitxo es troba amb un d'aquests quadrats es produeix una nota musical amb el to pertinent al color del quadrat i amb el timbre assignat a aquell bitxo en concret. Les coses es poden complicar encara més utilitzant uns quadrats especials que permeten fer ritmes amb *loops* i altres passatges més complexes. Aquest sistema ofereix una solució força equilibrada pel que fa a la creació simultània d'imatge i so, no obstant, la visualització oferta és molt estàtica en comparació amb el so.

Figura 13. El *Music Insects* de Toshio Iwai.

2.2.3 Els reproductors multimèdia i les utilitats VJ

Donat la seva popularitat, no podem acabar aquesta secció sense parlar dels reproductors multimèdia, i és que en els darrers anys, s'han fet amb un lloc dins cada un dels ordinadors personals, de fet, ara és més estrany no tenir un reproductor multimèdia que tenir-lo. Aquest fet és degut, probablement, a l'aparició dels nous formats de compressió d'àudio, com poden ser el mp3 o el *ogg vorbis*. El cas és que, avui en dia, aquests reproductors, com poden ser el *WinAmp* o el *Windows Media Player*, duen incorporats un conjunt bastant gran de *plug-ins* que permeten a l'usuari veure el resultat de la música de varies maneres distintes; creant d'aquesta manera un paral·lelisme entre el so i la imatge que generen. Aquests programes es solen basar en el següent esquema per generar les seves visualitzacions [1].

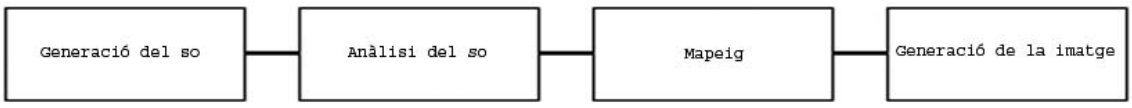


Figura 14. Esquema de funcionament d'un reproductor multimèdia estàndard amb *plugins* de visualització.

Però aquests sistemes no permeten cap interacció sobre la generació de la imatge, l'únic que permeten és anar canviant de visualització, és a dir, canviar l'ultima fase de l'esquema anterior. Quan un d'aquests visualitzadors es torna interactiu, passa a dir-se utilitat *VJ (Video Jockey)*, com poden ser *FreeJ*, *Arkaos* o *Resolume*. Aquests programes solen oferir interactivitat en cada una de les capes de l'esquema anterior.

Igual que, com hem vist fins ara, hi ha programes que generen imatges a partir de la música, també hi ha programes que generen música a partir de les imatges. En aquest cas, l'esquema anterior s'invertiria i ens quedaria com el següent.

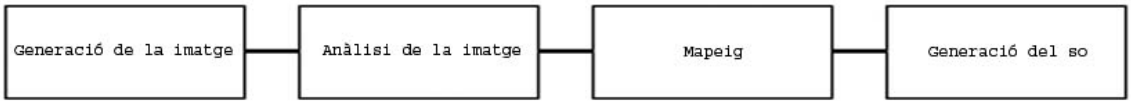


Figura 15. Esquema de funcionament d'un generador de música a partir d'imatges.

2.3 El treball de Golan Levin

En parlar dels sistemes visuals musicals recents, hem de parlar del treball realitzat per Golan Levin. Levin, en vista de tot el que s'ha explicat anteriorment i com a tesi del Master Of Science in Media Arts and Sciences al MIT [10], ha realitzat un estudi sobre la interpretació audiovisual en la que planteja una nova metàfora d'interfície per als sistemes visuals musicals.

A partir de les virtuts i de les debilitats de sistemes com els que hem vist al llarg d'aquest capítol, Levin proposa uns objectius bàsics que, segons ell, ens conduirien a un instrument audiovisual amb una expressivitat adient tant pel que fa al so com a la imatge. A continuació es comenten cada un d'aquests objectius.

Dinamisme i simultaneïtat d'imatge i so

El sistema ha de fer possible la creació i interpretació d'imatges dinàmiques i so, també dinàmic, amb simultaneïtat i en temps real. És a dir, no s'ha de produir només imatge, com passava en sistemes com el *Lumigraph* de Fischinger, sinó que el sistema ha de crear els dos feedback a partir de l'entrada rebuda.

Resultats interessants i variats

Amb l'instrument s'han de poder interpretar el major repertori de composicions possible. Si l'usuari esgota aviat les possibilitats de l'instrument, vol dir que l'expressivitat de l'instrument és limitada i que l'usuari l'acabarà avorrint. Es tracta doncs, de fer un sistema amb la màxima expressivitat possible.

Dimensions visual i sonora igual de mal·leables

El sistema ha de tractar de forma equitativa la imatge i el so. El sistema no es pot centrar en la imatge i a partir d'aquesta generar el so ni a l'inrevés. Segons Levin, és molt més interessant crear un sistema on les interpretacions siguin igual d'expressives tant en el domini sonor com en el domini visual.

Evitar llenguatges visual convencionals

S'ha d'evitar l'ús de convencions arbitraries o llenguatges visuals convencionals com poden ser les partitures o els diagrames. Aquests llenguatges requereixen un procés previ d'aprenentatge,

que hauria de ser evitat per tal d'aconseguir sistemes que puguin ser emprats immediatament. Sense aprenentatge previ.

Aprenentatge fàcil

Les accions bàsiques del sistema han de ser fàcils de deduir i s'haurien d'aprendre instantàniament amb el primer contacte amb l'instrument, mentre que accions més sofisticades han de ser també possibles encara que costi un poc més d'aconseguir. És a dir, la interfície del sistema ha de ser a la vegada fàcil d'aprendre a manejar i potent pel que fa a les possibilitats d'expressivitat.

Per tal d'assolir els anteriors objectius, Levin introdueix un nou paradigma d'interfície per a instruments audiovisuals. Segons diu, aquesta interfície es basa en una substància audiovisual extremadament variable i dinàmica que pot ser lliurement pintada, manipulada i esborrada en un context lliure de diagrames. Es tracta que l'usuari entri les dades mitjançant moviments naturals, com podria ser pintar en un camp de dues dimensions emprant qualsevol dispositiu electrònic. Aquestes dades son tractades per algoritmes d'anàlisi de senyal digital, algoritmes de filtratge o simulacions físiques i després són enviades a un sintetitzador de gràfics i a un sintetitzador de so. Això permetria la generació a temps real d'imatge i so de manera simultània. Per complir totalment amb els objectius, els *mappings* utilitzats haurien de fer referència a la percepció i haurien d'evitar utilitzar algun tipus de llenguatge visual o textual.

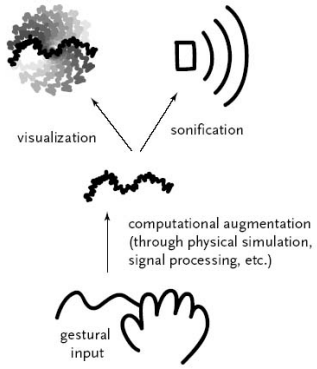


Figura 16. Esquema del paradigma proposat per Levin.

Com a recolzament a la seva tesi, Levin va desenvolupar dues sèries de sistemes que tenien com a finalitat implementar el seu paradigma d'interfície, així com, evidentment, assolir els objectius redactats anteriorment. En la primera sèrie només es volia crear sistemes de visualització, és a dir, només s'ocupaven de generar imatge. En la segona s'intentaven aconseguir tots els objectius anteriors i es van dissenyar sistemes per a la interpretació simultània d'imatge i so.

La construcció d'aquests primers sistemes sense àudio, entre 1997 i 1998, es pot interpretar com una manera d'entrenar i d'agafar idees i fer proves per a la posterior creació dels sistemes audiovisuals posteriors. En total, Levin va dissenyar quatre sistemes només visuals, són el *Streamer*, el *Escargogolator*, el *Polygona Nervosa* i el *Directrix*, alguns d'ells amb la col·laboració de Snibbe.

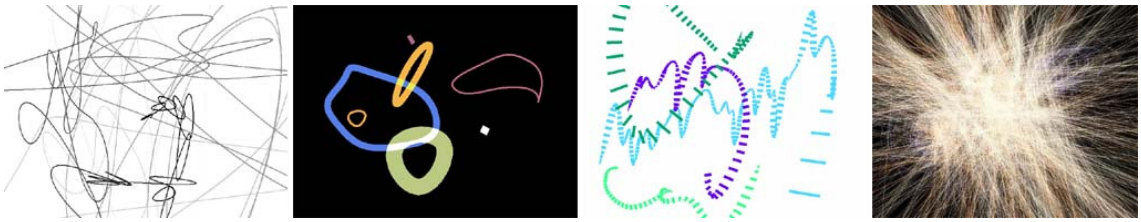


Figura 17. D'esquerra a dreta, *Streamer*, *Polygona Nervosa*, *Escargogolator* i *Directrix*.

La segona sèrie de sistemes està formada per un conjunt de cinc aplicacions desenvolupades entre 1998 i 2000 per Levin baix la direcció de John Maeda. Aquests cinc experiments intenten complir tots els objectius de Levin i generen imatge i so a temps real fent servir el seu nou paradigma d'interfície.

El primer d'aquests sistemes fou el *Yellowtail*, un sistema de creació i interpretació d'animació abstracte que a partir dels gestos del usuari genera patrons per a l'animació d'unes formes semblants a cucs. Això pel que fa la generació de la imatge. Per la generació de so trobem un espectrograma quadrat al mig de la pantalla que es podria dir que s'encarrega de generar el so a partir del brillo dels píxels que té a dins. És un sistema molt senzill d'utilitzar i permet un gran control sobre l'espectre del so, però els *mappings* existents entre imatge i so són molt simples i molt poc flexibles.

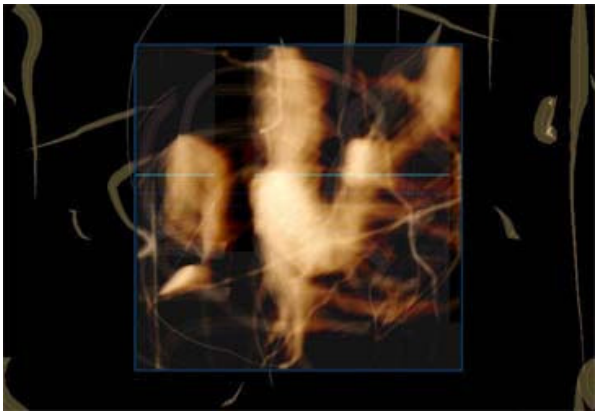


Figura 18. Un screenshot del *Yellowtail*.

Immediatament després del *Yellowtail*, i com a reacció directe a les seves carències, Levin va desenvolupar el *Loom*. Per a la construcció d'aquest sistema, Levin es va basar en el *Yellowtail*, però en va eliminar l'espectrograma central i va intentar que cada element visual tingués una correspondència amb un element sonor i a l'inversa. Aquesta correspondència donaria més feedback a l'usuari. Aquest sistema millora certs aspectes del *Yellowtail*, però segueix tenint uns *mappings* poc flexibles i alguns paràmetres no controlables.

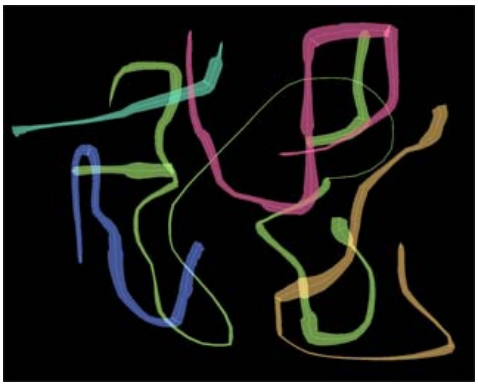


Figura 19. El *Loom* en funcionament.

El següent experiment de Levin fou el *Warbo*. En aquest sistema l'usuari crea un grup de rodones de diferents colors, on cada una representa un to en correspondència amb el seu color, i després, mitjançant una interfície basada en mouse i bolígraf òptic, els usuaris decideixen com i quan es fan audibles aquestes rodones de colors. Una altra vegada el sistema té uns *mappings* poc flexibles, però introdueix noves millores com poden ser la generació d'acords, l'encert de relacionar el color de l'objecte amb la seva freqüència sonora, o la introducció de la interfície formada pel mouse i el bolígraf que ens permet interactuar amb el sistema amb les dues mans.

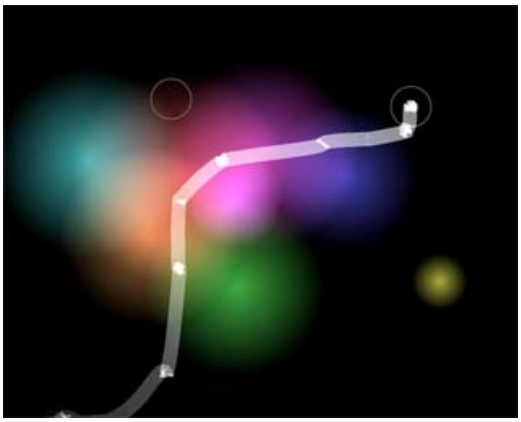


Figura 20. Una captura de pantalla del *Warbo* de Levin.

Un altre dels sistemes creats per Levin és l'*Aurora*. En aquest experiment, Levin introdueix la síntesi granular per generar el so del sistema, mentre que en la part visual es decideix pel que podríem anomenar núvols de colors. La relació imatge/so està molt ben tractada en aquest instrument i molts paràmetres visuals estan relacionats amb la síntesi granular del so. Com els altres que hem vist, es tracta d'un sistema fàcil d'emprar i té realment capacitat per a la improvisació en temps real a més de presentar una visualització gràfica innovadora.

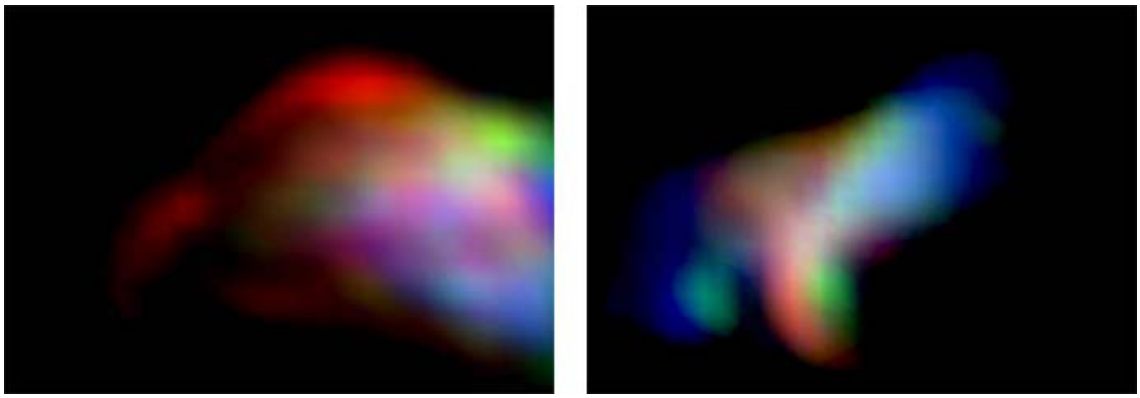


Figura 21. Dos screenshots de l'*Aurora*.

L'últim i també interessant sistema presentat per Levin en la seva tesi és el *Floo*. Es tracta d'un sistema que, com els anteriors, es serveix dels gestos dels usuaris per a generar àudio i vídeo. En aquest cas, igual que en l'*Aurora*, Levin opta per la síntesi granular per a la generació del so, però en canvi el feedback visual es basa en la simulació del fluix i la dispersió dels fluids. Encara que és un sistema força interessant té els mateixos punts dèbils que els altres sistemes de Levin que hem comentat, però a més és difícil d'aprendre a manejar i té algunes carències d'organització.

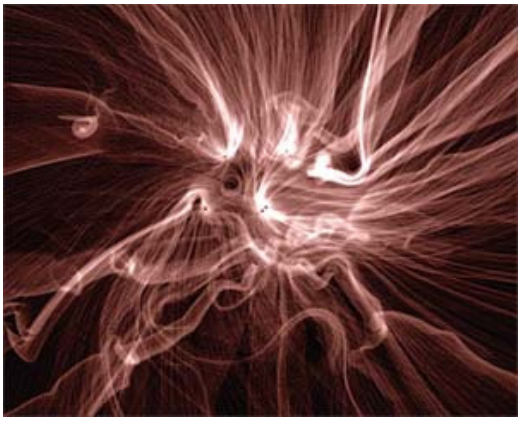


Figura 22. Una imatge del Floo en acció.

Sense cap tipus de dubte el treball de Golan Levin és força interessant. La declaració dels seus objectius i la definició del seu paradigma d'interfície són molt encertats i aconsegueix superar moltes de les limitacions que podíem veure en els sistemes anteriors al seu treball. Tot hi que, com hem vist, el seu treball també té algunes mancances.

2.4 El FMOL

No podem acabar el repàs històric sense comentar el que és sense cap tipus de dubte el precedent de la reacTable*. A l'any 1997 el grup català de teatre La Fura dels Baus li va proposar a Sergi Jordà, que després seria d'ideòleg de la reacTable* i el director d'aquest projecte, el desenvolupament d'un sistema de creació musical basat en Internet. El resultat va ser el FMOL [1] [2], un instrument musical virtual concebut com una eina per a la composició musical col·laborativa a Internet.

FMOL ens interessa especialment per la seva interfície única, encara que es serveix del mouse i del teclat, pel fet que es pot dir que té un circuit de feedback tancat pel que fa al so i a la imatge. Els gràfics de l'aplicació serveixen a la vegada com a interfície d'entrada per al control del so, i per mostrar l'activitat sonora del sistema. Així doncs, també es tracta d'una visualització d'àudio interactiva, ja que s'està modificant la visualització a la vegada que es toca sobre ella.

L'instrument tenia dos objectius clars: introduir nous intèrprets a la música electrònica experimental per una banda i que fos una eina simple i complexa a la vegada, és a dir, una eina que evités la frustració de l'usuari, però que al mateix temps fos capaç de crear música rica i diversa permetent diferents corbes d'aprenentatge.

Segons diu Jordà, FMOL és un instrument que no deixa de sorprendre'l. Després de set anys tocant-lo, encara aprèn coses noves sobre el seu instrument. Els objectius de l'instrument es consideren assolits. Cal dir que el FMOL s'ha fet servir en varis concerts, i que el FMOL Trio, format per Sergi Jordà i Cristina Casanova al FMOL i Pelayo F. Arrizabalaga als instruments convencionals, ha gravat varis CDs de música electrònica improvisada.

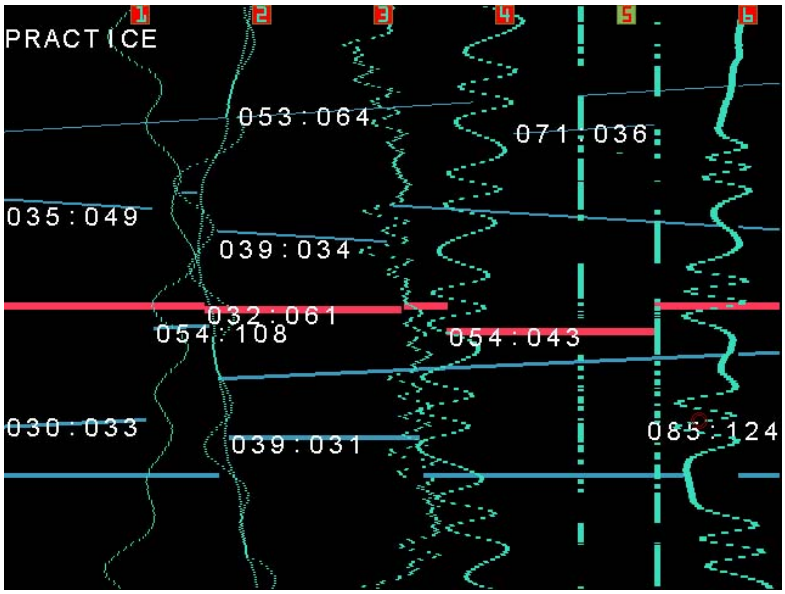


Figura 23. Una captura de pantalla del FMOL en funcionament.

El més interessant del FMOL des del punt de vista del meu projecte és, sens dubte, el feedback visual. Està comprovat que el feedback visual no és vital en els instruments convencionals, però aquests ofereixen altres tipus de feedback, apart d'evidentment el sonor, com pot ser el tacte de les cordes d'una guitarra. En el cas del FMOL, si no hi hagués feedback visual, només hi hauria feedback sonor i això limitaria molt la forma d'interactuar amb l'instrument. Per tant el feedback visual al FMOL és fonamental. Com podem veure en la figura que hi ha a sobre

d'aquestes línies, la interfície del FMOL està formada per una graella 6x6 dinàmica, que representa en tot moment l'activitat del sistema. Les línies verticals representen els generadors de so i les horitzontals els processadors d'efectes. L'usuari actua sobre aquestes línies fent clic sobre elles amb el mouse i arrossegant-les. El resultat és el so produït i la representació de l'ona generada en la línia del generador.

Si comparem el treball de Golan Levin amb el de Jordà, veurem que mentre l'objectiu primordial de Levin és aconseguir un instrument amb el que poder pintar mentre el fas sonar, en el FMOL de Jordà el que es pretén al final és aconseguir una interfície més intuïtiva per a la creació de música, i si per això s'ha de fer servir el feedback visual, doncs es fa servir.

En resum, ens trobem davant un instrument musical virtual, que veu augmentades les seves possibilitats expressives mitjançant un feedback visual, que representa simbòlicament l'estat de l'instrument d'una manera dinàmica. Aquest instrument és el FMOL, i a partir d'aquest sortirà la idea de la reacTable*, sobretot degut a que interactuant amb el FMOL, un es dona compte de les limitacions marcades per la interacció basada en teclat i mouse.

3. La reacTable*

3.1 Què és?

La reacTable* està pensada per ser un instrument musical electroacústic que segueixi la tradició del sintetitzador FMOL [1] [2] dissenyat per Sergi Jordà. La idea principal és la creació d'un instrument electrònic musical amb interfície tangible que permeti interpretacions expressives i col·laboratives en temps real a músics professionals, sense les limitacions marcades per les interfícies basades en pantalles de la música electrònica. Moltes d'aquestes interfícies tenen les possibilitats de control molt limitades i proveeixen un feedback molt petit tant per l'audiència com per l'intèrpret. És tracta llavors de crear un instrument que respongui a l'equació: **reacTable* = FMOL + MAX + Tangible**.

Com diu el seu nom, la reacTable* és un instrument musical basat en una taula rodona. Aquesta taula no té cap tipus de sensors, de cables o botons. Una webcam analitza permanentment la superfície de la taula, mentre un projector hi dibuixa una interfície interactiva i dinàmica a sobre. Manipulant un conjunt d'objectes que hi ha disponibles a sobre la taula, l'artista va construint i tocant l'instrument al mateix temps. Aquests objectes són majoritàriament passius i estant fets de plàstic o fusta amb diferents formes [3]. Els usuaris interactuen amb ells movent-los per sobre la taula, canviant la seva orientació o canviant la cara que contacta amb la taula. Cada un d'aquests objectes té una funció assignada, aquesta pot ser la generació, la modificació o el control d'un fluix de so i reacciona amb objectes propers que siguin compatibles amb ell. Així com la taula està equipada amb sensors que permetin la identificació i el seguiment de la posició i l'estat dels objectes, l'usuari no necessita ni ha de portar cap tipus de sensors o altres dispositius. Pel que fa al feedback d'aquest instrument tenim, per una banda, òbviament, el so que es genera mentre es toca, però a més tenim un feedback visual per mitjà d'una projecció d'una representació gràfica de les fluixos sonors i de control, així com de l'estat dels objectes a sobre la superfície de la taula.

Figura 24. Primer prototipus de la reacTable*

3.2 Objectius de la reacTable*

La meta del projecte reacTable* és la creació d'un instrument musical interactiu de l'estat de l'art en aquest àmbit. Aquest instrument ha de complir els següents objectius [1] [2]:

- Col·laboratiu: L'instrument ha de suportar un ús simultani de les seves funcions per varis intèrprets. Aquests usuaris podran estar interactuant amb la mateixa taula o en diverses taules connectades per mitjà d'una xarxa.
- Intuïtiu: Ha de ser fàcil d'aprendre a tocar, no s'ha de necessitar cap tipus de manual o d'instruccions i ha d'evitar les possibles frustracions de l'usuari, fins al punt que no ha de ser totalment entenedor, però ha de ser coherent i responsable.
- Sonorament ric, competitiu i interessant: Ha de permetre la creació, experimentació i exploració de nous sons i ha de permetre al seu intèrpret anar millorant i anar adquirint noves habilitats.
- Adaptable: La reacTable* ha de ser apropiat tant pels completament novells, per quan es posi en certes instal·lacions de cara al públic general, com per a músics avançats i professionals de la música electrònica.
- Interacció natural: Per interactuar amb l'instrument l'usuari no ha d'utilitzar ratolí, ni botons, ni teclat, ni cables, ni ha de portar cap dispositiu. Només s'interacciona manipulant el conjunt d'objectes existents sobre la taula.
- Flexible: Haurà de permetre un número flexible d'usuaris i aquests podran entrar o sortir del sistema sense avisar prèviament.
- Transparent: Tota la tecnologia involucrada en el sistema haurà de ser totalment transparent per l'usuari.

Així, com podem veure, es tracta d'un projecte que intenta, dins del món de la música electrònica, canviar d'alguna manera els bits pintats de les *GUI*s tradicionals per bits tangibles, que aprofitin la riquesa dels sentits de l'ésser humà, així com les habilitats que aquest ha desenvolupat al llarg dels anys d'interacció amb el món físic cosa que ja han intentat altres projectes com *SmallFish* o *Jam-o-Drum*.

3.3 Components de l'instrument

En la primera fase del projecte es van desenvolupar els conceptes basics de la reacTable* només en un prototipus software, que simulava la interacció tangible mitjançant una interfície gràfica convencional. Aquesta aproximació va permetre el prototipatge ràpid i la introducció de nous elements del sintetitzador i de nous elements d'interacció sense preocupacions sobre com es feia la seva implementació hardware. A la segona fase es va introduir un motor de visió per ordinador, que va permetre la construcció del primer prototipus tangible de la reacTable.

El sistema actual està implementat completament d'una manera modular, que permet la fàcil reutilització o substitució de qualsevol dels seus cinc components bàsics. Un mòdul sensor segueix la posició, l'orientació i l'estat de qualsevol objecte present sobre la taula. Els paràmetres extrets per aquest sensor es passen al component central de gestió, que interpreta els moviments dels usuaris en funció de les dades rebudes, generant xarxes de *patches* dinàmics [4] que guien als dos components de síntesi actuals, el component de feedback sonor i el component de feedback visual. Tots aquests components són completament independents i es comuniquen per mitjà d'un protocol bastant simple. Aquesta separació permet l'execució en diferents plataformes hardware evitant així possibles colls d'ampolla donat que cadascun d'aquests components requereix uns recursos computacionals concrets.

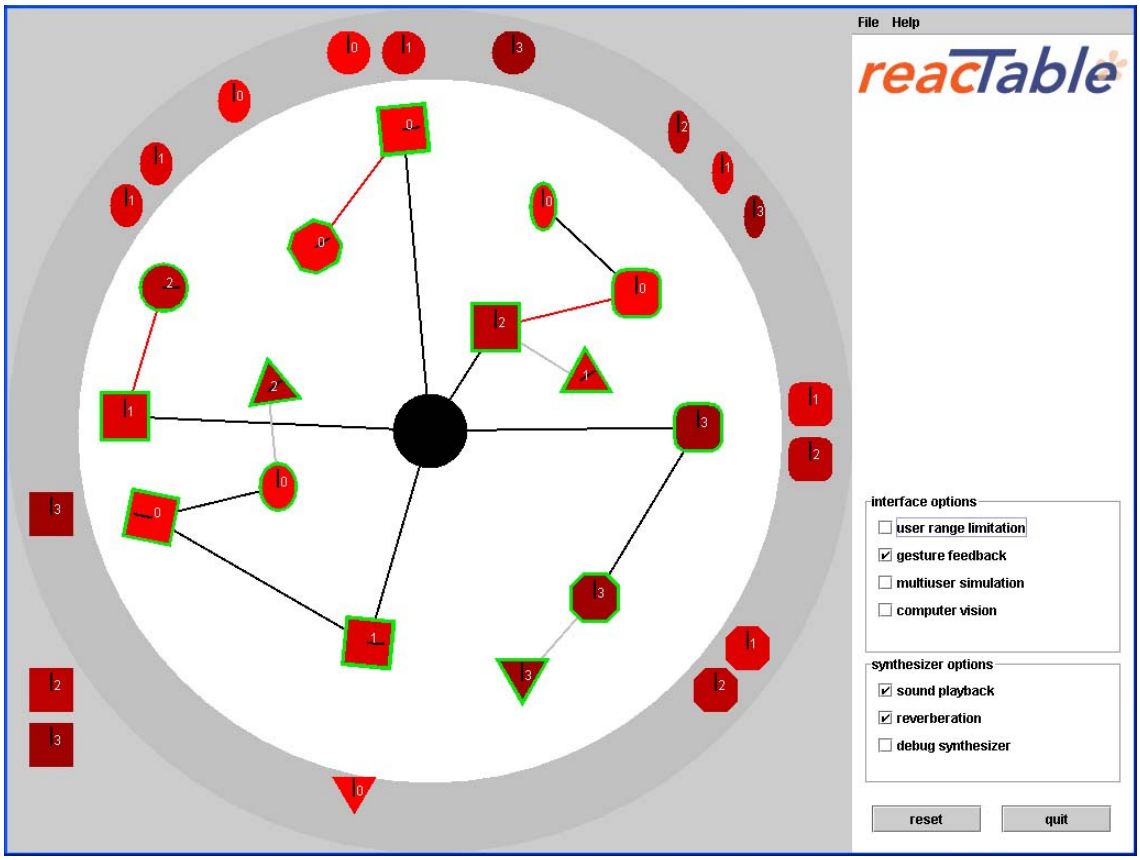


Figura 25. Captura de pantalla del prototipus software de la reacTable*.

3.3.1 Visió per ordinador

Com és pot veure en la figura 24, el primer prototipus de l'instrument conté una webcam centrada baix la taula que permanentment detecta el tipus dels objectes que estan sobre aquesta, així com la seva orientació i posició. Per aconseguir-ho la webcam està connectada a un ordinador que executa el motor de visió del *D-touch* [15] i es serveix de petits símbols visuals, que poden ser fàcilment impresos i aferrats als objectes, per tal de reconèixer-los. En la figura 26 podem apreciar l'aspecte d'aquests símbols.

Aquest sistema permet el seguiment dels objectes sobre la taula a un *frame-rate* raonable. El reconeixement d'aquest motor de visió és relativament robust donat el desenvolupament concurrent dels símbols i de l'algorisme de reconeixement. Per reconèixer els objectes el que es fa és bàsicament mesurar la quantitat de negre existent en el símbol. El conjunt de símbols actual permet un total de 128 dibuixos diferents, així que actualment es poden detectar 128 objectes diferents sobre la taula. El principal inconvenient d'aquest mètode és la visibilitat d'aquests símbols, que de moment s'ha solucionat aferrant-los a la cara inferior dels objectes. Cal mencionar també que s'està fent servir visió posterior, és a dir des de baix, per tal d'evitar possibles problemes d'oclusió per part de les mans o els cossos dels usuaris.

Figura 26. Un dels símbols emprats per reconèixer els objectes.

3.3.2 Síntesi de so

En el present prototipus la síntesi de so està implementada utilitzant el llenguatge de programació gràfica de so *Pure Data* [16]. Donat que el sistema de missatges intern de PD pot ser totalment controlat des de fora; la majoria de les operacions que són accessibles des de la interfície d'usuari poden ser adreçades a aquests missatges. Per això es va tenir que ampliar PD per permetre la desconnexió d'objectes per mitjà d'aquests missatges interns. Cal dir que aquests canvis han trobat també el seu lloc en la versió oficial de PD.

El motor de so implementa un gran conjunt de components bàsics tant de síntesi com de control, i que poden ser instanciats en el número que es desitgi a l'iniciar el motor. En temps d'execució, el motor rep missatges de control molt simples des de la unitat de gestió, que li indican com han d'estar les connexions tant de so com de control en aquesta xarxa d'objectes de processament.

Els objectes de processament són objectes d'alt nivell implementats com a abstraccions que estan construïts emprant unitats de més baix nivell disponibles en PD. Així doncs, tenim varis objectes, que comentarem més endavant, tots ells creats amb PD, com poden ser oscil·ladors analògics, oscil·ladors de taula d'ona o *granular synthesis*. És a dir, cada objecte tangible de la reacTable* té la seva corresponent abstracció implementada en PD.

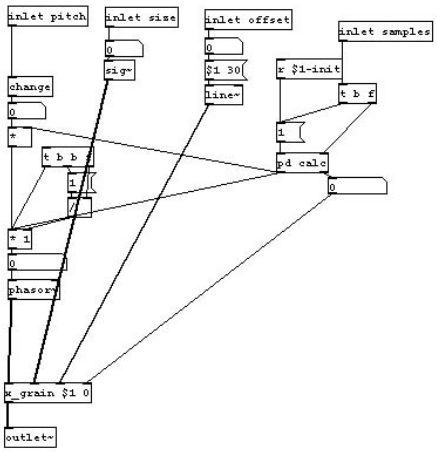


Figura 27. Implementació d'un *granular synthesis* en PD

Els objectes de la reacTable* es poden classificar en set diferents grups depenent de la seva funcionalitat. Així, tenim els següents grups d'objectes:

- Generadors: Són les fonts del so i poden produir varis tipus de sons, sintètics o gravats prèviament. Tenen una sortida d'àudio i varies entrades de control.
- Filtres d'àudio: Aquests són capaços de modificar el so que els hi entra mitjançant un algorisme intern, que va des d'un filtre passa banda fins a qualsevol possible efecte. Tenen una o dues entrades i una sortida d'àudio i varies entrades de control.
- Controladors: Són els objectes que poden modificar les propietats dels altres objectes mitjançant dades de control que es generen de manera algorítmica, ja sigui a partir d'oscil·ladors de baixa freqüència com a partir de fractals. De moment els controladors no tenen entrades però està previst que aviat en tinguin.
- Mescladors: Aquests tipus d'objecte poden prendre varies entrades d'àudio i convertir-les en una sola sortida, també d'àudio. Els mescladors invertits (*Splitters*), fan el contrari, passen d'una sola entrada a múltiples sortides.
- Rellotges sincronitzadors: Són objectes que poden influenciar varis objectes propers al mateix temps i de diferents maneres, per exemple enviant *triggers* sincronitzats, o corregint les freqüències baixes per tal de seguir un ritme donat. El seu paràmetre fonamental és el tempo.

- Contenidors d'alt nivell: Es tracta d'uns objectes que poden contenir virtualment *patches* construïts prèviament a partir dels objectes anteriors, amb la finalitat de poder construir estructures més complexes i per tant sons més complexes.

3.3.3 Síntesi visual

De la síntesi visual és del que s'encarrega aquest projecte i pel que hem pogut comprovar el feedback visual és un component indispensable de la reacTable*, ja que en els primers tests es va veure que aquest era crucial per a poder tocar l'instrument.

La seva funció és representar l'estat de l'instrument, mitjançant una projecció sobre la taula. Per fer-ho, aquest component rep dades tant des de la unitat de gestió central com des de el sintetitzador de so. A partir d'aquestes dades el sintetitzador d'imatges ha de ser capaç de crear aures a sota dels objectes actius i ha de representar els fluxos tant d'àudio com de control entre els diferents objectes, tot això en consonància amb el so que estem sentint.

Cal dir que el motor de síntesi d'imatges tindrà varis *skins* o temes visuals disponibles, per tal que es pugui triar el tema adequat per a cada sessió. A més, la creació de nous temes és relativament fàcil.

Aquest component, com veurem més endavant en aquest document, està implementat en C++ i amb l'ajut de llibreries gràfiques, que ja comentarem. Pel que fa la projecció de les imatges, aquesta es farà amb un projector situat a baix de la taula, igual que la webcam, per tal d'evitar oclusions per part de les mans o qualsevol altre part dels usuaris.

3.3.4 La interfície tangible

Podríem dir que la interfície tangible és la característica principal de la reacTable*, així com la seva principal raó de ser. És una interfície en la qual cada bloc virtual de síntesi és representat per el seu corresponent objecte físic, i mitjançant la manipulació d'aquests objectes els usuaris interaccionen amb l'instrument.

Així, tindrem disponible un conjunt d'objectes que poden ser manipulats pels intèrprets de varies maneres. Quan es posa un objecte sobre la taula, aquest s'activa, si l'anem movent per sobre la taula o el fem rotar anem canviant les seves propietats. Si dos objectes compatibles estan relativament pròxims es connecten. Així de la interacció s'extreuen dades com la posició de l'objecte, la seva orientació i la distancia i els angles que forma cada objecte respecte als altres amb qui està relacionat. D'aquesta manera podem arribar a generar xarxes d'objectes, és a dir *patches* dinàmics, creant nous sons i com no, música.

Els objectes de la reacTable* són senzills i passius [3], és a dir no venen amb cap tipus de cable, ni botó ni cap tipus de component electrònic a dins. L'usuari tampoc ha de portar cap tipus d'aparell o sensor per interactuar amb l'instrument. En un futur pròxim es pretén seguir també els moviments de les mans dels usuaris i que aquestes puguin influir sobre els fluixos així com pintar les formes de les ones dels oscil·ladors de taula d'ona.

El material, les formes o el tacte dels objectes de la taula no és casual, és a dir cada un dels objectes físics està construït pensant en l'objecte abstracte que representaria. Així la forma defineix els tipus genèric de l'objecte, per exemple els quadrats representen generadors, els cercles representen processadors, les estrelles, controladors i els triangles mescladors. La textura de la superfície de l'objecte s'empra per codificar els subconjunt al que pertany l'objecte, així un generador sinusoïdal normal pot ser representat per una superfície plana, mentre que un generador de soroll té una textura irregular o un *granular synthesis* pot ser representat per un objecte amb una superfície semblant a un paper de vidre. Per la seva part el material de l'objecte també fa la seva distinció, així si un objecte produeix sons sintètics, el seu objecte físic és de plàstic, en canvi un objecte que representi un *sampler* hauria de ser per exemple de fusta. De totes maneres aquest *mapping* encara ha de ser avaluat en posteriors tests.

Figura 28. Alguns objectes del primer prototipus de la reacTable*.

3.4 Projectes relacionats

Com la reacTable, són varis el sistemes basats en interfícies tangibles que han estat dissenyats amb la intenció d'aprofitar la riquesa dels sentits humans i de les habilitats que hem desenvolupat al llarg de la nostre vida en el món físic. De tots aquests sistemes n'hi ha un reduït grup, que cada dia s'està fent més gran, que dedica els seus esforços a la música, entre aquests hi podem trobar el *Smallfish* [17], el *Jam-O-Drum* [18], l'*Augmented Groove* [19], l'*Audiopad* [20] o evidentment la reacTable.

Figura 29. D'esquerra a dreta, el *SmallFish*, l'*Augmented Groove* i l'*Audiopad*.

Però en la reacTable* hi ha tres conceptes addicionals tant importants com la tangibilitat: la síntesi modular, la programació visual i el feedback visual. El concepte de síntesi modular prové dels primers sintetitzadors, tant en el domini analògic com en el domini digital. La síntesi modular ha demostrat els seu potencial sonor il·limitat i ha estat l'element essencial de tots els entorns de programació visual per so i música, com *MAX*, PD, *JMax* o *Reaktor*. D'altra banda, com estan demostrant tots aquests entorns, la programació visual constitueix avui dia el paradigma més flexible i estès per a la creació de música interactiva. Pel que fa al feedback visual, que és entès com una eina per maximitzar la comunicació entre l'usuari i l'instrument, és un concepte més recent i que està començant a mostrar el seu potencial. La reacTable* és probablement el primer sistema que intenta incorporar tots aquests paradigmes amb la intenció de construir un nou instrument musical, flexible, potent i intuïtiu.

4. Anàlisi dels requeriments del feedback visual

4.1 Usuaris del sistema

Els usuaris del mòdul de visualització, evidentment seran els mateixos usuaris de la reacTable*, tot i que el que ells faran serà interactuar amb un tot que serà la reacTable*. Així, per tant els usuaris del sistema aniran des de músics professionals especialitzats en música electrònica fins a usuaris novells i inexperts en el tema que es poden trobar en la reacTable* per exemple com una instal·lació en un museu.

Pel feedback visual, concretament, aquest ampli rang d'usuaris possibles ens conduirà a la realització d'unes imatges que puguin ser interpretades per qualsevol dels usuaris, és a dir hauran de ser prou naturals i intuïtives perquè puguin ser enteses per tots els usuaris possibles.

4.2 Requeriments del sistema

Després d'haver contextualitzat acuradament el projecte i d'haver definit el perfil dels seus usuaris estem preparats per fixar quins seran els requeriments que ha de complir l'aplicació per tal d'assolir tots els objectius que s'han marcat. D'aquests requeriments alguns venen implícits en els objectius del projecte, altres venen imposats des de dalt, altres venen donats per la tecnologia existent, i altres els he fixat jo mateix.

Els requeriments s'han dividit en tres categories, les típiques en la enginyeria del software, i que són les següents:

- Funcionals: Són els que enumeren quines són les funcionalitats que ha d'oferir el mòdul de visualització, és a dir expliquen què ha de fer l'aplicació.
- No funcionals: Aquests són els requeriments que no són propis de la funcionalitat de l'aplicació, sinó que venen donats per altres factors. Dit d'una altra manera expliquen quines restriccions tindrem a l'hora d'implementar i desenvolupar l'aplicació.
- Domini: Seran totes les característiques pròpies del domini al que pertany el projecte i que d'alguna manera poden influir sobre el desenvolupament d'aquest.

4.2.1 Requeriments funcionals

Des del punt de vista funcional, el mòdul de visualització de la reacTable* ha de complir amb els següents requeriments:

- S'han de generar imatges que representin l'activitat del instrument. Per fer-ho rebrà ordres des del component central de gestió i des del sintetitzador de so.
- S'ha de dibuixar una aura dinàmica a sota dels objectes actius. Les aures han d'oscil·lar i bategar en funció de les propietats del objecte al que pertanyen. Cada tipus d'objecte tindrà una aura diferent.
- S'han de representar les connexions existents entre els diferents objectes. Hi ha d'haver una diferència clara entre les connexions d'àudio i les connexions de control. Per les connexions d'àudio es projectarà la forma d'ona del so que passa entre els objectes connectats. Per les connexions de control, es projectarà un flux de partícules entre els objectes implicats.
- Les imatges creades han de ser visualment atractives i han de donar sensació de dinamisme, és a dir no ha de donar sensació de rigidesa.
- El programa ha de permetre l'ús de diferents temes visuals.

4.2.2 Requeriments no funcionals

Pel que fa als requeriments no funcionals, el mòdul de visualització haurà de complir tots els requeriments següents:

- El programa ha de funcionar baix el sistema operatiu Linux, ja que, presumiblement, el prototipus final de la reacTable* funcionarà baix aquest llenguatge. Estaria bé que funcionés també baix Windows i MacOS sobretot per a la realització de proves.
- El procés encarregat del mòdul de visualització ha de consumir el mínim possible de recursos del processador central, passant la major part de feina possible a la tarja gràfica.
- Les imatges s'han de generar a temps real, per exemple, quan un objecte sigui posat sobre la taula, l'aura pertinent a aquest objecte ha de ser dibuixada instantàneament, l'usuari de la taula no ha de percebre el retràs existent entre una acció i la resposta a aquesta.
- S'ha d'aconseguir un *frame-rate* de 30 imatges per segon, per tal de donar una sensació de moviment adequada.
- La comunicació amb el component central de gestió i amb el sintetitzador anirà per xarxa i seguirà el protocol UDP.

4.2.3 Requeriments del domini

El sistema es veurà afectat per un conjunt de restriccions imposades pel domini al que pertany, la reacTable condicionarà al sistema de la següent manera:

- El mòdul s'haurà d'afegir al prototipus existent de la reacTable*, per tant haurà d'encaixar dins l'arquitectura actual de la reacTable*.
- Les visualitzacions creades, així com els temes visuals proporcionats, s'hauran de poder adaptar a les diferents situacions en que es pot trobar la reacTable*. El sistema ha de poder canviar d'aspecte en funció de l'àmbit en que es troba, si s'utilitza la taula com a instrument en un concert no es el mateix que si aquesta està com a instal·lació en un museu.
- El sistema s'ha d'adaptar a les especificacions de la reacTable*. Així, per exemple, quan és diu que la reacTable* ha de ser apte tant per persones no iniciades en la música com per professionals de la música electrònica, el feedback aportat pel mòdul de visualització ha de servir tant pels uns com pels altres.

5. Eines utilitzades

5.1 Paradigma de programació

En l'enginyeria del software un paradigma de programació és la forma mitjançant la qual un el dissenyador d'un sistema implementa la solució al problema que se li ha plantejat. Així doncs, en funció del paradigma escollit, canvia la forma d'abordar els requeriments, el disseny i la implementació del sistema.

5.1.1 Paradigmes existents

Els paradigmes de programació reconeguts en l'enginyeria del software són els següents:

- Programació procedural o imperativa: El programa consisteix en una seqüència d'instruccions a ser executades de forma lineal i imperativa per l'ordinador. L'execució de cada una d'aquestes instruccions canvia l'estat intern de la màquina.
- Programació funcional: Es tracta d'un paradigma que intenta resoldre el problema existent de manera purament matemàtica, és a dir la intenció es trobar una funció que ens doni la resposta al problema. Aquesta funció pot estar composada d'altres funcions.
- Programació lògica: És un paradigma basat en la lògica de predicats. Enlloc de programar la solució del problema es tracta de definir-lo lògicament. La solució s'obté per deducció.
- Programació orientada a objectes: Es basa en fer una representació del món real aplicada a la programació. És a dir, es tracta de "construir" un conjunt d'objectes, que encapsulen les dades del programa i les operacions que es poden realitzar sobre aquestes dades i que es coordinen mitjançant un pas de missatges que pretén obtenir la solució al problema.
- Programació concurrent: Aquest paradigma considera el programa com un conjunt de processos, on cada procés s'executa línea a línea, mentre que tots s'executen concurrentment i es comuniquen i sincronitzen la seva execució.

Encara que qualsevol problema que sigui resoluble mitjançant un dels paradigmes anteriors serà també resoluble mitjançant qualsevol dels altres quatre, hi ha certs elements que fan que una solució utilitzant un paradigma sigui més òptima que utilitzant-ne un altre. Així, per exemple per aplicacions de tractament del llenguatge natural és més òptim utilitzar el paradigma lògic, i en canvi, per representar una escena en tres dimensions formada per varis objectes serà més intel·ligent utilitzar la programació orientada a objectes.

5.1.2 El paradigma escollit

Inicialment podem descartar la solució de la programació concurrent, pel simple fet que no necessitem crear varis processos per dur a terme la nostra tasca, potser si que es podria considerar pel que fa al projecte global de la reacTable*. També podem descartar la programació funcional i la programació lògica principalment perquè cap dels llenguatges que treballen amb aquests paradigmes dona facilitats per la programació gràfica. A més donats els requeriments de l'aplicació, si s'utilitzessin aquests paradigmes, presumiblement, s'utilitzarien massa recursos del processador.

Per tant ens queden la programació procedural i la programació orientada a objectes. Ambdós paradigmes podrien ser vàlids, però el paradigma que més s'adequa a les necessitats del projecte a realitzar és el paradigma de la programació orientada a objectes. Aquest paradigma ens donarà el grau d'abstracció necessari per poder tractar adequadament els objectes físics de la taula, donant-los una correspondència entre ells i els objectes de programació, que es podrà fer més detallada mitjançant herència.

5.2 El llenguatge de programació

5.2.1 Llenguatges de programació orientada a objectes

Després d'haver decidit el paradigma de programació que s'utilitzarà en el transcurs del projecte, s'ha de determinà quin serà el llenguatge de programació més adient per dur a terme la nostra tasca. Així, haurem de triar un dels llenguatges de programació orientats a objectes existents; les alternatives més representatives són el C++, el Java, el SmallTalk, o el C#. D'aquests quatre, el més interessant pels nostres interessos a simple vista sembla ser el C++.

El C++ fou creat al 1980 per Bjarne Stroustrub [21], un enginyer d'AT&T, baix el nom de "C amb classes" i és una evolució directe des de un llenguatge del paradigma procedural com és el C. Durant els anys següents el C++ va anar evolucionant, afegint herència, polimorfisme i altres conceptes. A l'any 1990 se'n va declarar un estàndard ANSI i vuit anys després es va fixar l'estàndard ISO. Aquest llenguatge, doncs, ens permet gaudir de les propietats i l'abstracció de l'orientació a objectes, però també ens ofereix la possibilitat d'emprar el paradigma procedural, pel fet que és una evolució del C.

5.2.2 Per què el C++?

La forma més clara de mostrar perquè ha estat el C++ el llenguatge escollit és mitjançant una llista d'avantatges i desavantatges de fer servir aquesta solució pel projecte.

Avantatges del C++

- Estàndard: El fet que aquest llenguatge sigui un estàndard és fonamental, ja que els seus dos competidors més directes són el Java, que és propietat de Sun Microsystems, i el C# que és propietat de Microsoft. Per tant, sempre serà millor utilitzar una eina estàndard, i més tenint en compte que l'aplicació ha de córrer baix Linux amb tots els problemes que suposaria això si la implementació es fes en C#.
- Flexibilitat: La possibilitat d'utilitzar el paradigma procedural dins un llenguatge orientat a objectes fa que el C++ sigui molt més flexible que la resta dels seus competidors. El fet de poder-se allunyar de l'abstracció en un moment donat fa molt més interessant el C++ que no pas altres llenguatges com el Java o el C++.
- Eficiència: Als requeriments s'ha especificat clarament que l'aplicació ha d'utilitzar el mínim de recursos possibles del processador. Si utilitzem Java o C# és molt difícil assolir aquest requeriment, ja que aquests llenguatges fan servir una màquina virtual per interpretar el *byte-code* que han generat els seus compiladors, i això afegeix una certa lentitud als programes, que en aquest cas podria ser crítica.
- Suport: El fet de que aquest llenguatge hagi estat molt utilitzat durant els últims anys i que hagi estat molt popular, ha contribuït a que s'hagin creat un nombre molt gran de llibreries que li donen suport. Moltes d'aquestes llibreries són lliures i per tant es podran utilitzar en aquest projecte. En el cas que ens ocupa això és un bon punt a favor, ja que haurem de tractar amb gràfics, i són varies les llibreries que faciliten l'ús de gràfics baix C++.

Desavantatges del C++

- L'aprenentatge d'aquest llenguatge no és gens fàcil i la seva estructura a vegades pot semblar caòtica.
- La flexibilitat que ens dona té també un punt obscur, i és que a vegades tanta flexibilitat acaba donant errors inesperats pel programador. L'exemple típic és l'ús de punters que permet treballar directament sobre les direccions de memòria, això pot millorar l'eficiència del programa, però a vegades pot dur també molts maldecaps.

- La mancança d'utilitats en el mateix llenguatge, és a dir, la no existència de classes per gestionar piles, llistes o altres estructures bàsiques. Però cal dir que aquest punt està ben cobert per la *Standard Template Library*, què és una col·lecció de llibreries estàndard de C++ que faciliten l'ús d'estructures bàsiques, les operacions d'entrada/sortida o la serialització.

Com és pot apreciar hi ha més avantatges que desavantatges en l'elecció del C++, i a més les desavantatges són menys significatives. Així, com es pot comprendre, C++ serà el llenguatge en que es desenvoluparà aquest projecte.

5.3 Les llibreries utilitzades

Una vegada escollit el llenguatge de programació més adient pel projecte hem de començar a estudiar quines llibreries seria interessant utilitzar. Donat que el projecte no tant sols ha de tractar amb gràfics sinó que ha de fer el possible per concentrar el seu treball en la tarja de vídeo per alliberar al processador central de càlculs, seria interessant trobar una llibreria que ens deixi accedir fàcilment al hardware gràfic, és a dir, que realitzi una abstracció del hardware gràfic i que pugui ser utilitzada des de C++. Actualment, són dues les llibreries punteres en aquest àmbit, d'una banda tenim el DirectX, i de l'altre l'OpenGL. Però el DirectX queda descartat des del principi perquè es tracta d'una llibreria propietat de Microsoft que només funciona sota Windows, i donat que el projecte ha de córrer sota Linux no podem comptar amb aquesta opció ja que va en contra dels requeriments inicials. Així doncs, s'utilitzarà OpenGL.

5.3.1 La llibreria OpenGL

OpenGL significa *Open Graphics Library*, és a dir llibreria de gràfics oberta. Els seus inicis daten del 1982 a les oficines de Silicon Graphics [22], on els seus enginyers, la feien servir com a estàndard per a projectes interns. La seva intenció era crear una API per al desenvolupament d'aplicacions amb gràfics 2d i 3d independent de la plataforma on es treballava. Fins aquell moment cada venedor de targes gràfiques tenia les seves pròpies llibreries, lo qual suposava un suplici pels desenvolupadors de software que volien que els seus productes funcionessin en múltiples plataformes hardware.

Durant els anys de treball intern a la companyia americana la llibreria va anar creixent i, finalment, en vista de com estava la situació, al 1994 Silicon Graphics va presentar OpenGL en el SIGGRAPH aquell any. L'OpenGL, com era d'esperar va tenir un gran èxit i finalment s'ha acabat situant com l'estàndard de facto per a gràfics més conegut, lliure pels desenvolupadors de software, i no pels de hardware, que han d'adquirir una llicencia.

Actualment, aquest estàndard està gestionat pel OpenGL ARB, que significa *Architecture Review Board*, i està format per professionals i investigadors tant del software com del hardware gràfic. Gestionar un estàndard sobre una tecnologia que no atura de créixer en tot moment és molt difícil, ja que per una banda tenim els fabricants de hardware que no deixen d'afegir novetats a les seves targes gràfiques i per l'altre els pobres desenvolupadors que volen mantenir l'estàndard estable. Per aquest motiu molta gent dubtava de la continuïtat de l'OpenGL, però en vista que s'acaba de treure l'especificació per a l'OpenGL 2.0, sembla que ens queda OpenGL per força temps.

Com hem dit l'OpenGL és una interfície software per abstreure el hardware gràfic de la nostra aplicació. Aquesta interfície està formada per més de 300 comades diferents que serveixen per especificar els objectes i les operacions necessàries per c aplicacions interactives amb gràfics tant en dues com en tres dimensions. Com a curiositat, podem dir que OpenGL està preparat fins i tot per treballar també en els casos en que l'ordinador que mostra els gràfics no és el mateix en que s'executa l'aplicació, per aconseguir-ho segueix una arquitectura client-servidor, on el client i el servidor poden ser dos ordinadors diferents connectats mitjançant una xarxa.

Programa d'exemple en OpenGL

La millor forma de veure com és OpenGL és mitjançant un exemple simple que ens donarà una idea de com s'utilitza OpenGL i de com és de senzill realitzar operacions que anys enrere eren bastant més costoses. L'exemple està extret de la guia de programació d'OpenGL.

```
#include <totelquenecessites.h>

main() {

   InicialitzaUnaFinestra();

   glClearColor (0.0, 0.0, 0.0, 0.0);         // Fixem el color de fons
   glClear (GL_COLOR_BUFFER_BIT);             // Netegem el buffer de color
   glColor3f (1.0, 1.0, 1.0);                 // Triem el color
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);    // Fixem la perspectiva
   glBegin(GL_POLYGON);                       // Diem el que volem dibuixar
      glVertex3f (0.25, 0.25, 0.0);           // Enviem als vèrtexs
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
   glFlush();                                 // Ens assegurem que s'hagin
                                              // enviat les comandes
   ActualitzaLaFinestra();
}
```

Llistat 1. Un senzill programa en OpenGL

I el que veuríem a la nostra pantalla a l'executar aquest programa és el que veiem a la pròxima figura.

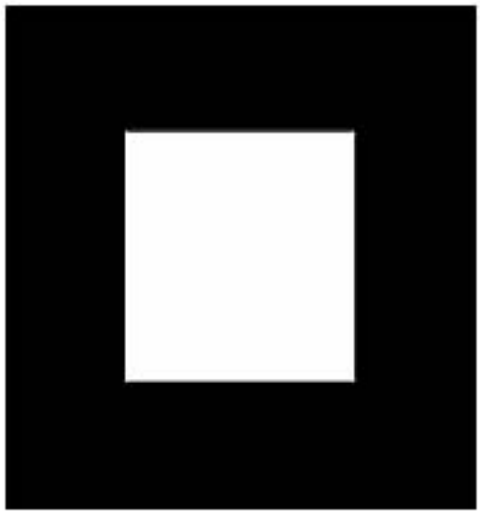


Figura 30. Resultat del programa del llistat 1.

La intenció que OpenGL sigui una interfície independent de la plataforma que pugui ser implementada en moltes plataformes hardware diferents, fa que no s'hagin inclòs dins la llibreria comandes per a la creació i gestió de finestres ni tampoc instruccions per gestionar l'entrada del usuari. Com podem veure en el programa exemple es necessita una caixa negra que ens inicialitzi les finestres i ens ajudi en la seva gestió. Així després d'analitzar i comentar els punts importants de l'OpenGL el nostre pròxim pas serà l'elecció d'una llibreria que ens serveixi per a l'abstracció del sistema de finestres.

L'OpenGL com una màquina d'estats

Segons els seus creadors, l'OpenGL és una màquina d'estats. El que fa l'usuari de la llibreria és seleccionar uns estats o modes d'actuació i aquests es mantenen així fins que l'usuari els torna a canviar, mentrestant aquest va enviant vèrtex a l'OpenGL que aquest pinta, situa i transforma en funció de l'estat actual. Els modes fan referència als colors, a les transformacions, als patrons... això s'entén molt bé amb l'exemple del color; tu dius a OpenGL que pintaràs en color blanc, com al codi d'exemple anterior, doncs, fins que no li diguis que pintaràs en un altre color tots els vèrtex que enviïs a OpenGL seran pintats de color blanc.

El Pipeline d'OpenGL

La majoria d'implementacions d'OpenGL tenen un ordre similar d'operacions, és a dir tots segueixen un esquema bàsic de funcionament, que és l'OpenGL *Pipeline*. Aquest esquema el podem veure a la figura 31. El dibuix mostra el camí que segueixen els vèrtexs introduïts dins l'arquitectura del OpenGL, és a dir, mostra cada una de les etapes per les que passen els vèrtexs. Una forma fàcil d'entendre aquest concepte és pensant en OpenGL com una cadena de muntatge on se li entren els vèrtexs, aquests passen per una sèrie de maquines on se'ls va modificant fins que arriben a la pantalla de l'ordinador.

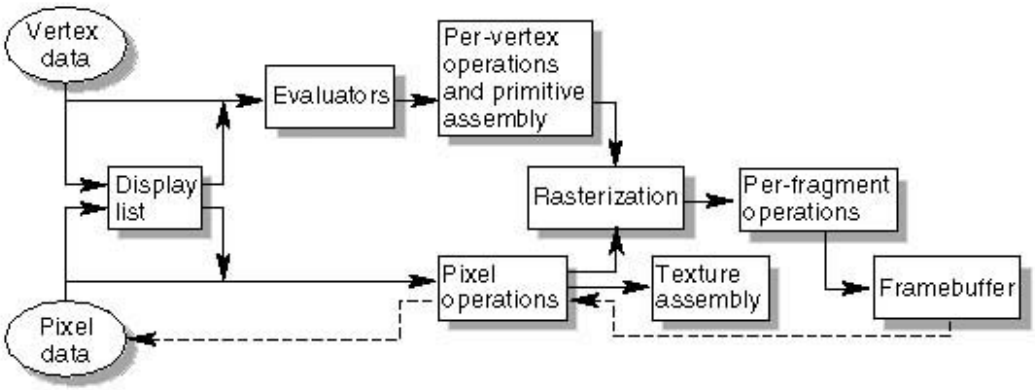


Figura 31. Ordre d'operacions del OpenGL.

Avantatges i inconvenients d'emprar OpenGL

La millor manera de justificar l'elecció del OpenGL és mitjançant la realització d'una llista d'avantatges i una de desavantatges de realitzar una aplicació gràfica com la que ens ocupa i veure cap a quina banda cau la balança. Els avantatges que he trobat són els següents:

- Està considerat com un estàndard amb tot lo que això comporta. Té el suport de la indústria del sector.
- Es tracta d'una llibreria lliure i multiplataforma, que dona resultats consistents independentment del hardware sobre el que treballa.
- La interfície que ofereix és clara, senzilla i relativament fàcil d'aprendre.
- Permet aprofitar les capacitats del nostre hardware gràfic d'una manera transparent, realitzant una gran abstracció.
- Porta uns quants anys funcionant sense problemes sobre diferents plataformes, el que demostra una gran estabilitat.
- És escalable, donat que pot funcionar tant en ordinadors personals com en supercomputadors.
- Està molt ben documentada.

Però no tot són avantatges, l'OpenGL té els següents inconvenients:

- La gestió de les extensions és bastant complicada i varia segons la plataforma, fet que fa que l'aprofitament màxim del hardware gràfic a vegades sigui bastant difícil d'aconseguir.
- No ofereix un sistema d'abstracció del sistema de finestres. Encara que, com veurem, es sol associar la llibreria GLUT al OpenGL i aquesta si que ofereix abstracció de finestres.

Com podem apreciar la llista d'avantatges és molt més gran que la llista d'inconvenients. Així, donat que la opció d'utilitzar DirectX ha quedat descartada des del principi, queda més que justificat l'ús del OpenGL.

5.3.2 L'abstracció del sistema de finestres

Com s'ha comentat abans, un dels objectius d'OpenGL era que el seu model de representació no depengués del sistema de finestres sobre el que treballava. El resultat és que OpenGL és independent del sistema de finestres.

Les operacions del sistema de finestres com la creació d'una finestra o la gestió dels seus events no estan incloses dins OpenGL i es el sistema de finestres el que se'n ha d'encarregar. No obstant, hi ha d'haver necessàriament una interacció entre el sistema de finestres i l'OpenGL, ja que s'ha de crear i aplicar el context OpenGL a la finestra on volem fer la representació. Aquestes interaccions estan descrites en una especificació que si depèn del sistema de finestres que utilitzem. Així, per exemple, la especificació GLX, descriu l'estàndard d'interacció entre OpenGL i el X Window System.

El no incloure operacions sobre finestres en OpenGL fa que la portabilitat del codi de renderització sigui molt més portable, però també fa que si el programa està escrit utilitzant la interfície del sistema de finestres natiu, el programa sigui dependent del sistema de finestres.

Per omplir aquest buit existent entre OpenGL i el sistema de finestres han sorgit una sèrie de llibreries o *toolkits* que s'encarreguen de la gestió de finestres, amb la intenció de fer portable no només una part del codi, sinó tot el codi d'un programa basat en OpenGL. La més coneguda d'aquestes llibreries possiblement sigui GLUT (*OpenGL Utility Toolkit*) i és molt útil per programar rapida i senzillament aplicacions OpenGL, i està dissenyada exclusivament per això. L'altre opció és la llibreria SDL (*Simple DirectMedia Layer*) que vindria a ser una API de desenvolupament multimèdia i multiplataforma lliure, que dona un gran suport a OpenGL però que a diferència de GLUT, no està dissenyada exclusivament amb aquest objectiu.

El que hem de fer és decidir quina d'aquestes dues llibreries s'adapta més als nostres interessos, és a dir hem de triar la llibreria que ens ajudi més a assolir els requeriments inicials del projecte. A continuació farem una comparativa entre aquestes dues llibreries estudiant els punts més interessants per a la nostra aplicació per després poder decidir amb quina de les dues ens quedarem.

- Multiplataforma: Ambdues llibreries són multiplataforma i funcionen perfectament sota Windows i sota Linux.
- Llenguatge de programació: Les dues funcionen nativament en C++, que és el llenguatge que emprarem, encara que SDL ha estat portada també a molts altres llenguatges.
- Gestió de finestres: Totes dues ens ofereixen la creació, gestió o tractament d'events de les finestres, així com el suport d'aquestes a OpenGL. Però mentre GLUT només es concentra en OpenGL, SDL ens ofereix més opcions no necessariament relacionades amb OpenGL.

- Gestió del temps i dels *threads*: Tant SDL, com GLUT tenen funcions pel control del temps i temporitzadors, però els controls d'aquestes opcions són més acurats en SDL, que permet, per exemple, una resolució de 10 mil·lèsimes de segon. Pel que fa al *multithreading*, SDL permet una gestió completíssima dels *threads* del programa, en canvi això no existeix en GLUT, que l'únic que ens ofereix és un sistema de *callbacks*.
- Eficiència: Segons el que he pogut esbrinar, pel que sembla ser les aplicacions basades en SDL són lleugerament més eficients que les basades en GLUT.
- Operacions comunes implementades: SDL porta una sèrie de funcions comunes en la programació multimèdia implementades que GLUT no du. Així, per exemple, SDL dur funcions com la lectura de *bitmaps* ja incorporades.
- Facilitat d'ús: Les dues llibreries són fàcils d'aprendre i d'utilitzar. Però hi ha que reconèixer que GLUT és molt més simple.

No hem repassat un per un tots els avantatges que aquestes llibreries ens poden oferir, però crec que si hem vist suficient per veure que mentre que GLUT és molt més simple i fàcil d'aprendre i utilitzar que SDL, la segona és molt més completa i ens dona més possibilitats, com el *multithreading* que resultarà útil pel projecte. Des del meu punt de vista, GLUT és molt útil per el disseny ràpid d'aplicacions gràfiques basades en OpenGL, però sobretot perquè quan l'utilitzes només t'has de preocupar dels gràfics, és a dir crec que és ideal per a demostracions sobre un efecte gràfic concret o coses similars. En canvi, SDL és una llibreria més professional, que es pot utilitzar per a projectes més complexes. Per tant, crec que SDL serà més útil que GLUT per al projecte.

La llibreria SDL

La Simple DirectMedia Layer [23] és una llibreria multimèdia multiplataforma i lliure creada per Sam Lantiga. El seu objectiu és donar facilitats als seus usuaris per la gestió d'àudio, de vídeo tant 2D com 3D, i dels dispositius d'entrada més comuns, el teclat, el mouse i *joysticks*. SDL suporta una gran quantitat de plataformes, entre les quals troben Linux, Windows o MacOS i pel que fa als llenguatges de programació, està escrita en C, però funciona nativament en C++ i s'ha portat a molts altres llenguatges. Es tracta d'una llibreria que s'ha utilitzat sobretot en la realització de jocs per Linux, reproductors de vídeo, emuladors i altres aplicacions multimèdia.

La gestió de finestres de SDL és el que més ens interessa i ens ajudarà en la creació i gestió de finestres i, juntament amb l'apartat de vídeo permet establir qualsevol profunditat de color, escriure directament sobre el *buffer* del marc gràfic, la utilització de varies capes amb transparències, suport per a la utilització d'OpenGL, així com carrega de *bitmaps* o altres funcions típiques. Tot això emprant l'acceleració hardware. L'API ens permet una gestió fàcil dels events de les finestres així com l'entrada de teclat, mouse o *joystick*, i ens permet crear els nostres propis events. Ens ofereix també una secció d'àudio que facilita la reproducció de fitxers de so i de CDs musicals, però no s'utilitzarà en aquest projecte donat que l'àudio de la reacTable* és gestionat per un altre mòdul. El que si utilitzarem és la gestió de *threads* que ens proporciona, que ens serà molt útil, i que és bastant senzilla d'utilitzar. També incorpora temporitzadors i amb una resolució de fins a 10 ms.

Definitivament, podem dir que SDL és la llibreria ideal pel projecte, ja que a part d'oferir-nos la gestió de finestres amb suport OpenGL, ens ofereix *multithreading*, que és una característica que també necessitem. I és que si ens decantéssim per GLUT, hauríem de buscar després una llibreria per gestionar *threads*, en canvi, escollint SDL, amb una sola llibreria solucionem dos punts importants. Crec que escollint SDL, augmentem una miqueta la complexitat però guanyem en funcionalitat.

Per fer-nos una idea de com és aquesta llibreria, el millor es fer una ullada a un exemple d'una situació senzilla i típica. Així a continuació, veurem un exemple que mostra com es faria en SDL per llegir un *bitmap* i mostrar-lo per pantalla. L'exemple està extret de la documentació oficial de la llibreria i cal tenir en compte que la llibreria ha d'estar prèviament inicialitzada i el mode de vídeo ja ha d'estar fixat abans de poder cridar aquesta funció.

```
void display bmp(char *file name)
{
  SDL_Surface *image;

  /* Carreguem el fitxer BMP dins d'una superficie SDL */
  image = SDL LoadBMP(file name);
  if (image == NULL)
  {
    fprintf(stderr, "Couldn't load %s: %s\n", file_name, SDL_GetError());
    return;
  }

  /*
   * Canviem la paleta de colors del mode de video actual per la paleta
   * per la paleta de la imatge que carreguem.
   */

  if (image->format->palette && screen->format->palette)
  {
    SDL_SetColors(screen, image->format->palette->colors, 0,
                  image->format->palette->ncolors);
  }

  /* Passem la superficie de la imatge a la superficie de pantalla */
  if(SDL_BlitSurface(image, NULL, screen, NULL) < 0)
  {
    fprintf(stderr, "BlitSurface error: %s\n", SDL GetError());
  }

  SDL UpdateRect(screen, 0, 0, image->w, image->h);

  /* Alliberem la superficie SDL on tenim guardada la imatge BMP */
  SDL FreeSurface(image);
}
```

Llistat 2. Carregar i mostrar per pantalla un BMP en SDL.

5.3.3 La programació de GPUs

En els darrers anys el hardware gràfic ha millorat considerablement. Primer gràcies al naixement de les targetes acceleradores i després amb el naixement de les GPUs (Graphical Processing Unit) programables. Fins fa relativament poc, les targetes gràfiques generaven les imatges a partir d'un conjunt de funcions fixes. La tarja es podia mirar com si fos una cadena de muntatge on s'introduïen els vèrtexs per una banda, se'ls aplicava una sèrie d'operacions, on cada operació era un element de la cadena de muntatge, i com a sortida obteníem una imatge. Aquestes operacions només es podien configurar lleugerament per ajustar alguns aspectes de la imatge, com si la nostra cadena de muntatge tingués alguns commutadors. Però amb l'aparició de les GPUs programables tot això ha canviat.

Actualment, el hardware gràfic és en gran mesura programable i permet canviar les funcions, abans fixes, de processament de vèrtexs i de fragments per petits programes en llenguatge assemblador anomenats col·loquialment *shaders*. Aquest programes s'utilitzen sobretot per la creació d'efectes i permeten canviar l'aparença dels objectes, i altres efectes més complexes com poden ser variar les condicions i els algorismes d'iluminació o aplicar textures dinàmiques als objectes mitjançant la utilització de formules matemàtiques. A la següent figura podem observar el model de d'operacions de le GPUs programables. Com podem veure en aquesta figura, hi ha dos tipus de *shaders*: els *vertexs shaders* que defineixen les operacions que es realitzaran sobre cada un dels vèrtexs que passen per la GPU i els *fragment shaders* que el que fan es operar a nivell de píxel.

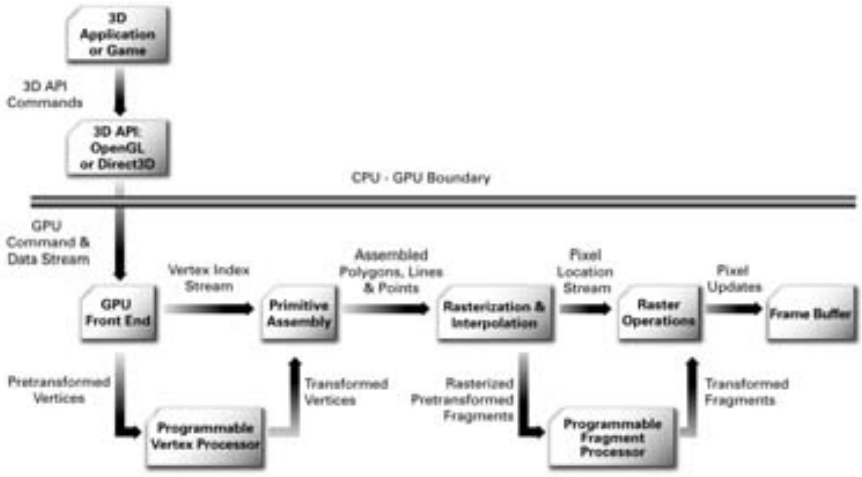


Figura 32. Model d'operacions d'una GPU programable.

Com sabem, en la teoria la programació a baix nivell dona al programador una gran flexibilitat, però en la pràctica els llenguatges de programació de baix nivell són molt difícils d'utilitzar adequadament. Per això han sorgit una sèrie de nous llenguatges dedicats exclusivament a la programació dels processadors de la targeta gràfica. Els primers llenguatges de programació de *shaders* en aparèixer foren PixelFlow, RTSL i RenderMan, posteriorment va aparèixer el HLSL que forma part del paquet DirectX de Microsoft i quasi simultàniament ho va fer el Cg de nVidia. I fa relativament poc ha sortit el GLSL (*OpenGL Shading Language)*.

De tots aquests llenguatges comentats anteriorment, els més estesos i més utilitzats per aplicacions gràfiques convencionals són el HLSL i el Cg i recentment el GLSL, que està en procés d'expansió. Així, si hagués d'escollir un d'aquests llenguatges per utilitzar-lo en el projecte només ho podria fer entre aquests tres, ja que els altres són bastant inaccessibles. Al HLSL el podem eliminar de la llista donat que només funciona baix plataformes Windows i forma part del DirectX. Per tant la decisió està entre el Cg i el GLSL. El cas és que GLSL és un llenguatge de recent creació i poques targetes gràfiques poden gaudir-ne sense utilitzar una espècie d'emulador i com que en les targetes que tinc a disposició el GLSL no pot funcionar com és degut he optat per la solució del Cg.

El llenguatge Cg

El Cg és un llenguatge d'alt nivell dissenyat per programar GPUs [25] [26]. Com el seu nom indica, Cg significa C for Graphics, aquest llenguatge està basat en la sintaxi i la filosofia de C, però està optimitzat i modificat per tal de fer més fàcil la construcció de programes que compilin en codi GPU el més optimitzat possible.

Aquest llenguatge ens serveix tant per escriure programes per al processador de vèrtexs com per al processador de fragments, dit d'una altra manera Cg ens permet crear *vertex shaders* i *pixel shaders*. Però no només això sinó que ve acompanyat de tota una arquitectura que facilita la posada en pràctica d'aquests programes que es poden utilitzar tant des de OpenGL com des de DirectX, tant sota Windows com sota Linux.

Els principals avantatges d'utilitzar llenguatges de programació de *shaders* com aquest en lloc d'utilitzar el llenguatge assemblador de les targetes gràfiques són els següents:

- Disminueix considerablement el cicle de creació de *shaders* en que s'ha d'anar canviant i provant el codi, cosa bastant important en la creació de prototipus d'efectes d'alta qualitat.
- El compilador optimitza el codi i s'encarrega de tècniques de baix nivell com poden ser la utilització de registres que a part que són difícils de realitzar poden conduir fàcilment a l'error.

- És molt més fàcil llegir i entendre el codi escrit en alt nivell, que no el codi en assemblador, amb tot lo que això comporta.
- Els programes escrits en un llenguatge d'alt nivell són més fàcils de portar a altres plataformes que no pas els programes escrits en llenguatge assemblador.

Aquests que hem vist són els avantatges de tots els llenguatges d'alt nivell per a programar GPUs, però el Cg en concret té a més aquests avantatges sobre els seus competidors:

- Es tracta d'un llenguatge multiplataforma que funciona tant amb OpenGL, com amb DirectX, tant sota Windows, com sota Linux, com sota OS X, i és suportat per hardware de nVidia, ATI, Matrox i tot el hardware programable que suporta OpenGL i DirectX.
- La seva arquitectura, en concret el Cg Runtime, facilita el pas de paràmetres entre l'aplicació i els programes de vèrtexs i fragments.
- Està totalment recolzat per part del sector, són ja moltes les empreses que es beneficien dels seus serveis. També influeix el fet que hagi estat desenvolupat a nVidia, que és un dels gegants del sector dels gràfics per ordinador.
- Està molt ben documentat. Hi ha alguns llibres i molta documentació electrònica a Internet, així com molts exemples de *shaders* que ens poden ser de gran utilitat per a la creació dels nostres *shaders*.

La millor manera de veure com és el Cg és veient un programa simple. A continuació veurem un exemple simple de Cg, extret de la pagina de NeHe [29], i la seva traducció a assemblador per poder comparar fàcilment entre les dues solucions. El *shader* que veurem és un *vertex shader*, és a dir, serà el programa que s'introduirà en la part de processament de vèrtexs de la GPU i s'executarà per cada un dels vèrtex que arribin a la targeta gràfica mentre aquest està activat. Aquest *shader* el que fa es variar la coordenada y dels vèrtex basant-se en sinusoides.

```
struct appdata                          // Primer definim una estructura
{                                       // per guardar els paràmetres que
  float4 position       : POSITION;     // rebrà el shader de l'apliació.
  float4 color          : COLOR0;
  float3 wave           : COLOR1;
};

struct vfconn                           // Aquí definim una estructura
{                                       // que guardarà els paràmetres
  float4 HPos    : POSITION;            // de sortida del shader.
  float4 Col0    : COLOR0;
};

vfconn main(appdata IN, uniform float4x4 ModelViewProj)
{
  vfconn OUT;    // Variable que contindrà la sortida del vertex
                 // shader (anirà al pixel shader si aquest existeix).

  // Canviem la coordenada Y del vèrtex en funció del sinus.
  IN.position.y = ( sin(IN.wave.x + (IN.position.x / 5.0) )
                  + sin(IN.wave.x + (IN.position.z / 4.0) ) ) * 2.5f;

  // Transformem la posició del vèrtex a l'espai de clipping homogeni
  // És una operació necesaria en aquests tipus de shader.
  OUT.HPos = mul(ModelViewProj, IN.position);

  // Posam al color de sortida el color que hem rebut com entrada.
  OUT.Col0.xyz = IN.color.xyz;

  return OUT;
}
```

Llistat 3. Un programa d'exemple en Cg.

Com podem veure en el llistat anterior, la semblança de sintaxi entre C i Cg és bastant gran. Caldria comentar l'aparició de tipus com float3, float4 o float4x4. Com és de suposar float3 és un *array* de tres *floats*, float4 és un *array* de quatre *floats* i float4x4 és una matriu de quatre per quatre *floats*. Així quan posem position.x estem accedint a la primera posició del vector position. Evidentment, aquests no són els únics tipus de vector preconstruïts en Cg, n'hi ha varies combinacions més i la seva raó de ser és que el processador de la targeta gràfica treballa directament amb aquests tipus i per tant és força interessant poder tractar amb ells a l'hora de programar.

El programa d'exemple en Cg és bastant fàcil de llegir i d'entendre, a més es tracta d'un programa curt i senzill. No pensem el mateix quan veiem la versió en assemblador del mateix programa. És el pròxim llistat.

```
MOV o[COL0].xyz, v[3].xyzx;
MOV R1, v[0];
RCP R0.x, c[4].y;
MAD R0.x, R1.z, R0.x, v[4].x;
MAD R2.xy, c[6].w, R0.x, -c[6].x;
EXP R2.y, R2.x;
ADD R0.xyz, c[5].xyzx, -R2.y;
MUL R0.xyz, R0.xyzx, R0.xyzx;
SLT R3.x, R2.y, c[6].x;
SGE R2.xyw, R2.y, c[6].yzyy;
MOV R3.yz, R2.xxyw;
DP3 R3.y, R3.xyzx, c[9].zwzz;
MAD R2.xyz, c[7].xyxx, R0.xyzx, c[7].zwzz;
MAD R2.xyz, R2.xyzx, R0.xyzx, c[8].xyxx;
MAD R2.xyz, R2.xyzx, R0.xyzx, c[8].zwzz;
MAD R2.xyz, R2.xyzx, R0.xyzx, c[9].xyxx;
MAD R2.xyz, R2.xyzx, R0.xyzx, c[9].zwzz;
DP3 R0.x, R2.xyzx, -R3.xyzx;
RCP R0.w, c[4].x;
MAD R0.w, R1.x, R0.w, v[4].x;
MAD R5.xy, c[6].w, R0.w, -c[6].x;
EXP R5.y, R5.x;
ADD R2.xyz, c[5].xyzx, -R5.y;
MUL R4.xyz, R2.xyzx, R2.xyzx;
SLT R3.x, R5.y, c[6].x;
SGE R2.xyw, R5.y, c[6].yzyy;
MOV R3.yz, R2.xxyw;
DP3 R3.y, R3.xyzx, c[9].zwzz;
MAD R2.xyz, c[7].xyxx, R4.xyzx, c[7].zwzz;
MAD R2.xyz, R2.xyzx, R4.xyzx, c[8].xyxx;
MAD R2.xyz, R2.xyzx, R4.xyzx, c[8].zwzz;
MAD R2.xyz, R2.xyzx, R4.xyzx, c[9].xyxx;
MAD R2.xyz, R2.xyzx, R4.xyzx, c[9].zwzz;
DP3 R4.x, R2.xyzx, -R3.xyzx;
ADD R0.x, R4.x, R0.x;
MUL R1.y, R0.x, c[4].z;
DP4 o[HPOS].x, c[0], R1;
DP4 o[HPOS].y, c[1], R1;
DP4 o[HPOS].z, c[2], R1;
DP4 o[HPOS].w, c[3], R1;
```

Llistat 4. El mateix programa del llistat 3 però en codi assemblador.

A simple vista podem veure que no només el programa és molt més llarg sinó que molt més difícil de llegir i entendre el que fa. La veritat és que veient aquest exemple sobren les paraules. És per aquesta raó que han anat sorgint aquests nous llenguatges de *shaders* i que com es pot veure ens ajuden bastant en la creació d'aquests programes.

Com és de suposar aquest programa ell tot sol no fa absolutament res, hem de tenir una aplicació que enviï els vèrtexs al hardware gràfic. En aquest cas, el que s'ha fet és utilitzar un programa en OpenGL que envia una malla de triangles que formen un quadrat a la nostra targeta gràfica. En la següent figura veurem primer el resultat que ens ofereix el programa OpenGL sense tenir el programa Cg activat, i després el veurem en dos estats diferents mentre el programa Cg està en funcionament.

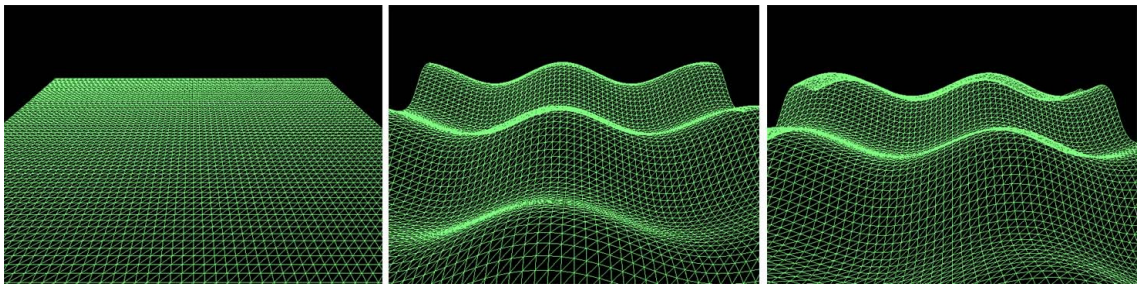


Figura 33. A l'esquerra veiem l'aplicació gràfica sense el *shader* en funcionament, les altres dues imatges mostren dos instants en que el *shader* està funcionant.

Arribats a aquest punt, i havent vist en que consisteix la programació de GPUs i els mitjans que ens proporciona el Cg per a dur-la a terme, teòricament podríem estar en condicions de respondre a la pregunta de si Cg resultarà útil o no per aquest projecte. Sincerament, la meva resposta és que no ho sé, però serà força interessant esbrinar-ho. El fet que es tracti d'una tecnologia relativament nova, i donat que la meva experiència en Cg prèvia al projecte és pràcticament nul·la, fa difícil poder respondre de manera segura. Per tant s'estudiarà Cg i s'intentarà aplicar al projecte en punts on pugui ser útil i al final del projecte veurem si ens és realment útil o si podríem haver-ne prescindit.

6. Anàlisi i disseny

Un cop s'han fixat els requeriments que ha de tenir l'aplicació i hem decidit les eines que utilitzarem per a complir-los només ens queda posar-nos a analitzar i dissenyar el que serà l'aplicació. En aquest capítol s'analitzaran en profunditat els requeriments exposats en el capítol 4 i seran convertits en una aplicació mitjançant la utilització de les eines comentades en el capítol anterior.

El primer que haurem de fer abans de posar-nos a cercar solucions al problema planejat i abordar els requeriments, serà analitzar el domini en que estarà l'aplicació.

6.1 Anàlisi del domini

Aquest projecte pertany a un domini concret del que ja hem parlat en els anteriors capítols. La reacTable* és el domini de l'aplicació. Com ja hem comentat el sistema actual de la reacTable* està implementat d'una manera totalment modular i aquest projecte s'encarrega d'un d'aquests mòduls que formen la reacTable*. Aquest projecte pretén desenvolupar el mòdul de síntesi de gràfics, però evidentment haurem d'analitzar amb quins components del sistema s'ha de comunicar aquest mòdul i haurem de pensar en com ho farà.

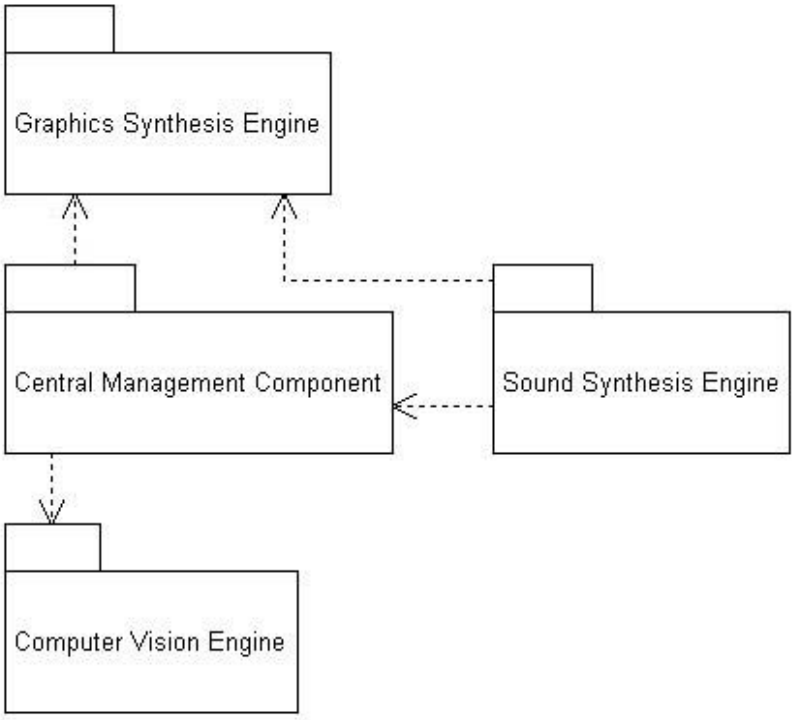


Figura 34. L'arquitectura modular de la reacTable*.

No repetirem tot lo explicat en el capítol 3, on s'explicaven els diferents components de la reacTable* un per un, sinó que anirem directe als que es comuniquen amb el nostre mòdul. Observant els requeriments del mòdul i el disseny global de la reacTable*, podem arribar a la conclusió que el mòdul de feedback visual només es comunica amb el mòdul de síntesi de so i amb el component central de gestió. La comunicació en els dos casos és unidireccional, el nostre mòdul rep informació del mòdul de síntesi de so i del component central de gestió però no envia informació a cap dels dos. Per tant, pel que fa al domini, en principi, només s'ha de tenir en compte la comunicació del mòdul amb aquests altres dos components.

La comunicació és realitzarà per xarxa mitjançant el protocol UDP com està especificat clarament en els requeriments. Així que l'aplicació s'haurà de comunicar mitjançant l'utilització de *sockets* UDP amb aquests dos mòduls. Millor dit, l'aplicació haurà de tenir dos *threads* que facin de servidors UDP, un que escolti els missatges que li arribaran des del component central de gestió i un que escoltarà els missatges que arribin des del component de síntesi sonora.

6.1.1 La comunicació amb el component central de gestió

El component central de gestió enviarà missatges al mòdul de visualització cada vegada que succeeixi alguna cosa a sobre la taula per informar-lo de l'estat del sistema en cada instant. Si un usuari de la taula posa un objecte sobre aquesta el component de gestió enviarà un missatge al mòdul de visualització dient-li que hi ha un nou objecte i aquest haurà de pintar l'aura pertinent a l'objecte, si l'usuari manipula l'objecte canviant la seva posició o el seu angle el mòdul de visualització serà informat dels canvis i així amb totes les possibles accions que es poden realitzar sobre la taula.

Els missatges que rebrà el mòdul de visualització del component central de gestió seran cadenes de caràcter amb un format concret i seguiran un protocol molt simple. Aquests missatges han estat predefinits per l'equip de treball de la reacTable*. També s'ha decidit que aquesta connexió vagi pel port número 3400. A continuació, en la següent taula, veurem els missatges que pot enviar el mòdul de gestió i la seva corresponent explicació.

Missatge	**Explicació**
clr;	S'ha fet un *reset* de l'instrument.
new osc;	Nou objecte de tipus oscil·lador.
new flt;	Nou objecte de tipus filtre.
new eff;	Nou objecte de tipus efecte.
new lfo;	Nou objecte de tipus oscil·lador de baixa freqüència.
new smp;	Nou objecte de tipus *sampler*.
new wav;	Nou objecte de tipus ona.
new gra;	Nou objecte de tipus sintetitzador granular.
new env;	Nou objecte de tipus envolupant.
new out;	Nova sortida d'àudio.
side n s;	L'objecte numero n està sobre la seva cara s.
pos n x y α;	L'objecte número n està a la posició x, y amb una orientació α.
act n 1;	L'objecte número n ha estat activat.
act n 0;	L'objecte número n ha estat desactivat.
con snd n1 0 n2 0;	S'ha creat una connexió d'àudio entre l'objecte n1 i l'objecte n2.
dis snd n1 0 n2 0;	S'ha desfet la connexió d'àudio entre l'objecte n1 i l'objecte n2.
con ctr n1 0 n2 0;	S'ha creat una connexió de control entre els objectes n1 i n2.
dis ctr n1 0 n2 0;	S'ha desfet la connexió de control entre els objectes n1 i n2.
hlk n1 n2;	La connexió entre l'objecte n1 i l'objecte n2 ha esdevingut forta.

Taula 1. Els missatges rebuts del component de gestió i la seva pertinent explicació.

Com es pot observar, els missatges són bastant simples i a la vegada molt útils, amb només aquests missatges podem saber en tot moment la posició, l'orientació i l'estat de cada un dels objectes de la taula així com les connexions entre ells. Caldria comentar el significat de l'últim missatge de la taula, una connexió forta vol dir que la connexió entre aquests dos objectes no es desfà fins que un dels dos objectes no abandoni la taula. Per fer que una connexió sigui forta hem de fer que els dos objectes es toquin, a partir d'aquell moment la connexió entre ells dos serà forta.

Es d'esperar que la comunicació entre el mòdul de gestió i el mòdul de visualització sigui constant i importantíssima, ja que sense aquesta connexió el mòdul de visualització no sap que és el que ha de dibuixar. Per tant s'haurà de parar especial atenció a com el mòdul de visualització gestiona la informació que li arriba per aquest canal comunicatiu.

6.1.2 La comunicació amb el component de síntesi de so

La comunicació existent amb el mòdul de síntesi de so i el mòdul de visualització té un número de missatges més petit que no pas la comunicació entre el mòdul de visualització i el component de gestió encara que possiblement tingui una freqüència més elevada. En aquest cas, com hem dit anteriorment, la comunicació també és unidireccional, també amb UDP però pel port 3500. El component de síntesi de so envia cap al mòdul de visualització la forma de la ona de cada objecte d'àudio que hi ha actiu sobre la taula i l'amplitud de l'oscil·lació de cada un dels objectes. La forma d'ona rebuda serà pintada com a connexió entre dos objectes d'àudio o entre l'objecte d'àudio al que pertany el so i el centre de la taula. En canvi, l'amplitud s'utilitzarà per tal de graduar les dimensions de l'aura de l'objecte en qüestió de manera que s'aconsegueixi reflectir visualment la freqüència interna de cada un dels objectes.

La sintaxi dels missatges d'aquesta comunicació seguirà més o menys la mateixa filosofia que en la comunicació amb el component central de gestió, però variarà un poc degut a la naturalesa dels missatges. Per l'amplitud de l'aura s'utilitzarà un missatge que començarà amb un identificador de missatge seguit del número de l'objecte seguit d'un número en coma flotant que representarà l'amplitud, és a dir més o menys el mateix que en els casos anteriors. En canvi, pel que fa a la forma d'ona el que es farà és enviar pel *sockets* UDP una estructura com la que veurem a continuació.

```
typedef struct
{
  short Id;
  short NumSamples;
  char* Buffer;
} AudioMessage;
```

Llistat 5. Estructura d'un missatge de tipus àudio.

Per tant el que s'enviarà serà un identificador de l'objecte, seguit del número de mostres que tindrà la forma de ona, seguit d'un vector de tantes mostres com indica el número anterior on hi ha, evidentment, les mostres que després el mòdul de visualització haurà de mostrar per pantalla sortint de l'objecte adequat i dirigint-se al lloc adequat.

Ambdós missatges serveixen per indicar el que ha de mostrar el mòdul de visualització en un instant concret. Donat que es pretén que l'aplicació gràfica funcioni a 30 frames per segon, lo ideal seria que per a cada objecte de la taula s'enviessin els dos missatges, el de l'amplitud i el de la forma d'ona, a cada frame, és a dir 60 missatges per segon per a cada objecte actiu. Això ens dona un tràfic considerable que haurem de gestionar d'una manera adequada.

6.2 Com s'afrontaran els requeriments

Una vegada s'han aclarit tots els punts del domini que poden influenciar l'aplicació, podem endinsar-nos en la manera com volem que el sistema compleixi els requeriments inicials. Per a fer-ho profunditzarem en cada un dels requeriments i decidirem el que volem fer per aconseguir que l'aplicació el compleixi.

6.2.1 Generació de les imatges en temps real

Aquest és un requeriment molt general que intentarem assolir mitjançant la construcció d'una aplicació en C++ que utilitzi els serveis de les llibreries SDL i OpenGL per a la generació de les imatges.

Per aconseguir generar les imatges haurem de fer que hi hagi una comunicació fluida entre l'aplicació i els dos components de la reacTable* que han d'enviar la informació a mostrar.

6.2.2 Mostrar aures sota els objectes actius

La visualització de les aures dels objectes actius és un dels requeriments bàsics del sistema. Com sabem el feedback visual ha de reflectir en tot moment l'estat en que es troba el sistema, per tant quan un objecte s'activa, s'ha de mostrar una aura a sota de l'objecte, així, l'aplicació haurà de dibuixar un element gràfic que representarà ser l'aura de l'objecte a un lloc determinat de la pantalla i haurà de bategar i rotar en funció de l'estat de l'objecte.

Les aures consistiran en un o varis *bitmaps* superposats mitjançant l'ús de l'*aplha blending* de l'OpenGL. Hi haurà dues possibilitats a l'hora de mostrar les aures, això ens resultarà molt útil per complir un altre requeriment que consisteix en permetre canviar el tema visual de l'aplicació. La primera serà la d'utilitzar *bitmaps* en escala de grisos als que canviarem el color a l'hora de pintar i que podrem superposar utilitzant varies capes mitjançant l'ús de la transparència.

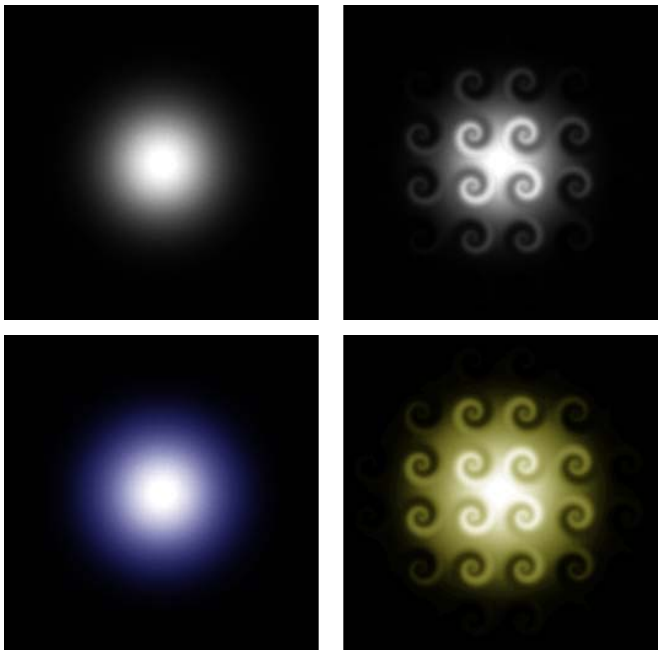


Figura 35. A dalt podem veure dos *bitmaps* en escala de gris, a sota els dos mateixos *bitmaps* però havent-los aplicat un color tal com farà l'aplicació.

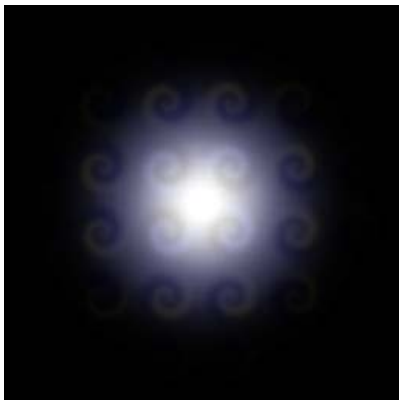


Figura 36. La fusió dels dos *bitmaps* anteriors amb el color i l'*alpha blending* aplicat, això és la primera possibilitat de representació d'aures de l'aplicació.

La segona opció serà la d'emprar *bitmaps* directament a color. Per aconseguir-ho haurem de fer servir màscares dels *bitmaps* colorats per tal d'aconseguir la transparència adequada en els llocs on no hi hagi res pintat en les imatges a color que utilitzem.

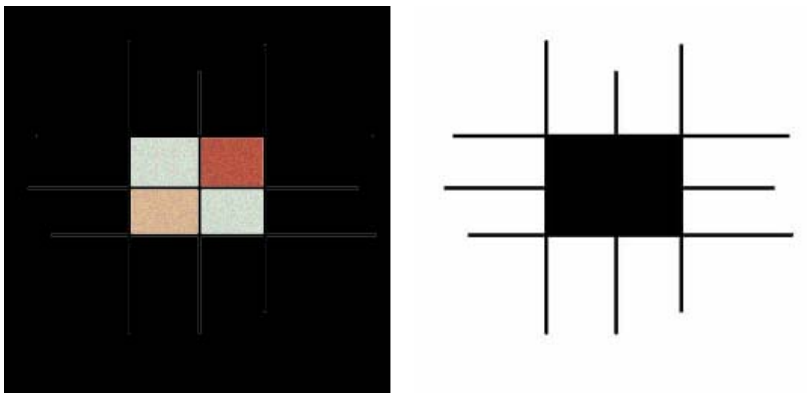


Figura 37. A l'esquerra un *bitmap* que conté una aura ja colorejada, a la dreta la màscara que utilitzarem per aconseguir la transparència adequada.

Cada tipus d'objecte tindrà una aura diferent i si estem utilitzant la primera de les possibilitats de representació el color de l'aura canviarà lleugerament en funció de la cara de l'objecte que estigui en contacte amb la taula.

La posició on es dibuixarà l'aura i l'orientació d'aquesta són dades que l'aplicació rebrà del mòdul de gestió. Aquest mòdul ens avisarà quan un objecte sigui activat i ens enviarà la posició i l'orientació del objecte en qüestió cada cop que una d'aquestes dues canviïn. Per a fer-ho emprarà els missatges que hem comentat en l'apartat anterior on analitzàvem el domini de l'aplicació. En canvi, la mida de l'aura ens la donarà el mòdul de síntesi de so que ens enviarà a cada instant la mida que ha de tenir l'aura per tal que el batec d'aquesta es correspongui d'alguna manera amb l'estat intern del objecte.

6.2.3 Mostrar les connexions entre els objectes

En el prototipus actual de la reacTable* podem trobar dos tipus de connexions entre objectes. Tenim per una banda les connexions d'àudio entre objectes generadors o manipuladors d'àudio i per l'altre les connexions de control entre un objecte de control i un objecte de tipus àudio. A l'especificació de requeriments diu clarament que s'ha de diferenciar visualment quin tipus de connexió estem representant, si una d'àudio o una de control. Així doncs, tractarem cada una d'una manera diferent.

Per a les connexions d'àudio optarem per una opció clàssica, però efectiva i coherent. El que farem serà simplement dibuixar la forma d'ona del so que va d'un objecte a l'altre. Es tracta d'una solució simple però que aporta tot el feedback desitjat, ja que per mitja de la forma d'ona l'usuari independentment de si te nocions d'acústica com si no es dona compte inconscientment de com afecten les seves accions a l'objecte. La unió d'aquest feedback amb el feedback sonor crea en l'usuari una sensació major de domini del sistema.

Aquí caldrà tenir en compte també el fet que els enllaços entre objectes poden tornar-se forts com s'ha dit en un altre passatge de la memòria. Per tant haurem de diferenciar també d'alguna manera entre enllaços normals i enllaços forts dins de les connexions d'àudio.

Per representar la forma d'ona dels objectes d'àudio s'emprarà directament una primitiva d'OpenGL, concretament la GL_LINE_STRIP, que va creant segments unint els vèrtexs que se li van passant un darrera l'altre. Això ens permetrà fàcilment projectar la forma d'ona del so simplement passant-li les mostres una darrera l'altra a l'OpenGL. Per tal d'evitar *aliasing*, utilitzarem les propietats d'*antialiasing* ofertes per OpenGL, és a dir, activarem el GL_LINE_SMOOTH.

Per diferenciar clarament entre els enllaços normals i els forts, el que farem és optar per la solució més intuïtiva. Aquesta solució és simplement canviar el gruix de les línies que dibuixaran la forma d'ona. Així que els enllaços normals seran més prims i els enllaços forts seran més gruixats, però ho haurem de cercar un equilibri raonable, ni els normals tenen que ser massa prims ni els altres massa gruixats, però la diferencia s'ha de veure a simple vista.

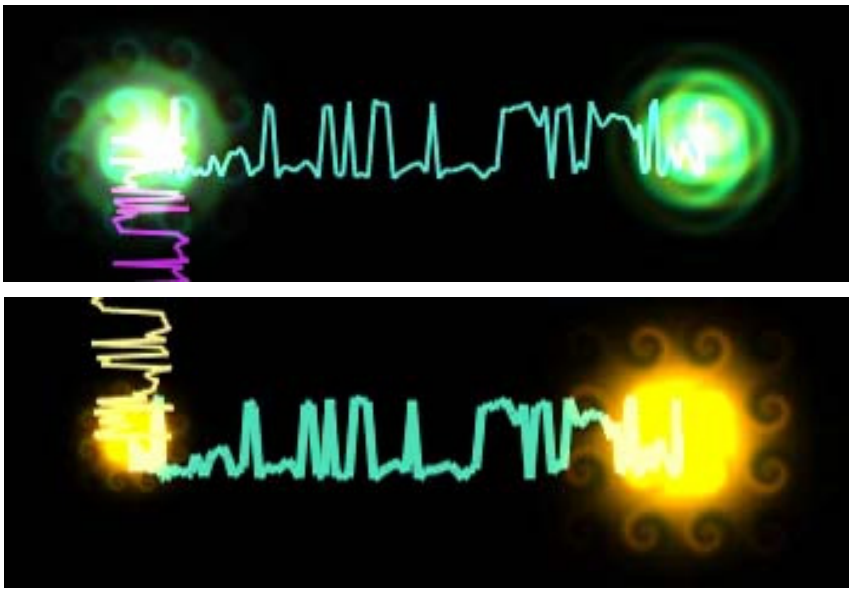


Figura 38. A dalt exemple d'enllaç normal, a sota un exemple d'enllaç fort.

El color en que es dibuixi la forma d'ona podrà ser aleatori o bé pot ser el mateix per tots els objectes. Així doncs, aquesta serà una de les opcions que tindrem a l'hora de crear un nou tema visual.

Les mostres de l'ona a dibuixar es rebran del mòdul sintetitzador de so. Així doncs, l'aplicació haurà d'escoltar permanentment al sintetitzador de so, que li anirà enviant les mostres d'àudio de cada un dels objectes d'àudio que hi hagi activats sobre la taula. El programa rebrà una llista de mostres que haurà d'interpretar i convertir en vèrtexs que passarà a l'OpenGL. Els enllaços entre objectes es crearan només quan el component central de gestió enviï un missatge a l'aplicació dient que s'ha produït una connexió entre dos objectes, serà també el component de gestió qui ens digui si un enllaç ha esdevingut fort o no.

Les connexions de tipus control seran tractades d'una manera bastant diferent. Al principi no estava gens clar com es volien representar aquests enllaços entre objectes de control i objectes d'àudio. Finalment, es va decidir que aquestes connexions serien representades mitjançant un fluix de partícules que sortirien de l'objecte de control i anirien a parar a l'objecte àudio.

Per construir aquest fluix s'emprarà una tècnica molt estesa avui en dia en el mon dels gràfics per ordinador, sobretot en el camp dels videojocs, la intenció és fer servir un petit sistema de partícules sistema de partícules. Els sistemes de partícules s'utilitzen normalment per modelar objectes que no tenen una superfície definida i que no són objectes rígids, és a dir objectes dinàmics, fluids. Els objectes més comuns que es solen modelar amb aquesta tècnica són el foc, el fum o l'aigua. El seu principi de funcionament és bastant senzill, és tracta de que l'objecte a modelar està format per moltes entitats més petites, les partícules on cada una d'aquestes té unes propietats concretes i actua en funció d'aquestes, és a dir cada una d'aquestes partícules té un comportament propi dins d'uns límits. Segons quin objecte es vulgui modelar les propietats de les partícules variaran. Cada partícula s'acostuma a representar mitjançant un quadrat sempre orientat cap a la càmera amb una textura determinada, que sol anar en funció de l'objecte a simular i pot ser que per un mateix objecte s'utilitzin varies textures diferents.

En el nostre cas no ens haurem de preocupar d'orientar el quadrat cap a la càmera, ja que treballem en dues dimensions. Referent a la textura emprarem la mateixa per totes les partícules, podran veure les textures utilitzades en les figures que vindran a continuació.

L'ideal seria que la quantitat de partícules del fluix i certes propietats com poden ser la velocitat de les partícules, el seu color o la seva mida estés directament relacionada amb el tipus de senyal que envia l'objecte de control o la freqüència en que ho fa, però no sembla que això es pugui abordar dins el termini del projecte, així que serà una de les possibles ampliacions. Així doncs, la mida i el numero de les partícules serà fix i el seu color i velocitat seran aleatoris.

Com hem dit que faríem amb les aures, amb els enllaços de control també donarem dues possibilitats de representació. La primera possibilitat, igual que amb les aures, serà la d'utilitzar com a partícula un *bitmap* en escala de grisos i quan el representem li assignarem un color i utilitzarem aprofitant les funcions de *blending* del OpenGL.

Figura 39. Bitmap que utilitzarem per al primer tipus de partícules.

L'altre opció disponible de representació no serà com amb les aures on s'oferirà la possibilitat de textures colorejades amb l'ús de màscara per a les transparències, sinó que serà una mescla d'això i de l'anterior solució. El que farem serà utilitzar textures parcialment colorejades amb màscara però també les podrem colorejar aprofitant les funcions del OpenGL. Per si no ha quedat massa clar, la següent figura il·lustrarà el procediment, mostrant la textura semi colorejada, la seva màscara i després una imatge del que seria el quadrat amb les textures aferrades i la coloració com es pretén que quedi en l'aplicació final.

Figura 40. A l'esquerra la textura base, al mig la màscara, a la dreta la partícula com és veurà en el programa.

A més d'aquestes dues possibilitats per a variacions del tema visual, també hi haurà la possibilitat de canviar el número de partícules que s'empraran per a generar el fluix que representarà l'enllaç de control.

6.2.4 Les imatges han de ser visualment atractives

Aquest és un requeriment molt subjectiu i que potser es podria complir simplement creant l'aplicació de la manera que hem dit fins ara, és a dir amb la representació de les aures i els enllaços utilitzant unes textures boniques l'aplicació podria ja generar imatges, des del meu punt de vista, d'una bellesa considerable. Però per no abusar del conformisme, he decidit utilitzar un efecte gràfic que pot millorar considerablement l'impacte visual. Aquest efecte és conegut amb el nom de *radial blur*, és a dir desenfocament o difuminat radial.

El *radial blur* consisteix en difuminar cada píxel de la imatge real en la direcció oposada al centre del difuminat. El problema és que amb el hardware actual és bastant costós realitzar aquesta operació en una resolució digne i a temps real i només els programes d'edició d'imatge el realitzen d'aquesta manera. Així doncs, el que es fa per les aplicacions a temps real és una falsificació d'aquest difuminat radial, però que queda bastant convincent i realment curiós i estèticament bonic. Podem trobar un exemple d'aquest efecte a la pàgina web de Neon Helium Productions [29].

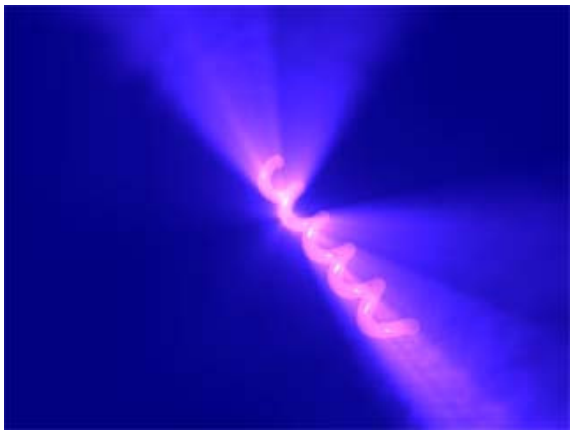


Figura 41. Exemple de *radial blur* de NeHe que intentarem aplicar en el programa.

La meva intenció és aplicar aquest efecte als enllaços sonors, que potser fins ara són els elements gràfics menys agraciats, estèticament parlant, del conjunt de l'aplicació.

El funcionament d'aquest efecte és el següent: primer dibuixem tot el que volem que es vegi afectat per l'efecte en una textura més petita que la que serà la resolució final, després pintem tota l'escena i finalment aferrem la textura petita a sobre de tota la finestra de render, així gràcies als filtres de textures d'OpenGL aconseguiríem un efecte semblant al desenfocament gaussià, i si en lloc d'aferrar una sola textura anem aferrant varies de manera radial respecte al centre de la finestra aconseguim un *radial blur* amb el centre de la finestra com a centre de desenfocament. En funció del desplaçament que hi hagi entre les textures aferrades i del número d'aquestes l'efecte serà més bonic o menys.

Segons el número de textures que s'emprin per fer l'efecte, segons la mida d'aquestes i segons la mida de la finestra de render el rendiment de l'aplicació pot baixar de manera dràstica, fent baixar molt el número de *frames* per segon possibles. Així que haurem d'anar molt en compte a l'hora de tractar aquests aspectes.

Primer desenvoluparem aquest efecte totalment en OpenGL 1.2, és a dir a l'estil proposat en l'exemple que hem comentat. Després intentarem utilitzar una extensió d'OpenGL, provarem els anomenats *Pixel Buffers*, pel render previ, i finalment, també provarem de crear aquest efecte mitjançant el llenguatge Cg.

6.2.5 Les imatges han de mostrar un cert dinamisme

Quan em van encarregar aquest projecte una de les primeres coses que em van dir va ser que les imatges generades havien de transmetre sensació de dinamisme, de flexibilitat, és a dir que no es tractés d'una visualització massa robòtica o estàtica. La millor manera de complir aquest requeriment varem pensar que seria donar una certa flexibilitat als enllaços a l'estil del *DynaDraw* de Paul Haelberli. És a dir, transmetre a l'usuari la impressió que els enllaços estan formats per elements flexibles sense necessàriament entrar en simulacions físiques d'elasticitat, només es pretén crear aquesta sensació.

Per donar als enllaços aquesta flexibilitat sense entrar en simulacions físiques, la millor solució que se'm va ocorre fou la utilització de corbes de Bézier. Aquestes són corbes cúbiques formulades als anys 60 per un enginyer de la Renault anomenat Pierre Bézier i que es basen en els polinomis de Hermite i actualment són molt utilitzades en sistemes de disseny assistit per ordinador.

Les corbes de Bézier és controlen mitjançant quatre punts, dos dels quals pertanyen a la corba i els altres dos no. Els dos punts que pertanyen a la corba defineixen en quin punt comença la corba i en quin punt acaba, és a dir són el punt d'inici i el punt final de la corba, mentre que els dos punts que no són de la corba defineixen la seva curvatura. A la següent figura en veurem alguns exemples.

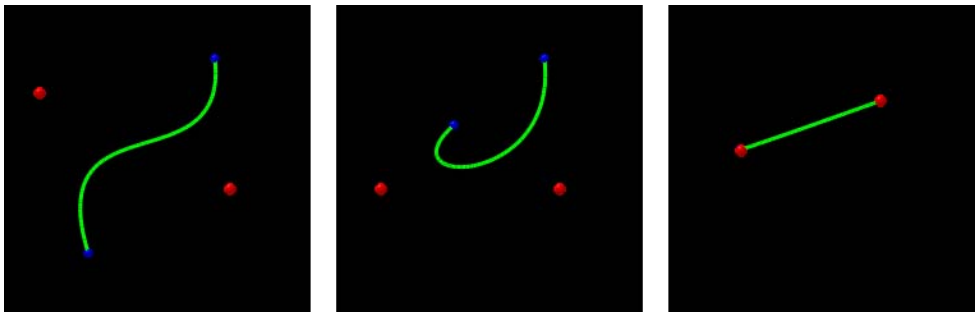


Figura 42. Diferents corbes de Bézier, els punts vermells són els punts de control i, evidentment, els punts blaus són el de inici i de final.

La intenció és fer que tant els enllaços de les connexions d'àudio com de les connexions de control segueixin corbes d'aquest tipus, on els punts de control seran gestionats de tal manera que els enllaços semblin ser flexibles.

Aquest efecte s'ajusta als requeriments de l'aplicació i a més donarà un toc distintiu a la visualització creada.

6.2.6 S'han de mostrar 30 imatges per segon

Per tal d'assolir el requeriment, que ens obliga a que el *frame-rate* de l'aplicació sigui de 30 imatges per segon, haurem de fer dues coses. La primera serà la de realitzar tot el que hem dit fins ara en aquest apartat de la manera més òptima que puguem per tal que el nostre hardware gràfic pugui realitzar totes les operacions que volem fer de manera eficient i el *frame-rate* de l'aplicació sigui el més superior possible a 30, és a dir, com més gran sigui el *frame-rate* millor.

La segona cosa que haurem de fer serà limitar l'aplicació de manera que no sobrepassi aquestes 30 imatges per segon. La llibreria SDL té certes funcions que ens ajudaran a fixar aquesta limitació.

Així, per complir aquest requeriment el que farem serà complir els requeriments gràfics de la manera més òptima possible i limitar a 30, mitjançant la llibreria SDL, el número de vegades per segon que es cridi la funció de render.

6.2.7 Diferents temes visuals

Aconseguir que l'aplicació compleixi un requeriment com és aquest de permetre diferents temes visuals pot no ser gens fàcil si l'aplicació no ha estat pensada per a fer-ho des d'un principi. En aquest cas ho sabem des del principi. El que es farà per assolir aquest requeriment serà tractar tots els elements gràfics amb la màxima flexibilitat possible, en el sentit de poder canviar fàcilment les textures de les aures o de les partícules, poder canviar el tipus de textura, poder canviar els colors o poder decidir si es volen utilitzar efectes o no. Tot això s'ha tingut en compte, com heu pogut llegir, en els apartats anteriors.

Per gestionar els temes visuals de manera adequada es crearà un nou i simple format de fitxer en el que es definirà el tema visual. L'aplicació haurà de ser capaç d'obrir aquest tipus de fitxer i interpretar-los de manera que mitjançant la carrega d'un fitxer o un altre la visualització utilitzi un tema visual o un altre. Aquest tipus de fitxer de temes visuals o de *skins* contindrà: el color de fons que es vulgui utilitzar, si s'utilitzarà o no l'efecte de les corbes de Bézier, si s'emprarà l'efecte de *radial blur* o no i amb quantes textures, després també contindrà els noms dels fitxers de les textures de les partícules i el número de partícules que s'empraran, hi posarem també una llista amb els noms dels fitxers de les textures que emprarem per les aures i, finalment, per a cada tipus d'objecte posarem quina textura o textures i colors utilitzarem per a la seva aura i també podrem fixar el color del seu possible enllaç sonor o deixar-lo aleatori.

Per assegurar-nos d'assolir aquest requeriment i comprovar que l'aplicació és apte per a gestionar varis *skins* per diferents que siguin, el que farem en el context d'aquest projecte serà preparar dos temes visuals radicalment diferents i, així, si aconseguim que l'aplicació funcioni bé amb els dos *skins* i puguem canviar de l'un a l'altre simplement canviant el fitxer a carregar serà que hem fet la feina bé.

Per al primer tema visual ens basarem en les visualitzacions dels programes reproductors de mp3, que tant de moda s'han posat darrerament. Aquest tema farà servir els dos efectes comentats anteriorment i una estètica més acord amb els últims temps.

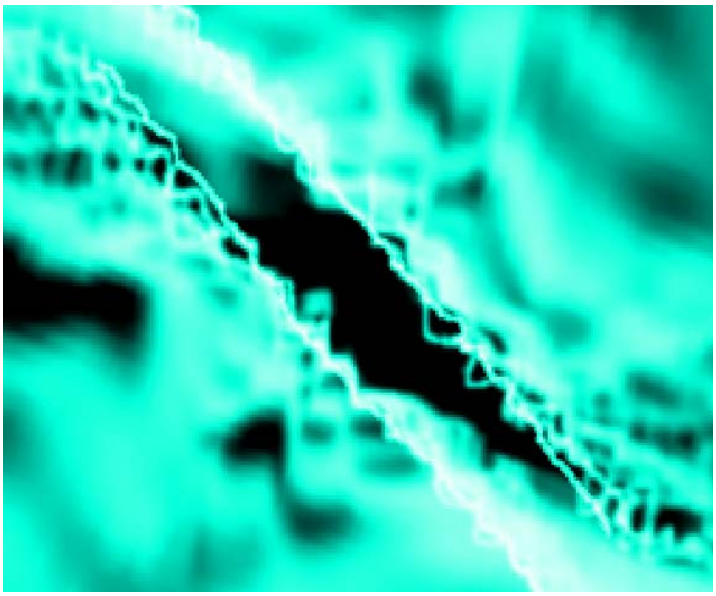


Figura 43. Visualització del reproductor multimèdia de Windows en el que ens basarem per al primer skin.

En canvi per al segon tema visual optarem per una cosa menys convencional i a la vegada més simple que l'anterior. No utilitzarem cap dels dos efectes que tenim disponibles i ens inspirarem en els quadres de Wassily Kandinsky.

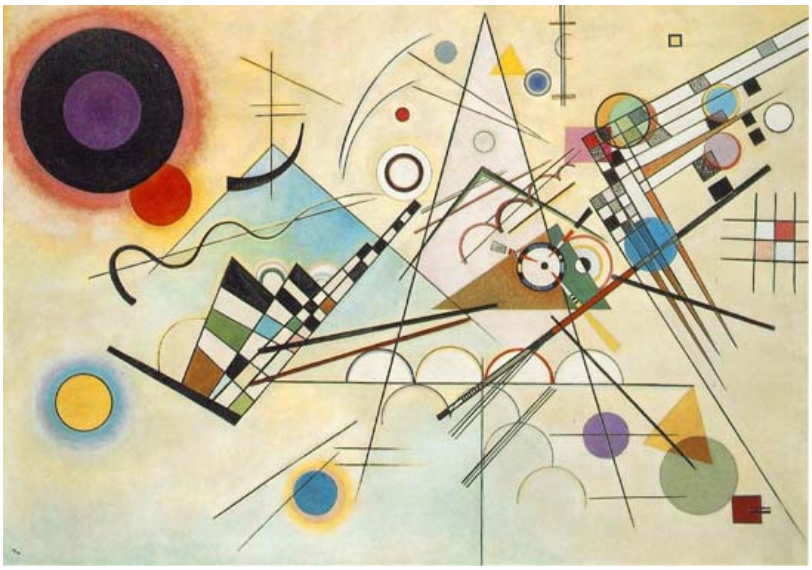


Figura 44. Quadre de Wassily Kandinsky que servirà d'inspiració.

Com es pot apreciar a simple vista, les dues fonts d'inspiració són bastant diferents, per tant si és fa bé sortiran dos *skins* prou diferents per avaluar si la gestió de temes visuals és adequada.

6.2.8 La comunicació amb els altres mòduls de la reacTable*

Com hem comentat anteriorment en l'apartat d'anàlisi del domini, la comunicació del mòdul de visualització amb els altres mòduls és imprescindible. Per tant, el compliment del requeriment referent a la comunicació amb el component central de gestió i amb el mòdul de síntesi de so és bàsic per al bon funcionament de l'aplicació.

La comunicació ja hem dit que serà unidireccional i que serà per xarxa mitjançant el protocol UDP i que els missatges que la nostra aplicació rebrà seran els que s'han descrit anteriorment. Per tant, en la nostra aplicació hi haurà d'haver dos *threads* que actuïn com a servidors, mitjançant l'utilització de *sockets* UDP, i que s'encarreguin, un d'escoltar els missatges del component de gestió i l'altre els missatges provinents del mòdul encarregat de la síntesi de so.

Però no ens bastarà només amb rebre els missatges, haurem de buscar una manera de guardar i interpretar degudament aquestes dades. Aprofitant els avantatges de la programació orientada a objectes haurem d'organitzar l'aplicació de manera que tinguem una estructura semblant a la que hi ha físicament sobre la taula per poder interpretar fàcilment els missatges rebuts.

Si aconseguim dissenyar i implementar adequadament tots els punts que s'han exposat i discutit en aquest apartat, es compliran tots els requeriments del mòdul de visualització i per tant s'hauran assolit els objectius d'aquest projecte.

6.3 Disseny i implementació

En aquest apartat s'explicarà la solució informàtica que proposa aquest projecte per tal d'assolir els requeriments proposats. Primer veurem com s'organitzarà el codi, quines classes es crearan i com es relacionaran entre elles, així com els patrons de disseny de software que s'utilitzaran. Després es descriurà, sense entrar massa en detall, cada una de les classes i finalment veurem per separat com s'han solucionat alguns dels aspectes més interessants.

6.3.1 Com s'organitzarà l'aplicació

A l'hora de dissenyar l'aplicació s'ha intentat aprofitar el fet d'estar programant en un llenguatge orientat a objectes com és C++. Així que s'ha desglossat l'aplicació en diferents classes i s'ha intentat aconseguir un nivell d'abstracció adequat. Però abans d'explicar l'organització en classes de l'aplicació, direm que en qüestió d'execució l'aplicació es dividirà en tres *threads*: el *thread* principal serà el que s'encarregarà de pintar tots els elements gràfics, un altre *thread* escoltarà els missatges provinents del mòdul central de gestió i un tercer *thread* rebrà els missatges del mòdul de síntesi de so. Aquests dos últims *threads* no només hauran d'escoltar els missatges sinó que els hauran d'interpretar i s'hauran d'alterar les propietats dels objectes que dibuixa el *thread* principal en funció d'aquests.

D'altra banda, pel que fa a l'organització en classes de l'aplicació podem dir que és sabut que el funcionament de la reacTable* està basat en la manipulació d'una sèrie d'objectes per sobre d'una taula i que la missió principal d'aquesta aplicació és projectar sobre aquesta taula aures per aquests objectes i connexions entre aquests objectes. Hi ha objectes de diferents tipus, però a hores d'ara i pel que fa a aquest projecte, podem dividir els objectes en dos tipus bàsics: objectes d'àudio i objectes de control. Així doncs, i donat que per a cada objecte haurem de pintar la seva aura i el seu enllaç sonor si escau, s'ha decidit crear una classe que representi un objecte de la reacTable* i que aquesta contengui els mitjans per a dibuixar la seva aura i el seu enllaç sonor. Aquesta és la classe `ReacObject` i serà la base de la nostra aplicació.

Un objecte haurà de tenir els mitjans per dibuixar la seva aura. Per això crearem una classe aura que sigui capaç de dibuixar i gestionar una aura tal com hem dit que la volíem en l'apartat anterior. Per tant, estarà formada per varies capes, on cada capa serà, per dir-ho de forma senzilla, un quadrat amb una textura colorejada.

Com s'ha dit, podem dividir els objectes de la reacTable* en objectes d'àudio i objectes de control. La diferència entre aquests dos conjunts és bàsicament el tipus de connexió que tenen amb els altres objectes, per tant el que farem serà definir una classe per cada un d'aquest tipus d'objectes i fer que ambdues classes heretin de la classe `ReacObject` i que cada una s'especialitzi en la gestió del seu tipus d'enllaç.

Tenim doncs dos tipus d'enllaç entre objectes, un és l'enllaç sonor i l'altre l'enllaç de control. Aplicarem la mateixa tècnica que s'ha emprat amb els objectes i crearem una classe que faci referència als enllaços en general, la classe `Link`, i d'aquesta en derivarem dues, una representarà els enllaços sonors i l'altre els de control. Aquestes classes contindran el necessari per a pintar els enllaços adequadament, els sonors mostraran la forma d'ona i els altres es serviran d'un sistema de partícules per pintar un fluix entre dos objectes.

Per aconseguir el fluix de partícules dels enllaços de control, crearem una classe que sigui l'abstracció d'un sistema de partícules i que pugui pintar el fluix tal com l'hem definit anteriorment. Així si fem que els enllaços de control heretin també d'aquesta classe, aquests seran capaços de pintar el fluix de control.

En canvi, per pintar els enllaços d'àudio hem dit que s'utilitzaria la primitiva GL_LINE_STRIP d'OpenGL per pintar les mostres d'àudio que passen d'un objecte a l'altre, però també hem dit que se li podrien aplicar corbes de Bézier. Per aconseguir això construirem una classe que ens permeti fer que els nostres enllaços sonors segueixin corbes de Bézier. També utilitzarem aquesta classe amb els enllaços de control.

Arribats a aquest punt, tenim una classe `ReacObject` que és "capaç" de dibuixar tot el que volem dibuixar d'un objecte, però no tenim cap mitjà per saber on hem de pintar aquest objecte, quina aura haurà de dibuixar o quines mostres ha de dibuixar. Tota aquesta informació ve des de fora de l'aplicació, concretament, del mòdul de gestió i del de síntesi de so. Per tant, haurem de crear una classe que ens permeti aquesta comunicació que com sabem anirà sota UDP, serà la classe `UDPServer`. Ara tenim una classe que reb els missatges però ens en falta una que els interpreti i posi en pràctica les ordres que hi ha en elles, aquesta serà la classe `ObjectContainer`, una de les classes claus en l'aplicació.

També utilitzarem un conjunt de classes per a fer la feina més senzilla que són molt utilitzades en el camp dels gràfics per ordinador. Ens referim a classes bàsiques com poden ser una classe per treballar amb punts, una per treballar amb colors i una altra que ens faciliti la càrrega de textures.

Per a poder observar d'una manera més clara l'organització en classes de l'aplicació a la pàgina següent els oferim un diagrama de classes on potser es veurà tot més clar. Per tal de simplificar un poc el diagrama, han quedat excloses del dibuix les classes referents als punts, als colors i a les textures. Així com la classe `UDPServer` que no té cap relació directa amb cap de les classes existents.

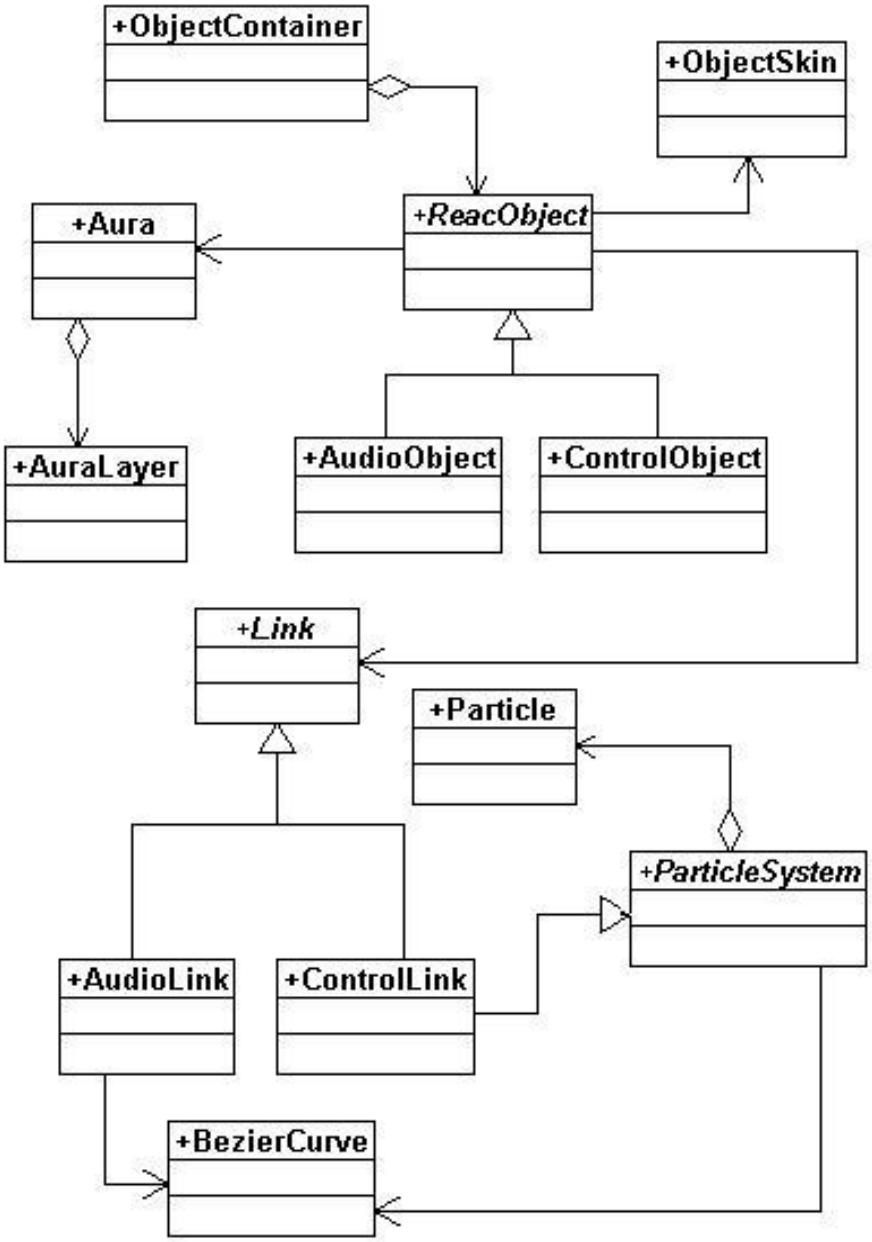
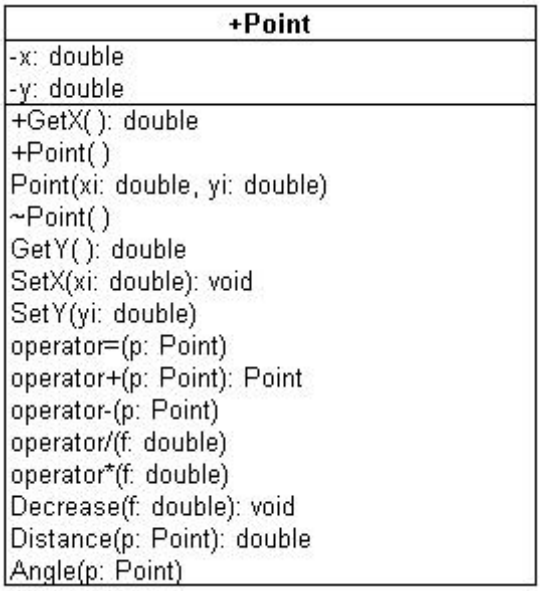


Figura 45. Diagrama de classes de l'aplicació.

6.3.2 Descripció de les classes

En aquest subapartat estudiarem una per una totes les classes que conformen l'aplicació construïda en aquest projecte. Direm la funció que té cada una d'aquestes classes dins el sistema i quines són les seves característiques més importants.

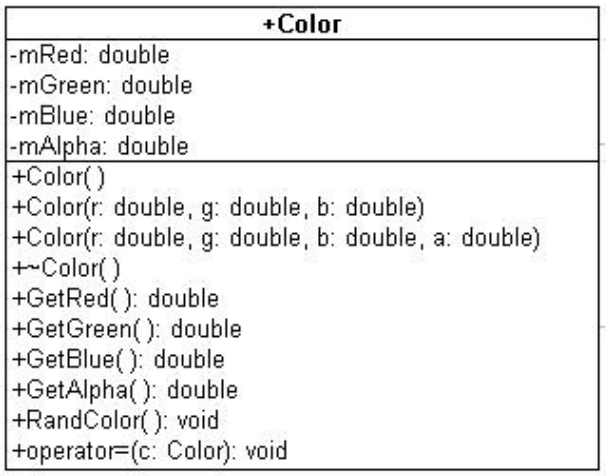
Point



Aquesta classe és l'abstracció d'un punt en dues dimensions. Així que com a variables té la x i la y i com a mètodes té les operacions més comunes que es poden aplicar a un punt, és a dir assignació, suma, resta, producte o divisió. També s'han afegit altres mètodes que permeten obtenir la distància d'un punt respecte a un altre o l'angle que forma un punt respecte a un altre punt i a l'eix d'abscisses.

Utilitzarem molt aquesta classe, per exemple l'emprarem per guardar les posicions dels objectes, els punts que formen una corba de Bézier o les posicions de les partícules. Cal comentar que també hi ha un mètode poc comú, que és el de decrement, que ens ajudarà sobretot per a la gestió dels punts de control de les corbes de Bézier, quan els utilitzem per donar flexibilitat als enllaços.

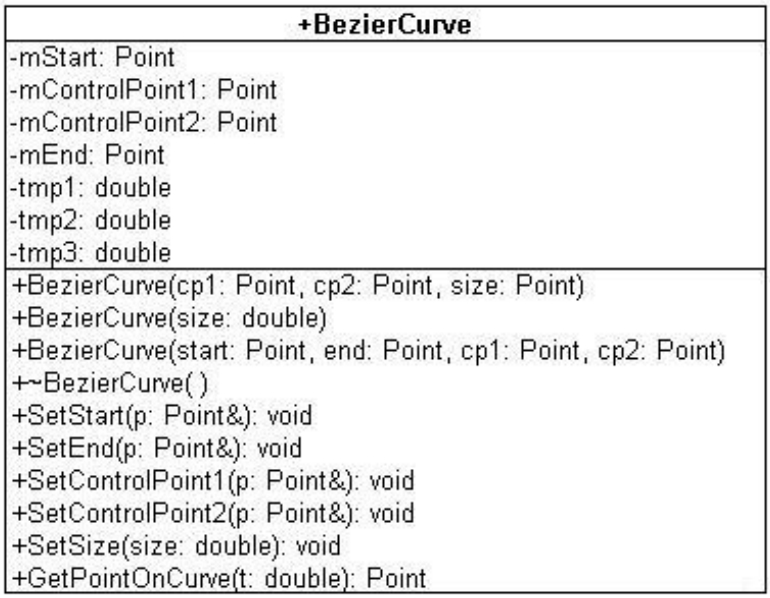
Color



La classe `Color` és la classe que emprarem per emmagatzemar tots els colors que utilitzarem a l'aplicació. Hi guardarem el color en quatre components, la quantitat de vermell, de verd, de blau i la component *alpha* per a la transparència.

Només tindrem dos mètodes, un per l'assignació i un altre per a generar un color aleatori, que és una funció que s'utilitzarà en alguns casos a l'aplicació.

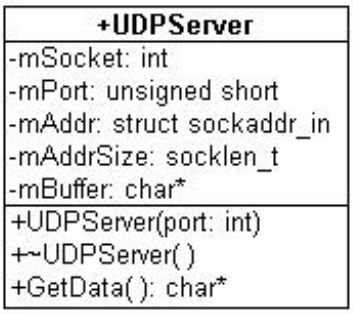
BezierCurve



És una classe que implementa, evidentment, el que seria una corba de Bézier. Està formada bàsicament per quatre punts: el de inici i el de final de la corba i els dos punts de control. Té un mètode, anomenat `GetPointOnCurve`, que donat un instant t ens retorna el punt on estarà la corba en aquell instant.

Aquesta és la classe que utilitzarem per fer que tant els enllaços sonors com els de control segueixin corbes de Bézier i puguin semblar flexibles d'alguna manera.

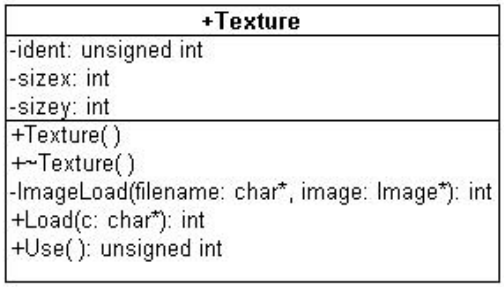
UDPServer



Aquesta classe encapsula tot el que necessitem per a crear un *socket* servidor UDP i per anar-hi rebent missatges. Basta que li diguem el port per el que s'han de realitzar les comunicacions a l'hora d'instanciar-lo i que anem llegint el que reb mitjançant el mètode `GetData`, que ens retorna una cadena de caràcters amb tot el que ha rebut el *socket*.

Emprarem aquesta classe per establir les comunicacions amb el component central de gestió i amb el mòdul de síntesi de so.

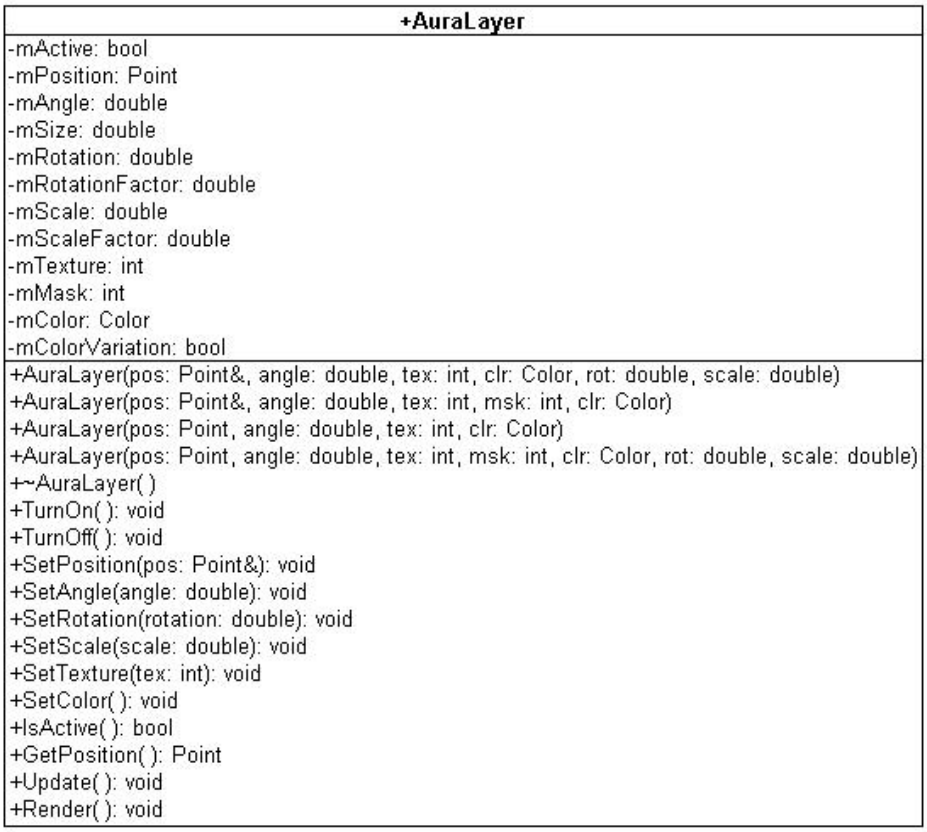
Texture



És la classe encarregada de gestionar les textures. El que fa exactament és obrir mapes de bits, i guardar-los en memòria en forma de textura llesta per utilitzar en OpenGL, de manera que quan vulguem utilitzar aquesta textura simplement haguem de cridar al mètode `Use` perquè ens retorni l'identificador amb que OpenGL coneix la textura guardada.

Aquesta classe serà molt emprada, ja que s'utilitzarà per a carregar totes les imatges que apareixen en l'aplicació.

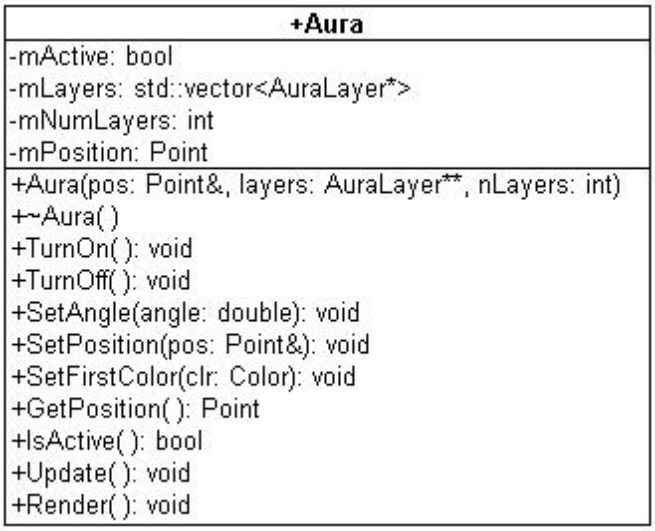
AuraLayer



Aquesta classe defineix una de les diferents capes que pot tenir una aura d'un dels objectes de la reacTable*. És a dir, és bàsicament una de les textures que poden formar una aura. Però no és només una textura. Cada una de les capes d'una aura te una posició, un angle, un color i una mida propia, a part de que pot estar activada o no.

Els mètodes més interessants d'aquesta classe són el `Update` i el `Render`. El `Update` actualitza els valors de la mida, per al batec de l'aura, i de la rotació si és necessari. Mentre que el `Render` el que fa és pintar la capa a la posició desitjada, amb l'orientació i la mida adequades. La capa és un quadrat amb una textura aferrada, o dos quadrats amb dues textures en el cas en que s'utilitzi una màscara, amb la funció de *blending* adequada per a cada cas.

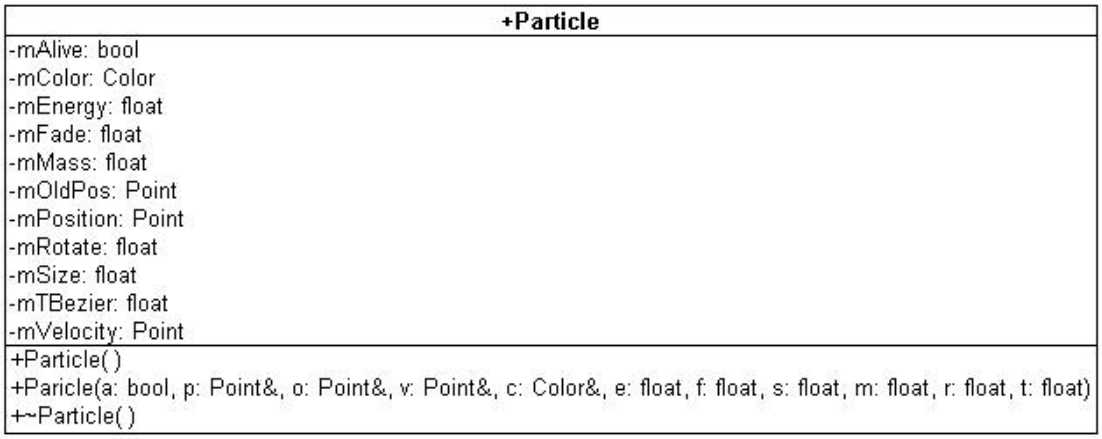
Aura



La classe `Aura` és bàsicament un vector de capes de aura, és a dir de `AuraLayer`. És l'abstracció directe de les aures que volem que surtin baix els objectes de la taula. Així doncs, la seva funció serà la de gestionar i dibuixar les aures dels objectes.

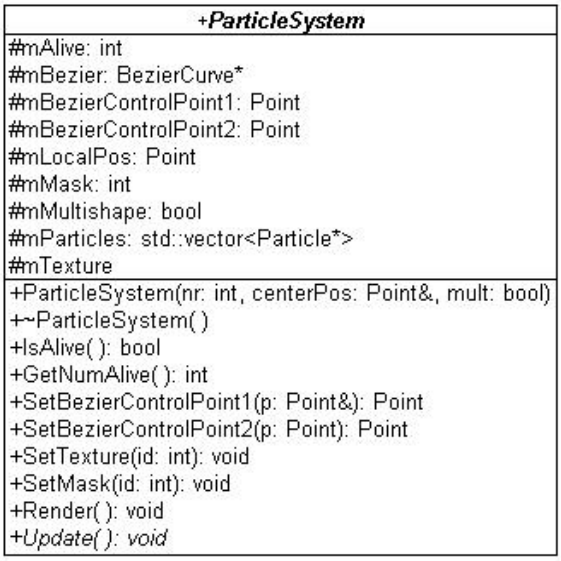
Aquesta classe conté un vector amb una o més `AuraLayer`, la posició on s'ha de dibuixar l'aura, és a dir la posició de l'objecte, un booleà que ens indica si l'aura està activa o no i un enter que ens diu per quantes capes està formada aquesta aura. El seu mètode de `Render` simplement recorre el vector de `AuraLayer` i crida el mètode `Render` de cada una d'aquestes i el mateix fa el mètode `Update` o el mètode `SetPosition`.

Particle



És una classe molt simple que només conté les propietats de les partícules. La utilitzarem única i exclusivament per al sistema de partícules, on emmagatzemarà les dades necessàries sobre cada una de les partícules del sistema.

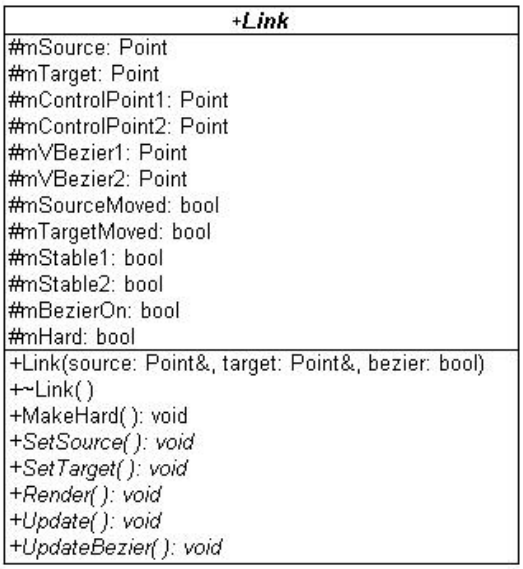
ParticleSystem



Aquesta és una classe abstracte que defineix com s'ha de pintar un sistema de partícules i conté un vector de partícules, un o dos enters, segons si utilitzarem màscara o no, que identifiquen les textures que s'utilitzaran per a dibuixar-les, així com un punter a una `BezierCurve` que ens ajudarà a que les partícules segueixin una corba de Bézier.

La seva funció de `Render` recorre el vector de partícules i les dibuixa com una textura aferrada sobre un quadrat, creat amb la primitiva GL_TRIANGLE_STRIP per a una millor eficiència, o dues textures en el cas en que utilitzem una màscara sempre utilitzant les funcions de *blending* de OpenGL adequades.

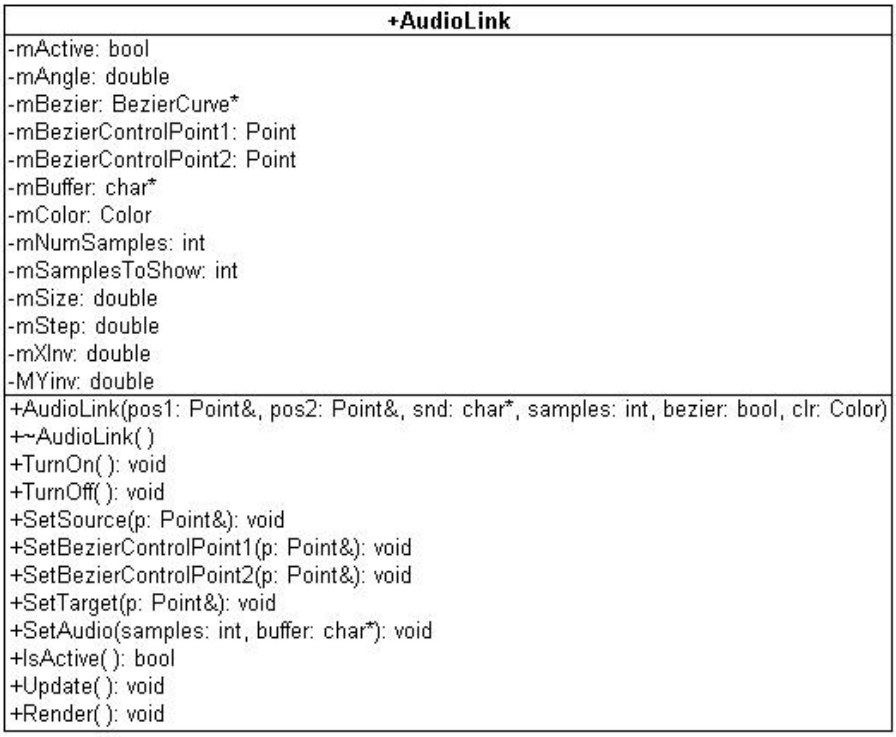
Link



Es tracta d'una classe abstracta que fa referència al concepte d'enllaç amb l'objectiu de representar les connexions entre objectes.

Aquesta classe és molt genèrica i serveix sobretot per fixar els punts en comú entre els enllaços d'àudio i els de control, com poden ser les variables membres destinades a la posició d'inici i de final, és a dir la posició de l'objecte emissor i la posició de l'objecte receptor respectivament, les variables encarregades de gestionar les corbes de Bézier, o la variable que ens indica si l'enllaç és fort o no.

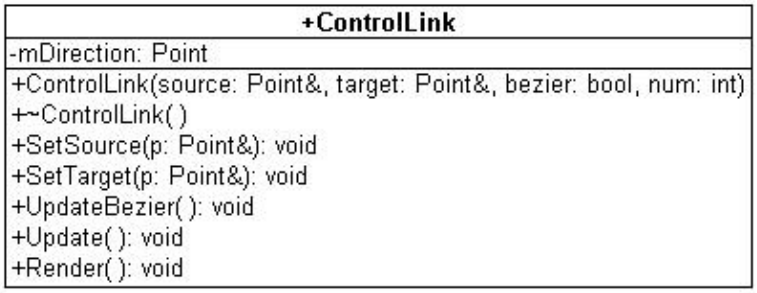
AudioLink



És una classe que hereta de la classe anterior, és a dir de la classe `Link`, i és, com es pot deduir del seu nom, la classe encarregada de l'abstracció dels enllaços sonors. La seva funció és bàsicament pintar entre l'objecte del que surt l'enllaç i el seu destinatari les mostres que envia aquest objecte emissor del so. Per fer-ho consta de totes les variables que hereta de la classe `Link` i variables pròpies com el color en que es dibuixarà, un apuntador a les mostres que es dibuixaran o una corba de Bézier.

El seu mètode `Update` s'encarrega d'actualitzar la corba de Bézier fent que els punts de control segueixin als punts d'inici i final de l'enllaç amb un petit retard i també s'encarrega d'actualitzar les mostres que s'han de representar. La funció `Render` el que fa és pintar una GL_LINE_STRIP d'un color i una gruixa concrets i que va de l'objecte d'origen a l'objecte de destí però que segueix un camí alterat per la corba de Bézier i per les mostres de la forma d'ona que ha de dibuixar.

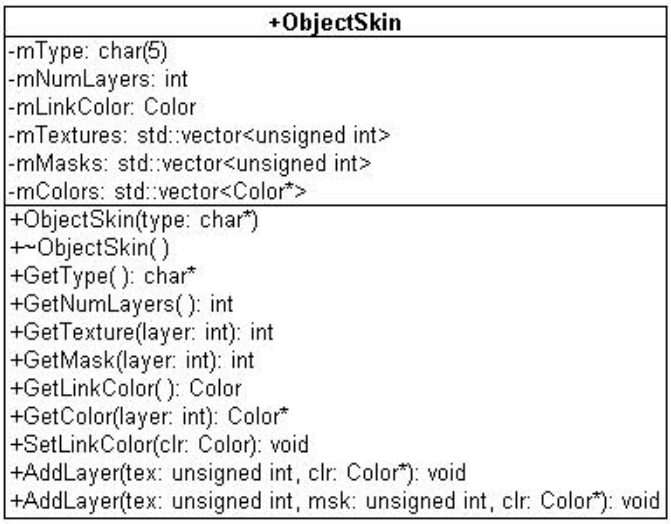
ControlLink



Aquesta classe representa els enllaços de control i deriva de la classe `Link` i de la classe `ParticleSystem`. La majoria de la funcionalitat d'aquesta classe li ve donada de les seves superclasses. El que fa, doncs, és aprofitar el que hereta de `ParticleSystem` per a dibuixar un fluix de partícules que va de l'objecte emissor de les dades de control a l'objecte receptor d'aquestes dades.

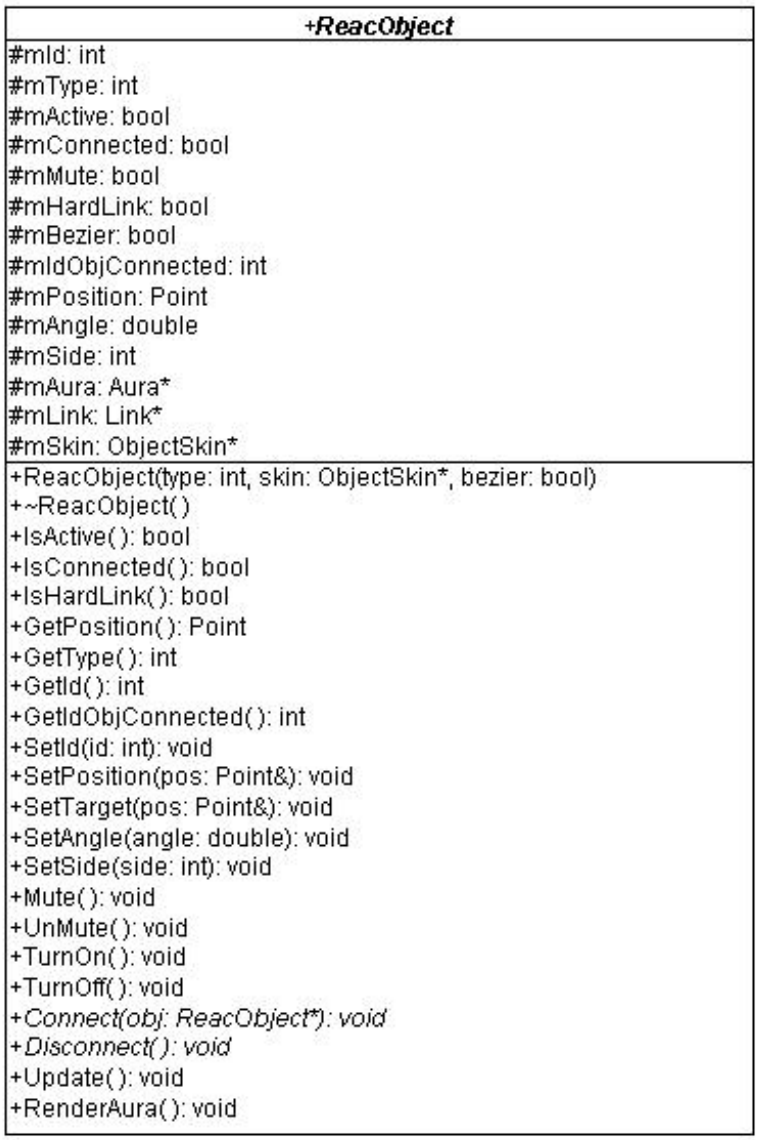
En aquesta classe trobem tant sols una funció membre interessant, la `Update`, ja que la funció `Render` s'hereta directament de `ParticleSystem`. La funció `Update` d'aquesta classe determina el comportament que tindran les partícules del fluix, mitjançant una gestió determinada de les propietats de les partícules.

ObjectSkin



És la classe que guarda i gestiona tota la informació referent al tema visual que afecta a un objecte de la taula. És a dir, conté informació sobre quines textures s'utilitzaran per a les aures de l'objecte, el numero de capes d'aquestes aures o els colors d'aquestes capes i dels enllaços que pugui tenir. S'instanciarà un `ObjectSkin` per a cada tipus d'objecte diferent, no pas per a cada objecte.

ReacObject

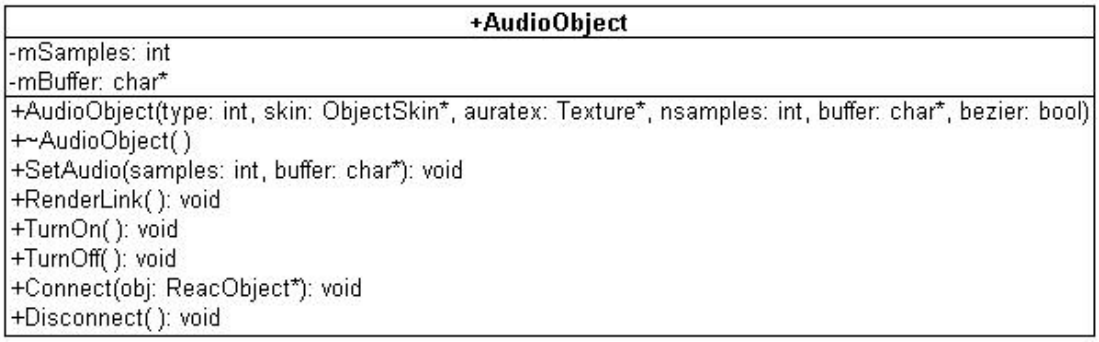


Aquesta és la classe abstracta que fa referència a un objecte físic de la reacTable*. Així doncs, aquesta classe conté l'identificador de l'objecte, el tipus d'objecte que és, informació sobre si està activat o no, si està silenciat o no, si està connectat o no a un altre i si la connexió és forta o no. També té informació relativa a la posició i orientació de l'objecte sobre la taula, així com sobre la cara que està en contacte amb la taula.

La classe en qüestió conté un apuntador a `Aura`, un apuntador a `Link` i un apuntador a `ObjectSkin`. Així doncs, apunta a l'aura que es dibuixarà a sota de l'objecte al que representa i apunta a l'enllaç que es dibuixarà en cas que aquest objecte es connecti amb un altre i també apunta a la classe que defineix com afecta el tema visual a aquest tipus concret d'objectes.

Els mètodes més interessants d'aquesta classe, tot i que són bastant simples, són `Update`, `RenderAura` i `RenderLink`. El mètode `Update` el que fa és, si l'objecte està actiu, cridar el mètode `Update` de l'aura a la que apunta i si l'objecte està connectat crida també el mètode `Update` del `Link`. I com és de suposar el mètode `RenderAura` crida al mètode `Render` de l'aura si l'objecte esta actiu, i el mètode `RenderLink` crida al mètode Render del `Link` si l'objecte està actiu, connectat i no està silenciat.

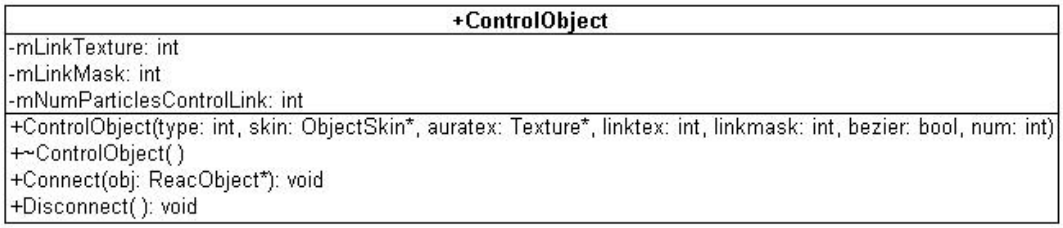
AudioObject



La classe que ens disposem a descriure és una subclasse de la classe anterior, la `ReacObject`. La classe `AudioObject`, com es pot deduir, serà la classe que s'encarregarà de dibuixar l'aura i l'enllaç sonor dels objectes d'àudio i guardar les dades que s'han de tenir sobre l'objecte. Per fer-ho es servirà de les funcions que hereta de la classe `ReacObject`.

El que canvia, a hores d'ara, d'un objecte d'àudio a un objecte de control és el tipus de connexió que realitzen, és a dir, l'únic que canviarà serà la manera de gestionar les connexions i el tipus d'enllaç a dibuixar. Per tant el que haurà d'implementar aquesta classe serà la forma com aquest objecte es connecta amb els altres, o es desconnecta, cosa que està implementada als mètodes `Connect` i `Disconnect` i també haurà d'implementar la gestió necessària dels enllaços sonors, o sigui, s'haurà d'encarregar d'enviar les mostres d'àudio l'`AudioLink` pertinent.

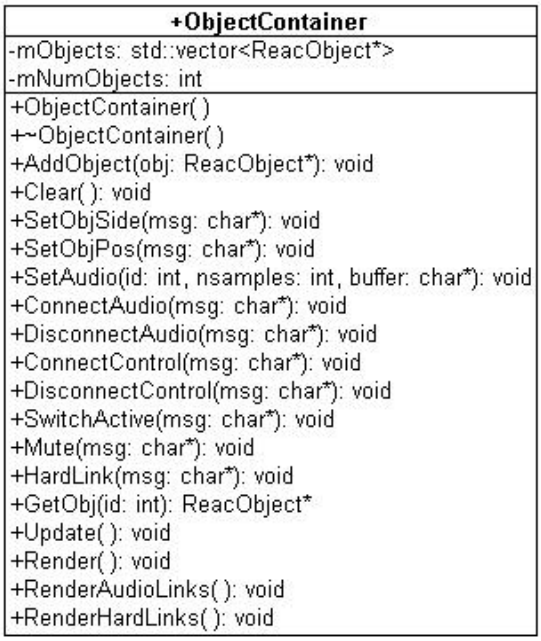
ControlObject



Com la classe anterior, aquesta classe hereta de la classe `ReacObject`, però en aquest cas s'encarrega de dibuixar l'aura i l'enllaç de control dels objectes de tipus control. Igual que la classe `AudioObject`, aquesta classe es servirà de les funcions que hereta de `ReacObject` i només afegirà petites funcionalitats degudes a les diferencies de connexió entre els diferents tipus d'objectes.

Així doncs, aquesta classe contindrà un identificador de la textura i un identificador de la possible màscara que s'empraran per a dibuixar les partícules del fluix de l'enllaç i un enter amb el número de partícules que s'utilitzaran. També definirà els seus propis mètodes per a la connexió i desconnexió on es crearà l'enllaç de control entre el propi objecte i l'objecte al que es connecta adequadament.

ObjectContainer



Aquesta classe podríem dir que actua com a intermediària entre el *main* de l'aplicació i la col·lecció d'objectes. Està formada simplement per un conjunt d'objectes, més ben dit per un vector de `ReacObject`, i un conjunt de mètodes que actuen sobre aquests.

El *main* de l'aplicació tindrà una instància d'aquesta classe, a la que el *thread* que escolta el component de gestió anirà afegint objectes, ordenant la seva connexió o desconnexió o fixarà la seva posició. En definitiva, entre el *main* i aquesta classe interpretaran els missatges rebuts i es manipularan els objectes en funció d'aquests. També el *thread* que s'encarrega de gestionar la connexió amb el mòdul de síntesi de so utilitzarà aquesta classe per anar enviant a cada objecte del que surt àudio les mostres que hauran de representar-se a cada instant.

6.3.3 Patrons de disseny de software utilitzats

Com sabem, en el procés de desenvolupament d'aplicacions, és molt probable que sovint ens trobem amb problemes de disseny amb els que ja s'ha enfrontat altre gent prèviament. Per evitar perdre el temps en reinventar les solucions, el que es fa és reutilitzar solucions que han funcionat en el passat, així varen néixer els patrons de disseny. Els patrons de disseny són el conjunt format bàsicament per una col·lecció de problemes de disseny d'aplicacions molt comuns i la millor manera d'afrontar-los.

Malgrat que la utilització de patrons de disseny en la orientació a objectes és molt comuna i malgrat tots els beneficis que ens poden oferir a l'hora de dissenyar una aplicació, en aquesta aplicació no són molts els patrons emprats. Almenys, no tinc consciencia d'haver-ne emprat més que un: el *Strategy*.

El *Strategy* és un patró de comportament, que té la intenció de definir una família d'algorismes, encapsular-los i fer-los intercanviables. Aquest patró permet que l'algorisme canviï independentment del client que l'utilitza. En la següent figura es pot apreciar el diagrama de classes del patró on podem veure que es basa en la creació de subclasses de la classe que conté l'algorisme i cada una d'aquestes subclasses sobreescriu el mètode on hi ha l'algorisme en qüestió.

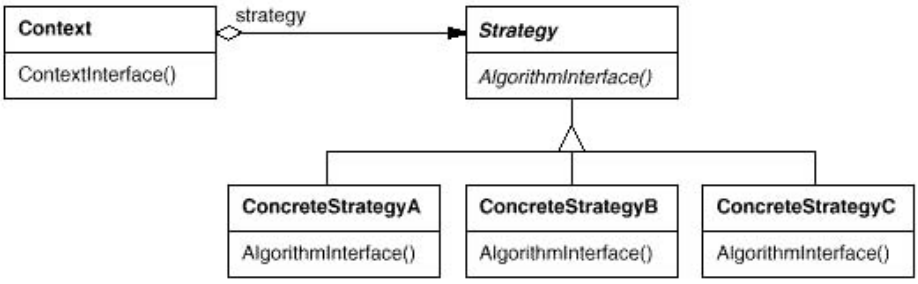


Figura 46. El patró *Strategy*.

En l'aplicació pertinent a aquest projecte, el patró *Strategy* s'ha utilitzat conscientment en dos casos. Si observem el diagrama de classes de l'aplicació, a la figura 45, veurem com hi ha dues situacions molt semblants a la que proposa el patró *Strategy* en la figura 46, però amb dues estratègies concretes enlloc de tres.

En el primer cas, la classe `Context` del patró *Strategy* a la nostra aplicació seria la nostra classe `ReacObject`, la classe `Strategy` seria la nostra `Link`, i les `ConcreteStrategy` serien la `AudioLink` i la `ControlLink`, mentre que l'algorisme a intercanviar seria bàsicament el mètode que s'encarrega de pintar els enllaços, el `Render`.

Així, des de la nostra classe `ReacObject`, que conté una instància de `Link`, podem cridar la funció que pinta l'enllaç pertinent a l'objecte de manera igual per a tots els tipus d'enllaç, però el `Link` pintarà o bé la forma d'ona d'un enllaç sonor o bé el fluix de partícules d'un enllaç de control, tot depenent de si la instància de `Link` que té el `ReacObject` és un `AudioLink` o un `ControlLink`.

L'altre cas té protagonistes en comú. Ara la classe `Context` serà la classe `ObjectContainer`, que conté un vector de `ReacObjects`, que ara serà la classe `Strategy`, mentre que, evidentment, les classes `ConcreteStrategy` seran les classes `AudioObject` i `ControlObject`. L'algorisme que canviarà d'una instància a l'altra serà la manera com es fan les connexions entre els objectes.

Cap dels dos casos en que hem aplicat el *Strategy* és estrictament fidel al patró que hem comentat, però si que els dos estan inspirats en ell.

6.3.4 Implementació del radial blur

De la manera que s'ha tractat el disseny i la implementació han quedat un parell de punts sense tractar adequadament. Concretament no s'ha dit pràcticament res de la implementació de l'efecte del *radial blur* i de la gestió dels temes visuals no se n'ha parlat suficientment. Així, en el que queda de capítol tractarem aquests dos punts.

La tècnica que emprarem per a implementar el *radial blur* consisteix en pintar tot el que vulguem que rebi aquest efecte directament en una textura més petita que la pantalla i després d'haver pintat tota la resta d'elements gràfics, anem aferrant aquesta textura, evidentment utilitzant *blending*, a sobre de la pantalla amb un petit desplaçament a cada iteració per aconseguir una distribució radial de la textura a sobre la pantalla.

L'acció de pintar directament sobre una textura és coneguda popularment com a *Render To Texture* i s'està començant a aplicar molt en els jocs per ordinador per a la realització de diferents efectes. En OpenGL, aquesta tècnica s'acostuma a realitzar de dues maneres diferents: una que utilitza funcions del OpenGL normals (podem veure el codi utilitzat en el llistat 6), i una altre en que s'utilitzen les extensions de OpenGL, concretament els *Pixel Buffers*, i que suposadament, és més ràpida (podem veure la seva implementació en el llistat 7).

Pintant en una textura més petita aconseguim major velocitat de processament, i si a més a la textura li apliquem els filtres de textures d'OpenGL, concretament el GL_LINEAR, quan posem la textura a tota la pantalla, aquesta ens surt difuminada.

```
inline void Render2Texture()
{
 glPushAttrib(GL_ALL_ATTRIB_BITS);

 glViewport(0, 0, OFF_SCREEN_TEX_SIZE, OFF_SCREEN_TEX_SIZE);
 glClear(GL_COLOR_BUFFER_BIT);

 glEnable(GL_BLEND);
 glBlendFunc(GL_SRC_ALPHA, GL_ONE);

 objs.RenderAudioLinks();

 if (BlurTex <= 0)
 {
        glGenTextures(1, &BlurTex);
        glBindTexture(GL_TEXTURE_2D, BlurTex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0,
                         OFF_SCREEN_TEX_SIZE, OFF_SCREEN_TEX_SIZE, 0);
 }
 else
 {
        glBindTexture(GL_TEXTURE_2D, BlurTex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                         OFF_SCREEN_TEX_SIZE, OFF_SCREEN_TEX_SIZE);
 }

 glDisable(GL_BLEND);

 glPopAttrib();
}
```

Llistat 6. La tècnica del Render To Texture en OpenGL 1.2, sense extensions.

```
inline void Render2TexturePBuffer()
{
 mypbuffer.Activate();
 glClear(GL_COLOR_BUFFER_BIT);
 objs.RenderAudioLinks();
 glBindTexture(GL_TEXTURE_2D, pbufferBlurTex);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, OFF_SCREEN_TEX_SIZE,
                     OFF_SCREEN_TEX_SIZE);
 mypbuffer.Deactivate();
}
```

Llistat 7. El Render To Texture utilitzant Pixel Buffers.

En els dos casos, el que fem és simplement pintar els enllaços sonors en una textura, el que passa és que en el primer cas hem de canviar el *viewport* i s'han d'utilitzar funcions més lentes, en canvi en el segon cas simplement es pinta directament sobre el *Pixel Buffer* i després es passa la imatge a una textura. La segona implementació és més ràpida però necessitem una targeta gràfica que suporti els *Pixel Buffers* i hem de utilitzar una classe anomenada pbuffer (no és la única manera d'emprar-los, però si la més fàcil), que podem trobar a la web de nVidia [24].

Pot semblar que aquesta segona implementació és més fàcil, però al final és tot el contrari, ja que és bastant complicat aconseguir compilar i fer el *link* amb les extensions OpenGL, sobretot sota Linux. En canvi, el resultat gràfic obtingut és el mateix.

Després d'haver pintat els enllaços sonors a la textura, el que hem de fer és tornar a pintar aquests enllaços però al *framebuffer* i juntament amb tots els altres elements a dibuixar, i un cop s'ha dibuixat tot el que s'ha de dibuixar hem de fer el render del *radial blur*. Com s'ha dit anteriorment, el que hem de fer és dibuixar un quadrat de tota la pantalla i aferrar-li la textura dels enllaços sonors, utilitzant la funció de *blending* adequada per mesclar la textura amb tot el

que s'ha dibuixat anteriorment. D'aquesta manera obtindríem un difuminat normal, però a nosaltres ens interessa un difuminat radial, per tant anirem dibuixant quadrats com aquest de manera radial respecte al centre de la imatge i canviant el nivell de transparència a cada iteració.

```
inline void RenderGLBlur()
{
  float alpha = 0.2f;
  float alphaInc = alpha / (float) numBlurSamples;

  float offset = 0.0f;
  float offsetInc = 0.025f;

  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL SRC ALPHA, GL ONE);

  if (r2t == R2T GL)  glBindTexture(GL TEXTURE 2D, BlurTex);
  else glBindTexture(GL_TEXTURE_2D, pbufferBlurTex);

  for (int i = 0;i < numBlurSamples;i++)
  {
        glColor4f(1.0f, 1.0f, 1.0f, alpha);

        glBegin(GL_TRIANGLE_STRIP);
        {
              glTexCoord2f(1.0 - offset, 1.0 - offset);
              glVertex2f( 1,  1);
              glTexCoord2f(offset, 1.0 - offset);
              glVertex2f(-1,  1);
              glTexCoord2f(1.0 - offset, offset);
              glVertex2f( 1, -1);
              glTexCoord2f(offset, offset);
              glVertex2f(-1, -1);

              offset += offsetInc;
              alpha -= alphaInc;
        }
        glEnd();
  }

  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Llistat 8. Com es pinta el *radial blur*.

Aquesta tècnica utilitza funcions OpenGL 1.2 i funciona a un *frame-rate* raonable sempre i quan no fem un número massa alt d'iteracions. Una alternativa a aquest mètode és la utilització del llenguatge Cg i la programació del processador de la targeta gràfica.

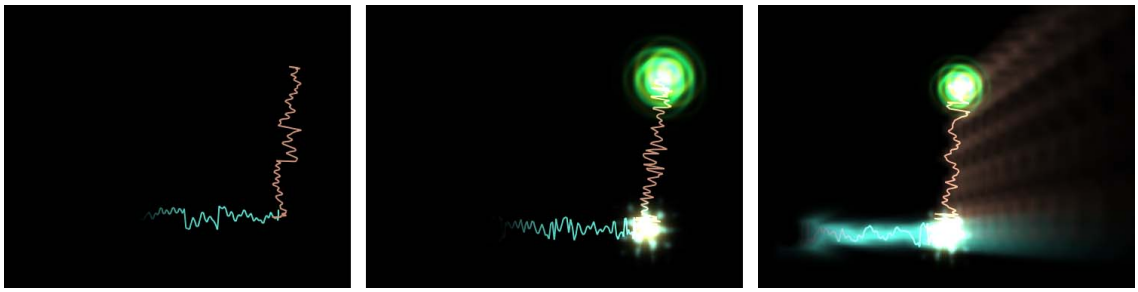


Figura 47. A l'esquerra la textura amb els enllaços que s'emprarà per fer el *blur*, al mig la representació sense l'efecte i a la dreta la imatge amb l'efecte aplicat.

Utilitzant Cg podem fer quatre de les iteracions realitzades per l'altre mètode posant un sol quadrat. Així, amb tres quadrats obtindríem el mateix efecte que amb 12 iteracions amb el mètode convencional. Per aconseguir això haurem de dissenyar un *vertex shader* i un *pixel shader* que facin una feina similar a la que fa el codi anterior, i cridar-los n vegades per aconseguir 4*n iteracions de les anteriors.

Així doncs, el que farem serà utilitzar un *pixel shader* que sigui capaç de combinar quatre colors donats per quatre parells de coordenades de textura en un sol color, assignant un pes específic a cada un d'aquests colors. Aquest *pixel shader* rebrà unes coordenades de textura que hauran estat modificades per un *vertex shader* que a partir d'un vèrtex i un vector de coordenades de textura crearà tres vectors de coordenades de textura aplicant el desplaçament radial necessari per emular l'efecte comentat. Així, s'aconseguirà el mateix efecte pintant un sol quadrat que en quatre iteracions del bucle del llistat 8. Per tant el que farem quan apliquem aquest efecte en Cg serà pintar en OpenGL un quadrat per cada quatre de les antigues iteracions.

```
v2f main(app2v IN, uniform float4x4 ModelViewProj)
{
 v2f OUT;

 OUT.position = mul(ModelViewProj, IN.position);

 OUT.color = IN.color;          // Passarem al pixel shader el color
                                // tal com l'envia l'OpenGL
 OUT.texCoord0 = IN.texCoord;

 float offset1 = 0.025f;        // Fixem els desplaçaments que tindrà
 float offset2 = 0.050f;        // el vèrtex rebut
 float offset3 = 0.075f;

 if (IN.position.x > 0)         // Desplaçem les coordenades de textura
 {                              // del vèrtex de manera radial
        OUT.texCoord1.x = IN.texCoord.x - offset1;
        OUT.texCoord2.x = IN.texCoord.x - offset2;
        OUT.texCoord3.x = IN.texCoord.x - offset3;
 }
 else
 {
        OUT.texCoord1.x = IN.texCoord.x + offset1;
        OUT.texCoord2.x = IN.texCoord.x + offset2;
        OUT.texCoord3.x = IN.texCoord.x + offset3;
 }

 if (IN.position.y > 0)
 {
        OUT.texCoord1.y = IN.texCoord.y - offset1;
        OUT.texCoord2.y = IN.texCoord.y - offset2;
        OUT.texCoord3.y = IN.texCoord.y - offset3;
 }
 else
 {
        OUT.texCoord1.y = IN.texCoord.y + offset1;
        OUT.texCoord2.y = IN.texCoord.y + offset2;
        OUT.texCoord3.y = IN.texCoord.y + offset3;
 }
               // Passem el vertex, el seu color i els diferents vectors
 return OUT;   // de coordenades de textura al pixel shader
}
```

Llistat 9. El *vertex shader* utilitzat en l'aplicació.

```
half4 main(v2f IN,      uniform sampler2D blurMap0,
                        uniform sampler2D blurMap1,
                        uniform sampler2D blurMap2,
                        uniform sampler2D blurMap3) : COLOR
{
 half4 weight = half4(0.5, 0.4, 0.3, 0.2);          // Declarem els pesos
                                                    // de cada color
 half4 c0 = tex2D(blurMap0, IN.texCoord0);          // Obtenim els colors
 half4 c1 = tex2D(blurMap1, IN.texCoord1);          // a partir de les
 half4 c2 = tex2D(blurMap2, IN.texCoord2);          // coordenades de
 half4 c3 = tex2D(blurMap3, IN.texCoord3);          // textura

 half4 color = c0 * weight.r + c1 * weight.g        // Fem la mescla
               + c2 * weight.b + c3 * weight.a;     // dels colors

 color.a = IN.color.a;          // Assignem la coordenada de transparència
                                // que vindrà donada de l'OpenGL
 return color;
}
```

Llistat 10. El *pixel shader* encarregat de fer la mescla de colors.

El problema és que tot i que es pinten menys quadrats, la targeta gràfica (les proves s'han fet en una geforce FX 5700 LE) tarda aproximadament el mateix temps per realitzar les dues tècniques. Possiblement en targetes més potents, on el sistema de *shaders* és més avançat, la diferència de rendiment seria més gran.

A continuació s'ofereix una taula amb els *frame-rates* obtinguts per mitjà de les diferents tècniques comentades realitzant vuit iteracions de *radial blur*. Les files indiquen de quina manera s'ha fet el *Render To Texture* i les columnes com s'ha pintat l'efecte. Així es pot veure fàcilment les diferencies de rendiment existents entre les possibles combinacions. Cal dir però, que el resultat visual és el mateix en tots els casos. Per aconseguir aquestes dades s'ha tret la limitació de 30 imatges per segon i s'ha mirat el número màxim d'imatges per segon que dona l'aplicació en les diferents possibilitats.

	OpenGL 1.2	**Utilitzant Cg**
OpenGL 1.2	73 *fps*	77 *fps*
Utilitzant Pixel Buffers	76 *fps*	79 *fps*

Taula 2. Els *frame-rates* obtinguts en una geforce FX 5700 LE amb 256 Mb de RAM sense cap tipus d'*overclocking* i a una resolució de 1024 x 768 píxels.

Tot i que en la taula no es veu reflectit, altres proves realitzades demostren que el coll d'ampolla de l'aplicació, parlant del rendiment gràfic, es troba a l'hora de pintar l'efecte. El *blending* píxel per píxel de dues imatges de 1024 x 768 és costós, però si el *blending* es fa 8 o més vegades encara ho és més. Aquest és el preu que hem de pagar per gaudir del *radial blur*, però donat que el requeriment diu que s'han d'aconseguir 30 imatges per segon, crec que ens podem permetre fer-lo servir.

6.3.5 La gestió dels temes visuals

Amb el que hem dit fins ara ha quedat bastant clar que l'aplicació estarà preparada per a funcionar amb diferents temes visuals. Hem dit que hi haurà una classe que emmagatzemarà la informació sobre el tema visual respecte a cada tipus d'objecte i que haurem de crear un nou format de fitxer on posar totes les dades referents al *skin*, però no hem dit ni com serà aquest fitxer ni com passarem les dades del fitxer a l'aplicació. Tot això ho veurem en aquest subapartat.

El format del fitxer de temes visuals

El fitxer que albergarà les dades necessàries serà un simple fitxer de text amb una estructura concreta. Repassarem primer les dades que haurà de contenir el fitxer en qüestió:

- El color de fons que tindrà la visualització.
- Si els enllaços han de seguir o no les corbes de Bézier.
- Si ha d'emprar o no el *radial blur* i si s'ha d'utilitzar, amb quantes iteracions
- El nom del fitxer de la textura que utilitzarem per les partícules.
- El nom del fitxer de la textura de la màscara per les partícules si escau.
- El número de partícules que s'utilitzaran per els enllaços de control.
- Els noms dels fitxers de les textures que s'utilitzaran per a les aures dels objectes.

- Per a cada tipus d'objecte, com serà la seva aura, és a dir, per quines capes estarà formada, indicant les textures i el color que se li aplicarà, i també el color de la seva connexió, en cas que sigui un objecte d'àudio.

Tota aquesta informació anirà en el fitxer en format de text pla. En el següent llistat posem un tros comentat del tema visual inspirat en les visualitzacions dels reproductors d'àudio.

```
bkg clr 0.0 0.0 0.0 1.0                 // El color de fons serà negre

bzr on                                  // Els enllaços seguiran Bézier

fxs on 12                               // S'utilitzarà radial blur amb
                                        // 12 iteracions
ctr tex Textures/particula.bmp          // S'emprarà aquesta textura
ctr num 30                              // Emprarem 30 partícules

tex 0 Textures/aura1.bmp                // Les aures tindran aquestes textures
tex 1 Textures/aura2.bmp
tex 2 Textures/aura4.bmp

obj osc                                 // L'objecte de tipus oscil·lador
{
  2 clr 0.0 0.0 1.0 1.0 msk -1          // Utilitzara la textura 2 amb color
  0 clr 0.0 1.0 1.0 1.0 msk -1          // blau i sense màscara...
  1 clr 1.0 1.0 0.0 1.0 msk -1

  lnk rand                              // El color de la connexió serà
}                                       // aleatori
```

Llistat 11. Un tros d'un fitxer de tema visual.

La lectura del fitxer

Per tal de llegir i interpretar la informació continguda en aquests fitxers s'ha creat una nova funció en el mateix *main* del programa. Per qüestió de comoditat no s'ha utilitzat la orientació a objectes per aquesta tasca, encara que segurament hauria estat més adequat.

Aquesta funció implementa un mini *parser* capaç d'interpretar l'estructura d'aquests fitxers. Quan troba informació sobre la visualització en general, com pot ser el color de fons, ho emmagatzema en la variable adequada, quan troba una textura la carrega en memòria i la prepara per OpenGL, i quan troba una descripció d'un tipus d'objecte crea una nova instància de la classe `ObjectSkin` i hi emmagatzema tota la informació referent a aquest tipus d'objecte.

Només ens queda dir com s'informa a l'aplicació quin és el fitxer de *skin* que ha de carregar. Això ho farem simplement passant el nom del fitxer com argument del programa quan ens disposem a executar-lo. Si no posem cap fitxer com argument el programa en carregarà un per defecte.

7. Resultats i conclusions

Arribats a aquest punt només queden dues coses a fer: la primera, observar els resultats obtinguts; la segona, mirar enrere i valorar tota la feina feta. Això és el que es proposa en aquest capítol que tanca el projecte.

Primer veurem com ha quedat l'aplicació finalment i quina és la seva resposta quan la posem a prova. Després, i per acabar el present document, es faran reflexions sobre la feina duta a terme, s'exploraran les línies futures de treball i s'exposarà una valoració personal.

7.1 Resultats obtinguts

L'objectiu d'aquest projecte, com diu el seu títol, és la creació d'un mòdul interactiu de visualització d'àudio en temps real. Per tant, degut a que el resultat és un element totalment audiovisual amb imatge i so dinàmics, no podem mostrar els resultats de la manera més adequada en aquest document, ja que per veure'ls adequadament es necessitaria un prototipus físic de la reacTable*. El que farem, doncs, és mostrar unes captures de pantalla que reflecteixin els resultats de l'aplicació i després dues fotografies de l'últim prototipus de la reacTable*.

En les següents pàgines podran veure dos exemples de visualitzacions creades per el mòdul desenvolupat. Per a cada exemple s'ha posat primer una captura de pantalla del prototipus software de la reacTable* per veure la situació dels objectes sobre la taula i les connexions existents entre ells i a baix s'ha posat una captura de pantalla de la visualització resultant de tal situació sobre la taula fictícia.

Com és poden imaginar, els objectes en el prototipus software són les caixes i les rodones de color vermell, el número blanc que duen a sobre és la cara que està en contacte amb la taula, la ratlla negre ens indica la seva orientació i si l'objecte està actiu s'envolta l'objecte amb una línia verda. Les connexions, evidentment, són les línies que apareixen entre els objectes i són de color negre en el cas de connexions d'àudio, gris en el cas de les connexions de control i vermelles en el cas que es tracti d'un enllaç fort entre dos objectes.

Per mostrar els resultats s'han escollit dos exemples, una visualització per a cada tema visual existent, i així també es pot apreciar la capacitat de canvi dels temes visuals. Podran veure una visualització utilitzant el *skin* basat en les visualitzacions dels reproductors de mp3, i una altra basada en el *skin* inspirat en l'obra de Kandinsky.

Al mostrar els resultats d'una manera estàtica, no és poden apreciar ni com flueixen les partícules dels enllaços de control, ni com es transmeten les ones d'uns objectes als altres, ni com reaccionen els elements a la manipulació dels objectes en temps real i per tant tampoc podem veure la flexibilitat dels enllaços basada en les corbes de Bézier. Tot i això, crec que es pot veure més o menys com respon l'aplicació.

En les imatges posteriors podran veure com s'han assolit alguns dels requeriments com representar l'estat en que es troba l'instrument, dibuixar aures a sota els objectes i les connexions entre ells diferenciant entre connexions de control i connexions d'àudio, així com també és podrà confirmar la possibilitat d'utilitzar varis temes visuals.

Altres requeriments no podran ser comprovats mitjançant les següents imatges. Així, no es podrà analitzar si la interacció és en temps real o si la visualització aconsegueix un *frame-rate* de 30 imatges per segon. Ni tampoc si el sistema funciona realment tant sota Linux com sota Windows. Però els confirmo que aquests requeriments han estat també assolits i espero poder-ho demostrar en la presentació.

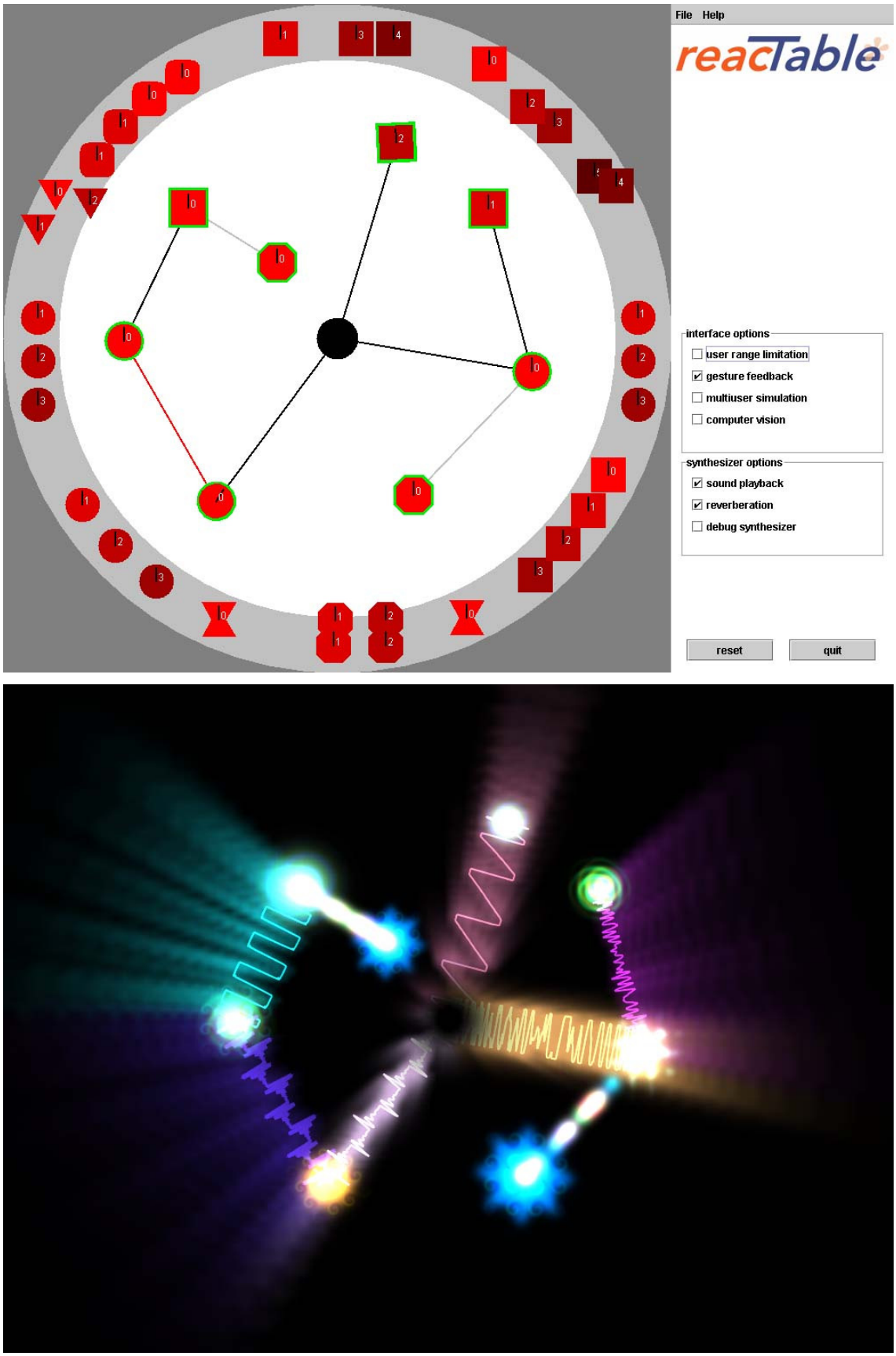


Figura 48. Captures de pantalla del prototipus software de la ReacTable* i la visualització corresponent creada pel mòdul aquí implementat.

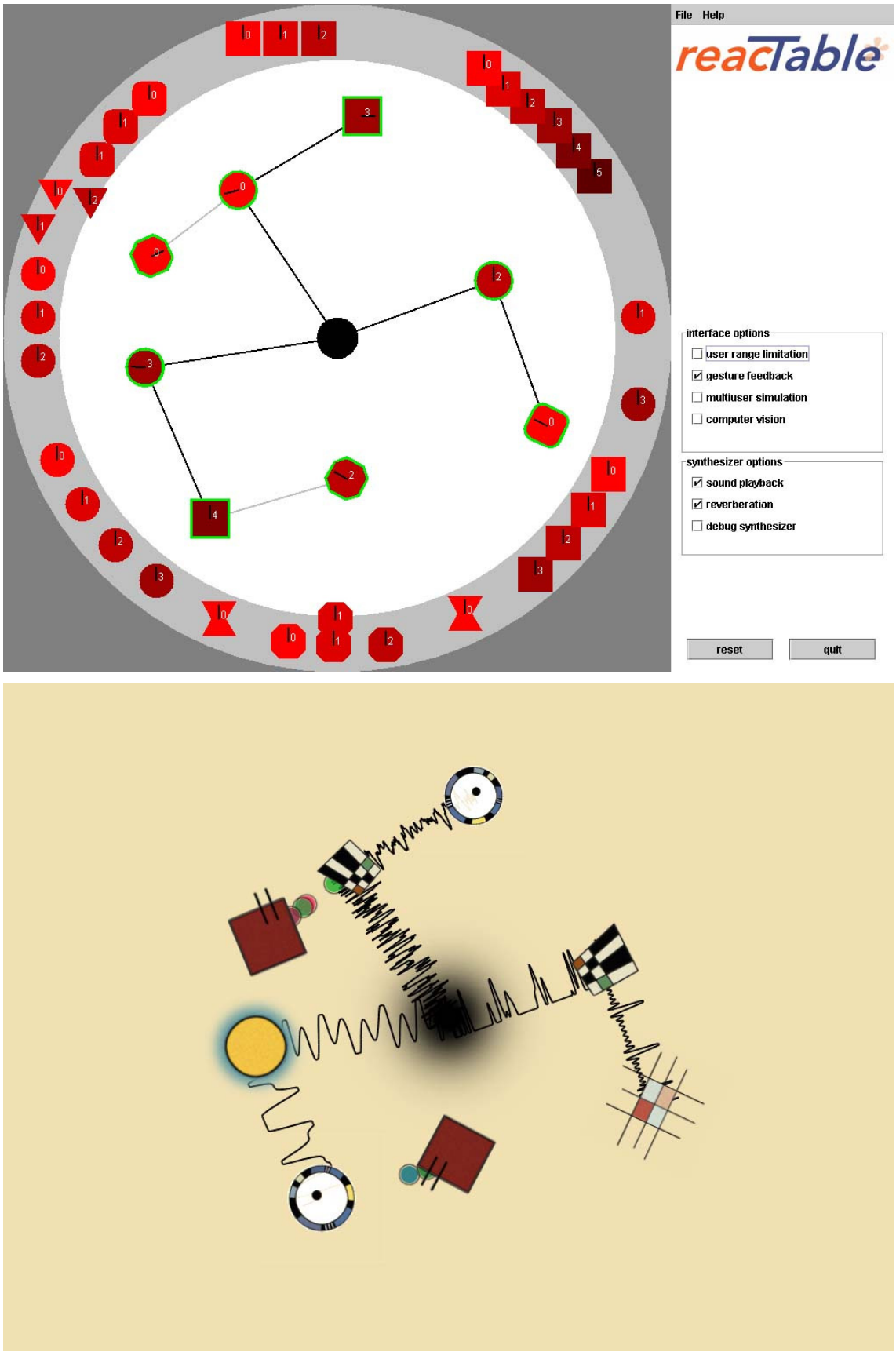


Figura 49. A dalt, una captura de pantalla del prototipus software de reacTable*, a sota la visualització creada amb el tema visual inspirat en l'obra de Kandinsky.

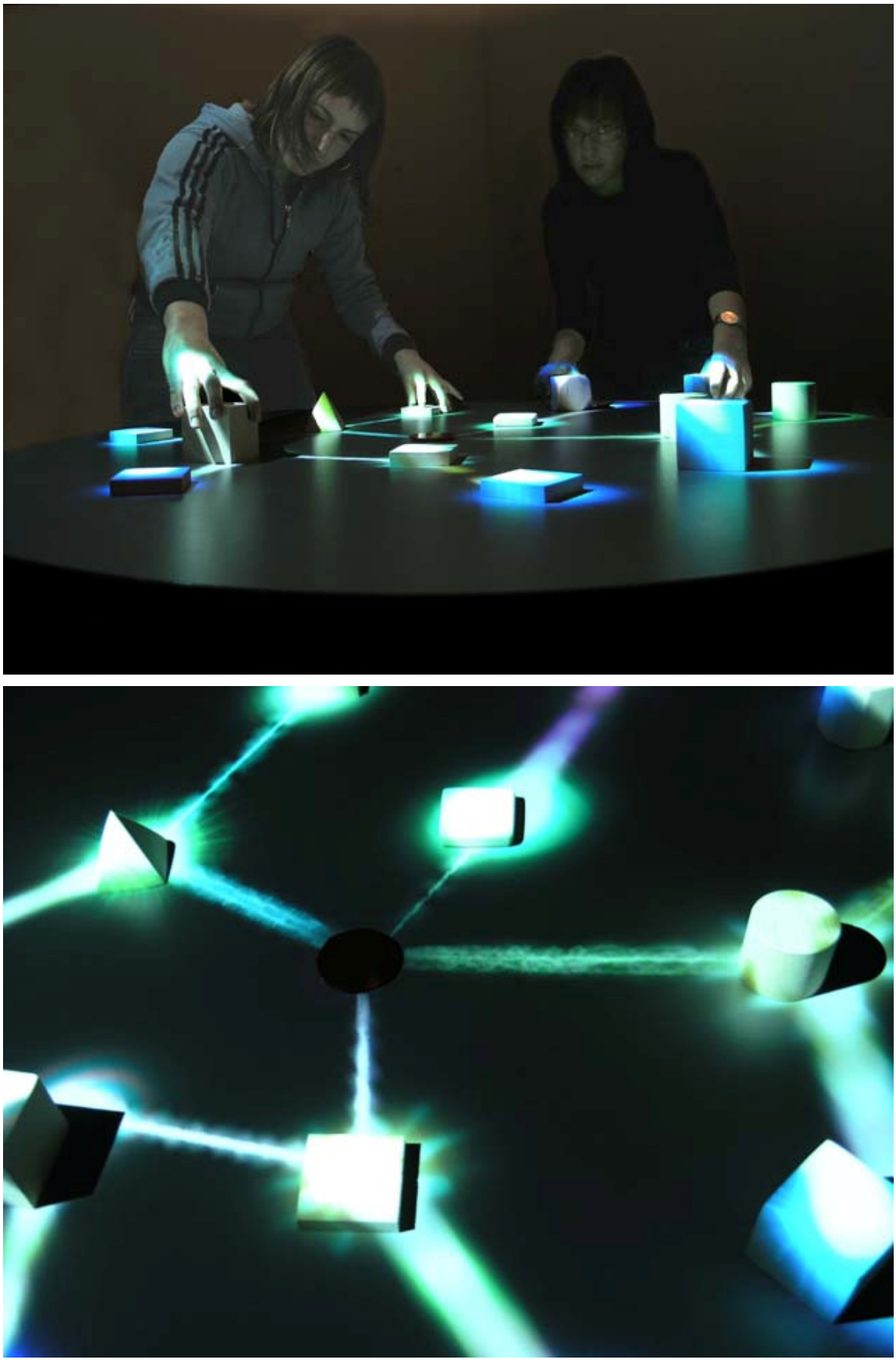


Figura 50. Fotos del prototipus actual de la reacTable* en acció. En aquesta versió la visualització es projecta des de dalt.

7.2 Reflexions sobre la feina feta

Tot i que s'han complert tots els requeriments que es varen plantejar al principi del projecte, crec que encara queda molta feina a fer sobre aquest mòdul de visualització, sobretot pel fet que la reacTable* encara està en desenvolupament. Però el que si que s'ha aconseguit és crear una base sobre la que treballar.

Una vegada acabat el projecte, podem dir que l'elecció de les eines a utilitzar va ser encertada, el C++ i l'OpenGL han donat el rendiment que s'esperava d'ells, mentre que la llibreria SDL m'ha sorprès cap a bé. Respecte al Cg, m'ha semblat un llenguatge força interessant, però en el cas concret del *radial blur* no ens ha aportat cap gran millora respecte al OpenGL o almenys no he trobat la manera d'aconseguir-ho, igualment crec que es tracta d'un llenguatge amb un gran futur i amb moltes possibilitats.

Crec que l'organització de les classes és adequada, però potser es podria millorar. Potser si s'haguessin aplicat més patrons de disseny la implementació hauria estat una mica més elegant i més orientada a objectes, però igualment així compleix la seva funció.

Pel que fa a la implementació purament gràfica, crec que les aures s'han quedat bastant bé i amb prou flexibilitat per oferir diferents temes visuals. Les connexions també s'han aconseguit, però potser les de control s'haurien de correspondre més amb el fluix real de dades. I referent als temes visuals, penso que és una cosa que ha sortit força interessant i el contrast entre els dos temes existents ara per ara dona moltes possibilitats.

Per tant, crec que s'ha aconseguit el que es volia des de un principi, crear un mòdul que donés el feedback visual necessari a l'usuari de la reacTable* per tal que aquest pugui tocar l'instrument mitjançant una interfície més intuïtiva.

7.3 Línies futures de treball

Malgrat que s'han assolit els requeriments inicials, s'ha de reconèixer que la feina no s'acaba aquí, i és que el projecte reacTable* del qual forma part aquest projecte de fi de carrera no està acabat. La reacTable* és un projecte en desenvolupament que ara per ara està en evolució constant. Per tant, les línies futures de treball sobre el mòdul de visualització plantejat en aquest document, estaran lligades al desenvolupament futur de la reacTable*.

Així, l'aplicació desenvolupada s'haurà d'anar adaptant als canvis que pugui sofrir la reacTable* i evolucionar de forma conjunta. A mesura que es vagin afegint funcionalitats a la reacTable* s'hauran d'anar incorporant al mòdul de visualització noves formes de representació capaces de donar el feedback gràfic necessari a aquestes noves funcionalitats.

A part de tot això, hi ha certs aspectes del mòdul implementat que poden ser millorats, i aconseguir així resoldre els requeriments d'una manera més eficient i adequada. A continuació s'oferirà una llista de possibles millores a realitzar sobre el projecte i que per tant formarien part del treball futur a realitzar:

- Es podria canviar el format dels fitxers de temes visuals, canviant l'utilització del format propi per una manera més elegant de guardar aquesta informació com pot ser la utilització del llenguatge XML. Així, també podríem canviar el mini parser utilitzat per un que utilitzi una de les moltes llibreries que faciliten la lectura de XMLs.
- Una altre possible millora referent als *skins* seria oferir la possibilitat de un canvi de tema visual en temps d'execució.
- Fer un *mapping* adequat entre els fluixos de partícules, que s'utilitzen per a representar els enllaços de control, i les dades de control que realment es passen d'un objecte a un altre.
- S'hauria de mirar d'optimitzar el codi creat i millorar el control d'errors existent, que actualment és bastant pobre.

- Es pot tractar d'afegir nous efectes a les visualitzacions o intentar millorar els que ja hi ha. Així com explorar encara més en profunditat les possibilitats que ens pot oferir un llenguatge com Cg per a la realització d'aquests.

Com es pot veure, queda encara molta feina a fer. Ara el projecte està encaminat, però falta encara camí per recórrer.

7.4 Valoració personal del projecte

Quan fa més o menys un any vaig començar aquest projecte no era plenament conscient de la quantitat de feina que m'esperava. Era el projecte de major envergadura al que m'havia enfrontat durant els anys de carrera. Em quedaven per davant moltes hores d'estudi i de desenvolupament.

Una gran part del temps dedicat al projecte l'he invertit en investigació i estudi. Aquesta labor m'ha conduït a investigar els sistemes de visualització d'àudio existents i he acabat força sorprès al veure que els primers sistemes visuals musicals tenen segles d'història.

Durant el desenvolupament de l'aplicació, he ampliat els coneixements de C++ i OpenGL i m'he familiaritzat un poc més amb l'entorn Linux. Però també he hagut d'aprendre a manejar una nova llibreria, la SDL, i he aprés un nou llenguatge, el Cg, que malgrat que no ha tingut l'aplicació que m'esperava en aquest projecte l'he trobat força interessant.

Cal esmentar que apart de la motivació natural pel fet que es tracta del projecte de final de carrera, em sentia especialment motivat degut a la meva passió per la música i també perquè m'agrada molt la infografia. I es que aquest projecte es basa principalment en això, en la música i en els gràfics per ordinador. A més m'he sentit especialment motivat perquè la reacTable* em sembla un projecte força innovador i molt interessant.

Considero que durant el període que ha durat aquest projecte he après moltes coses noves i he madurat com a enginyer, i suposo que aquest és l'objectiu d'un projecte de final de carrera. Així, estic content tant pels resultats del projecte com per totes les coses que aquest m'ha aportat.

8. Bibliografia

[1] JORDÀ, Sergi. *Sonographical Instruments: From FMOL to the reacTable**. Proceedings of the 3rd Conference on New Instruments for Musical Expression (NIME 03), Montreal, Canada, 2003.

[2] JORDÀ, Sergi. *Interactive Music Systems For Everyone: Exploring Visual Feedback As a Way for Creating More Intuitive, Efficient And Learnable Instruments*. Proceedings of the Stockholm Music Acoustics Conference (SMAC 03), Stockholm, Sweden, 2003.

[3] KALTENBRUNNER, Martin; O'MODHRAIN, Sile; COSTANZA, Enrico. *Object Design Considerations for Tangible Musical Interfaces*. Proceedings of the COST287-ConGAS Symposium on Gesture Interfaces for Multimedia Systems, Leeds, United Kingdom, 2004.

[4] KALTENBRUNNER, Martin; GEIGER, Günter; JORDÀ, Sergi. Dynamic *Patches for Live Musical Performance*. Proceedings of the 4th Conference on New Interfaces for Musical Expression (NIME 04), Hamamatsu, Japan, 2004.

[5] BERNARD, Klein. *Colour-Music: The Art of Light*. Crosby, Lockwood & Son, London, 1927.

[6] POPPER, Frank. *Origins and Development of Kinetic Art*. New York Graphic Society, 1968.

[7] PEACOCK, Kenneth. *Instruments to perform color-music: Two centuries of technological experimentation*. Leonardo, 1988.

[8] MORITZ, William. The *Dream of Color Music, And Machines That Made it Possible*. Animation World Magazine, Issue 2.1, April 1997.

[9] ROADS, Curtis. *The Computer Music Tutorial*. Cambridge, MA: MIT Press, 1996.

[10] GOLAN, Levin. *Painterly Interfaces for Audiovisual Performance*. Master Thesis, Massachusets Institute of Technology, 2000.

[11] HAEBERLI, Paul. *DynaDraw*. Silicon Graphics Corporation, Mountain View, California, 1989.

[12] MAEDA, John. *Digital Expression: A Framework for Artistic Creation of Forms On and Using the Computer*. Ph.D. Thesis, University of Tsukuba, Japan, 1995.

[13] SNIBBE, Scott. *The Motion Phone*. Proceedings of Ars Electronica '96. Christine Schopf, ed. 1996.

[14] IWAI, Toshio. *Music Insects*. Interactive exhibit at the San Francisco Exploratorium. 1992.

[15] COSTANZA, Enrico; SHELLEY, S. B.; Robinson, J. *D-touch: A Consumer-Grade Tangible Interface Module and Musical Applications*. Proceedings of Conference on Human-Computer Interaction (HCI03), Bath, UK, 2003.

[16] PUCKETTE, Miller. *Pure Data*. Proceedings of the International Computer Music Conference. San Francisco: International Computer Music Association, 1996.

[17] Pàgina web del SmallFish, http://hosting.zkm.de/wmuench/small_fish

[18] BLAINE, Tina; PERKINS, Tim. *Jam-O-Drum, A Study in Interaction Design*. Proceedings of the ACM DIS 2000 Conference, ACM Press, NY, August 2000.

[19] POUPYREV, Ivan. *Augmented Groove: Collaborative Jamming in Augmented Reality*. ACM SIGGRAPH 2000 Conference Abstracts and Applications, 2000.

[20] PATTEN, James; RECHT, Ben; ISHII, Hiroshi. *Audiopad: A Tag-based Interface for Musical Performance*. Proceedings of the 2002 Conference on New Interfaces for Musical Expression (NIME-02), Dublin, Ireland, 2002.

[21] STROUSTRUP, Bjarne. *An overview of the c++ programming language*. The Handbook of Object Technology, 1999.

[22] WOO, Mason; NEIDER, Jackie; DAVIS, Tom; SHREINER, Dave. OpenGL *Programming Guide*. third ed. Addison-Wesley, 1999.

[23] Pàgina web de SDL, http://www.libsdl.org

[24] La pàgina web de desenvolupadors de nVidia, http://developer.nvidia.com

[25] La pàgina web de Cg, http://developer.nvidia.com/page/cg_main.html

[26] RANDIMA, Fernando; KILGARD, Mark. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[27] RANDIMA, Fernando. *GPUGems*. Addison-Wesley, 2004.

[28] La pàgina web de ShaderTech: The GPU programming, http://www.shadertech.com

[29] La pàgina web de Neon Helium Productions, http://nehe.gamedev.net

[30] La pàgina web d'OpenGL, http://www.opengl.org

[31] FOLEY, James. *Computer Graphics: Principle and Practice second edition in C*. Addison-Wesley, 1990.

[32] La pàgina web Game Tutorials, http://www.gametutorials.com

[33] La pàgina web de Mark Harris, http://www.markmark.net/

[34] La pàgina web de GameDev.net, http://www.gamedev.net

[35] La pàgina web de Flip Code, http://www.flipcode.com

[36] La pàgina web del Winamp, http://www.winamp.com