AMETRINE  
(framework grafic)

|  |  |  |
| --- | --- | --- |
| Îndrumător |  | Elev |
| Prof. Butyka Mihai |  | Truța Andrei |

# **CUPRINS**

|  |  |
| --- | --- |
| 1. [Introducere](#_Introducere) ...................................................................................... | pag 3 |
| * 1. [Prezentare generală](#_Prezentare_generală) |  |
| * 1. [Motivație](#_Motivație) |  |
| * 1. [Conținut](#_Conținut) |  |
| 1. [Istoria bibliotecilor grafice](#_Istoria_bibliotecilor_grafice) ............................................................. | pag 4 |
| * 1. [Definiție](#_Definiție) |  |
| * 1. [Începutul](#_Începutul) |  |
| * 1. [Astăzi](#_Astăzi) |  |
| 1. [Ametrine](#_Ametrine) ......................................................................................... | pag 5 |
| * 1. [Ideologie](#_Ideologie) |  |
| * 1. [Aspect](#_Aspect) |  |
| 1. [Limbaje de implementare](#_Limbaje_de_implementare) ............................................................... | pag 6 |
| 1. [Lista dinamică](#_Lista_dinamică) ................................................................................ | pag 7 |
| * 1. [Structură și inițializare](#_Structură_și_inițializare) |  |
| * 1. [Utilizare și curățare](#_Utilizare_și_curățare) |  |
| 1. [Lista dinamică „înghesuită”](înghesuită#_Lista_dinamică_) ............................................................ | pag 9 |
| * 1. [Concept și structură](#_Concept_și_structură) |  |
| 1. [I/O al sistemului](#_I/O_al_sistemului) .............................................................................. | pag 10 |
| 7.1 [Manager de evenimente](#_Manager_de_evenimente) |  |
| 7.2. [Tastatura](#_Tastatura) |  |
| 7.3. [Mouse](#_Mouse) |  |
| 7.4. [Proceduri callback](#_Proceduri_callback) |  |
| 7.5. [Sistemul de ferestre](#_Sistemul_de_ferestre) |  |
| 1. [Implementarea OpenGL](#_Implementarea_OpenGL) ................................................................. | pag 16 |
| * 1. [Shadere](#_Shadere) |  |
| * 1. [Buffere](#_Buffere) |  |
| * 1. [Uniforme (variabile globale)](#_Uniforme_(variabile_globale)) |  |
| * 1. [Texturi](#_Texturi) |  |
| * 1. [Pipeline](#_Pipeline) |  |
| * 1. [Render pass](#_Render_pass) |  |
| 1. [Structură și modul de lucru](#_Structura_și_modul) ............................................................. | pag 22 |
| * 1. [Cerințe obligatorii](#_Cerințe_obligatorii) |  |
| * 1. [Desenarea pe ecran](#_Desenarea_pe_ecran) |  |
| 1. [OBJ Loader](#_OBJ_Loader) .................................................................................... | pag 25 |
| * 1. [Structură](#_Structură) |  |
| * 1. [Implementare](#_Implementare) |  |
| 1. [Pământul](#_Pământul) ........................................................................................ | pag 27 |
| * 1. [Detalii](#_Detalii) |  |
| 1. [Bibliografie](#_Bibliografie) .................................................................................... | pag 29 |

# **Introducere**

## **Prezentare generală**

În scopul acestei lucrări am ales să creez un set de funcții, proceduri și structuri care îmbinate formează un framework grafic cross-platform (Windows și UNIX) bazat pe API-ul de comunicare la nivel mic cu o unitate de procesare grafică OpenGL. Numele ales este ,,Ametrine”, care provine de la piatra prețioasă cunoscută sub același nume.

## **Motivație**

Un proiect care necesită un suport grafic poate deveni extrem de complex într-un timp scurt, lucru care reduce eficiența utilizatorului prin încetinirea proceselor de depanare, reîntoarcere asupra codului deja scris și clădire a unor noi părți ale proiectului. De asemenea, este nevoie de un efort semnificativ pentru a acumula cunoștințele necesare utilizării unui API grafic precum OpenGL, Vulkan sau Metal, acestea fiind diverse și cantitative. Astfel, am creat framework-ul grafic Ametrine pentru a ușura munca utilizatorului într-un proiect care are nevoie de o unitate de procesare grafică.

## **Conținut**

Ametrine constă în trei mari unități distincte însă suprapuse. Prima unitate este reprezentată de ,,platformă”, care se ocupă de comunicarea cu sistemul de operare: evenimente ale ferestrelor, I/O de la dispozitivele periferice, citirea de pe disk și crearea fișierelor. A doua unitate este reprezentată de motorul grafic; este cea mai complexă parte, care oferă funcții și proceduri care se axează pe un model de lucru abstractizat de la modul de folosire al OpenGL-ului standard. Se ocupă cu preluarea datelor grafice de la utilizator și prelucrarea acestora pentru ca în final să fie afișate pe ecran, după instrucțiunile definite. Unitatea finală este cea de stocare a datelor într-un mod eficient și ușor de folosit. Sistemul se folosește de mai multe liste pentru a avea o amprentă minimă în memorie însă fără a fi nevoie de căutarea obiectelor la fiecare realocare de memorie. În capitolele ce urmează, acest sistem va fi descris în detaliu, însă per ansamblu, constă în două structuri informatice: o listă dinamică simplă și o listă dinamică ,,înghesuită”, care se folosește de prima pentru a memora datele în funcție de un identificator constant.

# **Istoria bibliotecilor grafice**

## **Definiție**

O bibliotecă grafică este o bibliotecă de programe concepută pentru a ajuta la redarea graficii computerizate pe un monitor. Acest lucru implică de obicei furnizarea de versiuni optimizate ale funcțiilor care se ocupă de sarcini obișnuite de render și se poate face doar în software, rulând pe CPU și accelerat hardware de un GPU, în cazul PC-urilor moderne. Utilizând aceste funcții, un program poate asambla o imagine pentru a fi redată pe un monitor. Bibliotecile grafice sunt utilizate în principal în jocuri video și simulări.

## **Începutul**

Primul sistem de proiectare grafică (CAD) a fost dezvoltat de Evan Sutherland ca teză de doctorat la MIT în 1963. Se numea Sketchpad și permitea unui utilizator să schițeze o piesă mecanică pe ecranul unui computer, să pună constrângeri asupra piesei și să pună computerul să calculeze design-ul exact al piesei. A fost o idee revoluționară care a dus la recunoașterea potențialului ideii de grafică cumputerizată.

La început, dacă doreai să creezi grafică pe computer, trebuia să scrii programe care comunicau direct cu hardware-ul grafic. Pe măsură ce au fost create noi modele hardware, software-ul a trebuit să fie complet rescris pentru a funcționa cu noul hardware. De-a lungul anilor 1970, au fost create multe „biblioteci grafice” pentru a rezolva această problemă, dar majoritatea acestora au lucrat cu sisteme de operare individuale și nu au putut fi transferate cu ușurință pe noi tehnologii (sisteme de operare și hardware). În anii 1980, ISO (International Organization for Standardization) a încercat să creeze o bibliotecă standard de grafică numită PHIGS. S-a afirmat că este răspunsul definitiv pentru crearea de grafică pe computer pe toate platformele de calcul.

## **Astăzi**

În prezent, bibliotecile grafice sunt des întâlnite si răspândite. Pe langă motoarele grafice de dezvoltare a jocurilor precum Unity și Unreal Engine care predomină industria jocurilor pe calculator, putem întâlni aceste biblioteci și în dezvoltarea aplicațiilor software. Qt este cea mai populară alegere pentru persoanele care necesită o aplicație grafică cross-platform, stând la baza unor programe precum KDE, Opera Browser, Google Earth și Quora.

# **Ametrine**

## **Ideologie**

Ideea mea a fost una simplă: să creez o bibliotecă de funcții și proceduri care să îmi permită mie și oricărui alt utilizator o modalitate de a lucra cu OpenGL mai accesibilă. De exemplu, procesul de „legare” a tuturor listelor de date, alegerea corectă a modului de interpretare ale acestora și așa mai departe este ascuns în spatele unei singure funcții care se ocupă de toate - aceasta reprezintă ideea de bază.

Legat de această ușurință a folosirii, am decis de la început că este necesară o soluție pentru gestionarea „obiectelor” create de OpenGL și nu numai. Astfel am decis să implementez un sistem de gestiune a memoriei care este cât se poate de simplu din punctul de vedere al utilizatoului: în momentul creării unui obiect, se returnează un identificator sub formă de număr. Din acel moment, ID-ul obținut este suficient și necesar pentru orice operație grafică, pe când în spatele cortinei poziția acestuia în memorie este liberă să varieze, totul fiind ascuns față de utilizator.

Astfel, am ales să pun accentul pe simplitate. La momentul creerii unui obiect, există o listă lungă de detalii care trebuie specificate, însă toate recurg la valori pre-alese în cazul în care nu sunt specificate, fiind nevoie să fie date doar detaliile critice, scurtând codul și crescând eficiența vitezei de lucru. Bineînțeles, simplitatea a fost maximizată în limita puterilor mele limitate de nevoia pentru control; în cazul în care utilizatorul dorește să aibă un impact mai ridicat asupra programului, are la dispoziție uneltele necesare, însă acest lucru a necesitat un compromis: procesul de descriere a unui obiect poate deveni complex.

## **Aspect**

Sunt de părere că framework-ul meu este mai accesibil decât API-ul OpenGL, și mai presus de aceasta, alegerea mea de a mă folosi de capacitățile moderne ale limbajului de programare C (mai exact, inițializarea structurilor la declarare și faptul că valoarea lor implicită este 0) a dus la un aspect plăcut al codului de unde rezultă ușurința de a-l citi și de a folosi biblioteca. După cum se poate observa la începutul bibliotecii, dependența de structuri este evidentă. Gruparea datelor în acest fel și folosirea unei singure structuri ca parametru pentru funcții este, în opinia mea, o soluție mult mai elegantă când vine vorba de un număr mare de opțiuni.

# **Limbaje de implementare**

Limbajele de implementare folosite sunt C și GLSL (OpenGL Shading Language).

C este un limbaj de programare relativ minimalist ce a fost dezvoltat la începutul anilor 1970 de Ken Thompson și Dennis Ritchie. Acesta lucrează la un nivel mic, fiind foarte apropiat de hardware și se aseamănă cel mai mult dintre toate limbajele de programare cu Assembly. Am ales să folosesc acest limbaj deoarece este limbajul care îmi oferă cel mai ridicat nivel de control asupra aplicației din toate punctele de vedere. Una din abilitățile limbajului C deja menționată de mine și necunoscută de multe persoane este inițializarea unei structuri în momentul declarării și valoarea implicită 0 a acesteia, lucru folosit în cadrul aplicației mele.



GLSL este o componentă obligatorie când vine vorba de aplicații grafice bazate pe API-ul de la OpenGL deoarece acest limbaj a preluat structura C-ului și a dus-o într-o altă direcție, adăugând elemente necesare descrierii unui shader și nu numai. Exemple de noi cuvinte cheie des întâlnite sunt: „in”, „out”, „uniform”, „vec3”, „layout”, „sampler2D”. Elementul de surpriză în folosirea acestul limbaj este modul de lucru cu vectorii. De exemplu, un vector 4D poate fi definit astfel:



Acest limbaj are diverse alte detalii și scurtături peste care nu voi trece însă o altă tehnică ce merită exemplificată este „swizzling-ul”; dacă privim un vector ca pe o structură cu membrii (x, y, z, w), (r, g, b, a) sau (s, t, p, q) putem extrage acești membri în următorul mod:



# **Lista dinamică**

## **Structură și inițializare**

Modelul de listă dinamică pe care l-am ales prezervă tipul static de dată al declarației. Inițializarea unei liste dinamice se face prin crearea unui „header” de 16 bytes în fața blocului de memorie alocat listei dinamice propriu-zise, care conține două valori: cantitatea de memorie ocupată în HEAP și cantitatea de memorie folosită din totalul de memorie alocată. În momentul în care nu mai există spațiu, blocul de memorie este realocat și spațiul este dublat.



Deși apelul către funcția de inițializare este complex (datorită primului argument, un pointer către un alt pointer), toate apelurile de inițializare se întâmplă în spatele cortinei în cazul utilizării standard a aplicației, deoarece acesta este locul pentru care lista a fost concepută. Datorită schimbării locației în memorie, este singura soliție. Utilizatorul poate adresa valorile listei în modul clasic ( lista[i]), iar accesul la „header” se face prin macro-uri ajutătoare:



De asemenea, pentru a preveni inițializări nedorite, lista trebuie declarată inițial drept un pointer către NULL de tipul de dată dorit. Parametrul *value\_size* reprezintă mărimea în bytes al unui element, pentru ca listei să i se alocheze această dimensiune (astfel, începe cu spațiul necesar pentru un singur obiect).

## **Utilizare și curățare**

Utilizarea acestei liste se face după modelul *push & pop*. Cea mai simplă metodă de a adăuga date unei liste este alipirea acesora la final, eliminând nevoia de a specifica un index și de a muta valori. Ametrine pune la dispoziție două macro-uri ajutătoare pentru realizarea acestor acțiuni.



Acțiunea de adăugare se desfășoară după următoarele etape: (1) se verifică dacă elementul dorit are loc în listă și marirea listei în caz negativ, (2) insearea valorii și (3) actualizarea header-ului. La ștergere lucrurile sunt mai simple; lista nu scade în mărime alocată (din motive de performanță) ci (4) actualizăm header-ul scăzând marimea elementului, prefăcându-ne că nu mai există. După cum a fost deja precizat, accesul în listă se face în stilul clasic, *lista[index]*.

Curățarea listei se poate face în două moduri: resetarea mărimii ocupate la 0 în header sau dealocarea de memorie, în funcție de nevoia aplicației și a utilizatorului. Din nou, Ametrine oferă macro-uri și funcții pentru aceasta.



# **Lista dinamică “înghesuită”**

## **Concept și structură**

Sistemul pe care am ales să-l numesc o listă dinamică “înghesuită” este o grupare de două liste dinamice despre care am vorbit mai sus, organizate astfel: prima conține elementele propriu-zise iar cea de-a doua conține indexul elementului în prima listă.



Logica sistemului se bazează pe faptul că fiecare obiect are un identificator numeric (ID) care, folosit ca indice pentru lista *indices* indică poziția valorii elementului în lista *elements.* În momentul eliminării unui obiect din listă, un alt element este mutat în acel loc. Astfel putem observa că denumirea de listă „înghesuită” provine de la această metodă de a înlătura spațiul gol în memoria alocată.

Preferința macro-urilor vine dintr-un motiv practic; prin utilizarea acestora în locul unor funcții sau proceduri, tipul de dată al „parametrului” este prezervat.



# **I/O al sistemului**

## **Manager de evenimente**

Datorită necesității cross platform, manager-ul de evenimente are două implementări diferite, una pentru Windows și una pentru X11 (GNU/Linux), care sunt abstractizate în spatele unei structuri identice. Procedurile diferă din cauza structurii diferite ale API-ului de comunicare cu sistemul, însă principiul este același: pentru fiecare actualizare, se iterează prin lista de evenimente ale sistemului iar la fiecare pas este aplicat un filtru de interpretare; dacă evenimentul corespunde cu unul dintre filtre, datele sale sunt trimise procedurilor Ametrine, iar altfel, evenimentul este dat sistemului. De exemplu, WM\_KEYDOWN reprezintă codul evenimentului apăsării unei taste, iar manager-ul de evenimente filtrează acest cod.



Evenimentele care mă interesează sunt cele legate de apăsarea/ridicarea unei taste, apăsarea/ridicarea unui buton al mouse-ului, mișcarea mouse-ului, mișcarea/redimensionarea ferestrei și apăsarea butonului de închidere al ferestrei. În capitolele următoare voi intra în mai mult detaliu asupra acestui aspect, însă ferestrele sunt structurate sub formă de arbore, fiecare având un părinte. În momentul recepționării unui eveniment de închidere a unei ferestre, manager-ul verifică dacă aceasta este ultima fereastră „principală” (fără părinte), și oprește aplicația în acest caz.



## **Tastatura**



Procesarea evenimentelor tastaturii rezultă în actualizarea unui LUT, care este mai apoi accesat când utilizatorul apelează una dintre funcțiile de verificare a tastelor. Acest LUT este creat la inițializarea aplicației și variază în funcție de sistemul de operare, iar celelalte două liste reprezintă starea fiecărei taste înregistrate. Deoarece uneori este nevoie ca un buton să fie apăsat în permanență, am ales să folosesc două liste, iar verificarea se face prin compararea stării din frame-ul curent (*map*) cu starea din frame-ul precedent (*prev\_map*).



Sistemul a fost ales în special pentru simplicitatea acestuia și sunt puse la dispoziția utilizatorului mai multe funcții pentru verificarea stării tastelor, atât prin compararea între frame-uri cât și verificarea doar a frame-ului curent. Pentru vizibilitate și comoditate, am ales să creez un *enum, am\_key\_map* cu toate tastele pe care biblioteca le suportă, utilizatorul fiind astfel scutit din a căuta codurile acestora prin manualele sistemului de operare sau alte biblioteci standard (de exemplu, AM\_KEYCODE\_F1, AM\_KEYCODE\_W).

## **Mouse**



În comparație cu tastatura, numărul mic de butoane al mouse-ului îl scutesc de un LUT, însă acesta este o componentă mai complexă, deoarece, pe lângă starea butoanelor simple (up/down), apar poziția acestuia pe ecran și rotița. În rest, actualizarea listelor și a celorlalte date se face ca la tastatură, prin manager. De asemenea Ametrine oferă posibilitatea de a bloca mouse-ul, actualizându-i poziția la fiecare frame pentru a-l centra în fereastră. Acest mod de lucru este util pentru aplicațiile care necesită mișcări complexe și lungi ale mouse-ului, precum o aplicație 3D unde camera este controlată de mouse.



## **Proceduri callback**

În bucățile de cod din secțiunea manager-ului pot fi observate proceduri „callback”. Când vine vorba de input, Ametrine nu are un set permanent de proceduri care să interpreteze evenimentele, ci în schimb depozitează niște pointeri către proceduri, formând astfel sistemul de callback-uri. Ametrine vine cu un set de proceduri predefinite care nu fac nimic altceva decât să interpreteze un eveniment și să actualizeze datele interne, însă acestea pot fi înlocuite de către utilizator oricând, deoarece apelul către funcție se face printr-un pointer.



Astfel, un utilizator poate implementa propria sa funcție care acționează ca un „hook”. După cum se poate observa în procedura exemplu de mai sus, nu toți parametrii sunt folosiți de implementarea predefinită, însă aceștia există, în cazul în care o nouă implementare a procedurii are nevoie de ei. Un exemplu practic pentru o astfel de procedură poate fi numărarea tastelor apăsate într-un interval de timp.

## **Sistemul de ferestre**

Sistemul de management al ferestrelor a fost descris pe scurt în capitolele anterioare. Ferestrele urmează o ierarhie în formă de arbore, fiecare având o fereastră părinte. „Nodul 0”, sau punctul care reprezintă lipsa unui părinte variază în funcție de sistem însă este definit sub formă de macro. Acest sistem permite crearea unei ferestre în cadrul altei ferestre, alcătuind o structură de fereastră în fereastră. Manager-ul de evenimente recepționează redimensionarea și mutarea oricărei ferestre definite, iar depozitarea ferestrelor este primul exemplu practic al listei dinamice „înghesuite”. Voi prezenta modul în care un obiect este creat aici, însă acesta este valabil pentru orice obiect înregistrabil, nu doar pentru ferestre.



Aici putem observa și crearea unui „GL context”, care este o structură specifică pentru OpenGL, care înregistrează fereastra ca un obiect ce urmează să folosească API-ul OpenGL. De asemenea, *gladLoadGL()* este o funcție ce aparține bibliotecii GLAD, al cărui scop este încărcarea DLL-urilor OpenGL, pentru a înlătura nevoia de a face acest lucru manual.

Din nou, un un lucru care se aplică la toate funcțiile ce creează un obiect este parametrul. Am ales să utilizez limbajul C sa capacitatea sa maximă, folosindu-mă de structuri, despre a căror inițializare am discutat deja. Astfel, parametrii funcției se transformă într-o frumoasă listă ce poate fi refolosită oricând.



Nespecificând poziția ferestrei, aceasta va lua valoarea (0, 0) datorită modului de interpretare al limbajului, iar astfel am încercat să creez biblioteca având ca scop încurajarea specificării numai datelor importante, lăsând motorul grafic să se ocupe de restul.

Ametrine pune la dispoziție diverse funcții și proceduri care ajută la gestionarea manuală a ferestrelor, dacă este cazul. Deși puține la număr, acestea sunt suficiente pentru a avea un control deplin asupra oricărei ferestre.



# **Implementarea OpenGL**

## **Shadere**

După cum am menționat deja, am ales să fac o abstracție de la denumirile OpenGL clasice. API-ul numește un „shader” un fișier care descrie o parte din procesul prelucrării datelor, iar un „shader program” reprezintă totalitatea shaderelor care descriu modul în care datele sunt procesate și afișate pe ecran. Am ales să denumesc un „shader program” doar „shader” deoarece în industrie și pe internet, acești termeni sunt interschimbabili.

Înregistrarea unui shader se face în același stil ca și la ferestre: o funcție primește ca parametru un *struct* descriptiv și returnează un identificator. Ametrine oferă posibilitatea creării shaderelor de tip *vertex*, *geometry*, *compute*, *fragment*.



Păstrarea identificatorului la creare nu este obligatorie însă este recomandată.

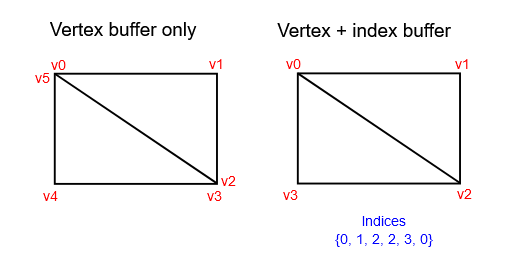
## **Buffere**

Conceptul de „buffer” în OpenGL este unul cât se poate de simplu, fiind o listă cu spațiu rezervat în memoria grafică. În funcție de modul de utilizare, driver-ul plăcii video optimizează poziția acestuia; de exemplu, dacă dorim să avem un buffer care este actualizat în fiecare secundă, acesta va avea prioritate și va fi poziționat cât mai în față, pentru ca accesul la acesta să fie mai rapid. În momentul creării, trebuie să specificăm și modul de interpretare al acestuia, prin tipul de dată.

Ametrine oferă structuri și funcții separate pentru fiecare tip de buffer pentru înlăturarea posibilelor greșeli de implementare, deoarece OpenGL pune la dispoziție un număr considerabil de tipuri de buffer. Am ales să implementez următoarele buffere: vertex, index, storage, frame.

Un vertex buffer poate să conțină orice fel de date dorim, fiind cel mai des folosit pentru specificarea poziției în spațiu al unui obiect, culoarea în fiecare punct al acestuia, coordonate UV pentru texturi.

Un index buffer reprezintă un pas de optimizare al memoriei, permițându-ne să specificăm sub formă de listă *vertex*-urile figurilor geometrice, înlăturând astfel repetiția acestora. În exemplul de mai jos, dreptunghiul din dreapta are lista de indici   
{ 0, 1, 2, 2, 3, 0}. Astfel, am reușit să stocăm doar 4 vertices, în loc de 6.



Un storage buffer este o componentă necesară lucrului cu shaderele de tip compute, dar nu numai. Avantajul acestora este accesul atomic (folosindu-se de așa numitele „bariere” care blochează accesul până când procesul curent este terminat) și faptul că nu este nevoie să îi definim o mărime la creare.

Un frame buffer poate fi văzut ca o textură. Acesta pastrează datele unei imagini (intermediare sau finale) rezultate în urma aplicării shaderelor. De exemplu, imaginea de pe ecran este de fapt un frame buffer care conține valorile pentru fiecare pixel.

## **Uniforme (variabile globale)**

Un „uniform” este o variabilă globală a unui shader. Acestea pot fi văzute ca niște parametri ai shader-ului. Acestea intervin când avem nevoie să transmitem cantități mici de informație shaderului, pentru care un buffer este inutil. De exemplu, poziția camerei în spațiu sau matricea de transformare a unui obiect, în cazul în care aplicația conține un obiect sau mai multe obiecte care suferă aceeași transformare.

## **Texturi**

O textură este o imagine încărcată în memoria video a plăcii grafice. Crearea unei texturi este migăloasă deoarece aceasta necesită multe informații despre modul în care va fi utilizată și modul de interpretare a acesteia. Pentru început, Trebuie să știm structura fișierului (RGBA8 cu câte 8 biți pentru fiecare culoare și transparență, RGB32F, cu care 32 de biți pentru fiecare culoare însă sub formă de *floating point number* și nu integer).



O altă componentă importantă este modul în care tratăm textura în afara coodronatelor sale. Acest mod este specificat atât pentru axa verticală cât și pentru axa orizontală, oferindu-ne flexibilitate.



Ultimul detaliu important este modul în care tratăm pixelii imaginii. Aici avem două posibilități, NEAREST, care repezintă imaginea în sine, pixel cu pixel, și LINEAR care aproximează culoarea unui pixel în funcție de vecinii săi pentru a evita o imagine pixelată.

De asemenea, există și *mipmap*-uri, care sunt copii ale imaginii la rezoluții mai mici, iar acestea sunt alese automat de OpenGL în funcție de distanța camerei până la textură.

Ajungem astfel la *struct*-ul descriptiv al unei texturi. Acesta este voluminos însă Ametrine oferă valori „default” pentru toate elementele acestuia, fiind nevoie ca utilizatorul să specifice un anumit câmp doar când este necesar.



Câmpul *path* reprezintă calea către un fișier de tip imagine. Dacă acest câmp este specificat, imaginea va fi încărcată în memorie cu ajutorul bibliotecii *stb\_image.h* iar câmpurile width și height vor fi completate automat. Mai jos sunt prezentate două metode de creare a unei texturi.



Luând în considerare că majoritatea imaginilor sunt structurate ca RGBA8 (8 biți pentru fiecare culoare), crearea unei texturi devine trivială, întrucât majoritatea câmpurilor vor fi completate automat de către Ametrine.

## **Pipeline**

Un pipeline este o abstractizare pe care o face Ametrine pentru a descrie modul în care un anumit tip de obiecte este procesat. Este unul dintre elementele cele mai complexe prezente în această biblioteca. Pentru început, conține detalii despre „blending”, adică modul în care culorile se amestecă unele cu altele (important pentru transparență).

Blending-ul este descris printr-o funcție și două moduri (pornire și final), iar pe baza acestora, OpenGL deternimă modul de combinare al culorilor.



*Depth testing*-ul reprezintă un test pentru a determina dacă un obiect este în fața sau în spatele altuia. Acest test este necesar deoarece dacă procesam obiectele din spatele scenei după ce le procesăm pe cele din față, acestea se vor suprapune. Este descris de o singură funcție care specifică semnul comparării valorilor adâncimii.



*Stencil testing*-ul reprezintă un test care se face la finalul procesării imaginii și care elimină anumiți pixeli în funcție de o textură „mască” pe care o putem specifica. De asemenea, putem seta diverse operații pentru fiecare rezultat al testului.

*Face culling*-ul este un test care ia in considerare ordinea în care o primitivă este procesată (sens trigonometric sau invers trigonometric). În funcție de poziția camerei, un obiect care este „cu fața” la noi va fi în sens trigonometric, iar unul cu spatele în sens invers trigonometric. Astfel, dacă un obiect este cu spatele, vom sări peste procesarea acestuia, deoarece nu va fi vizibil pe ecran.



Pe lângă aceste teste, un pipeline mai conține informații despre shader-ul pe care îl folosește, tipul de primitivă (triunghi, linie, punct), etc.



## **Render pass**

Un „render pass” reprezintă totalitatea pașilor aplicați unui frame buffer. Aici pot exista mai multe aplicări de shadere, diverse actualizări de date, dar în final mereu va rezulta un frame buffer, care poate fi folosit ca dată de intrare pentru un nou render pass, de exemplu, acesta fiind o oarecare textură. Aici putem specifica „măști” pentru *depth testing* si *stencil testing* precum și frame buffer-ul cu care vom lucra.



# **Structură și modul de lucru**

## **Cerințe obligatorii**

Pentru ca biblioteca să funcționeze, este nevoie de declararea a 3 proceduri: procedura de inițializare (este apelată o singură dată, la începutul aplicației, ideală pentru crearea obiectelor și procesarea datelor), procedura de actualizare (este apelată la fiecare actualizare, ideală pentru desenatul pe ecran) și procedura de închidere (este apelată după ce aplicația primește semnalul de terminare, ideală pentru curățarea memoriei alocate în timpul execuției). De asemenea, este nevoie de o descriere generală a aplicației, urmată de un *while loop*.



## **Desenarea pe ecran**

Deși inițializarea datelor și actualizarea acestora poate fi făcută în orice fel dorit, desenarea pe ecran necesită urmarea unei anumite structuri bine definite de către Ametrine.



Pentru început, trebuie să marcăm începutul unui render pass, după care vom seta un pipeline pentru a fi folosit, iar mai apoi vom seta descrierea datelor propriu-zise, ca într-un final să apelăm funcția de desenare. Opțional, vom curăța ecranul la fiecare actualizare pentru a evita efecte nedorite. Deși procesul este migălos, decizia a fost luată intenționat deoarece se aliniază cu standardul motoarelor grafice folosite în industrie.

Pentru siguranță, Ametrine utilizează un „frame cache”, ceea ce înseamnă ca datele sunt mai întâi copiate într-un loc separat înainte de a fi folosite, pentru a preveni posibile modificări nedorite pe parcursul execuției. Momentan acest lucru nu este o problemă dar în viitoare versiuni, unde multi-threadingul este o posibilitate, mă va scuti de un efort.



# **OBJ Loader**

## **Structură**

Datele unui obiect pot fi scrise de mână dar acest lucru este valabil doar pentru obiecte simple, precum cuburi, figuri bidimensionale. Pentru obiecte complexe, este nevoie de ceva mai puternic, iar fișierele .obj sunt simple de înțeles și des folosite în industrie.

Structura unui fișier .obj nu este foarte complexă, însă eu am ales să nu implementez toate caracteristicile regăsite în structura fișierului, optând pentru folosirea modulelor Ametrine. Astfel, elementele de structură ale fișierului pe care am ales să le interpretez sunt:



## **Implementare**

Citirea și stocarea datelor nu este o problemă dificilă, având în vedere structura bine definită a fișierului; implementarea mea citește caracter cu caracter, scanând pentru caracterele *v* și *f*. În cazul în care *f* este urmat de un spațiu, procesez indicii. Pentru *v*, se ia pe cazuri, și se adaugă în liste dinamice. Problema apare însă în momentul în care un vertex (care se pot repeta în enumerarea fețelor) are simultan mai mult de o coordonată UV sau o normală. OpenGL nu permite așa ceva, iar astfel trebuie să fie create noi vertices în aceeași poziție pentru a putea procesa corect obiectul. Interpretările pentru *v* sunt banale, astfel ma voi concentra pe *f*.





Procesul de creare a noi vertices este realizat recursiv și ilustrat în bucata de cod de mai sus. Dacă sunt 100% identici, se crează doar un indice nou. Altfel, se merge în jos pe o listă înlănțuită până se ajunge la nodul inițial; la acesta, se crează un nou vertex cu caracteristicile inițiale, nereușind să găsim unul 100% identic cu acesta.

Datorită acestui sistem, Ametrine este capabil de a lucra cu obiecte complexe, iar împreună cu restul sistemelor oferite, poate fi folosit pentru orice necesită o vizualizare grafică. Biblioteca este una simplă, nu este menită pentru performanță, dar dacă respectăm standarde ale programării grafice eficiente si optimizate, putem împinge performanțele bibliotecii la limite surprinzătoare.

# **Pământul**

## **Detalii**

Pentru a exemplifica și a demonstra capacitățile bibliotecii mele, am ales sa creez o mică aplicație 3D. Am reprezentat un model al Pământului, încercând să folosesc cât mai multe dintre dotările Ametrine. Așadar, în paragrafele de mai jos, voi detalia procesul de creație.

Pentru început, am avut nevoie de o sferă. Deși Pământul nu este o sferă exactă, o aproximație este suficientă pentru această demonstrație. Cea mai simplă metodă pentru crearea unei sfere este normalizarea unui cub, procedeu ales și de mine. Pentru viitoare optimizări, algoritmul meu permite crearea unui număr specificat de sub-fețe pentru fiecare față a cubului.

Problema apare atunci când dorim să înfășurăm o foaie în jurul unei sfere. Știm din studiul geometriei că acest procedeu este complicat, însă cu ajutorul funcțiilor trigonometrice, putem crea un algoritm pentru a lua punctele imaginii și a le aplica sferei.



După cum știm, suprafața planetei nu este plană; am ales să folosesc conceptul de „textură de înălțime”, care este o textură alb-negru, reprezentând înălțimile fiecărui punct, în intervalul [0, 255]. Folosind aceasta, pot scoate punctele sferei în exterior, creând astfel un relief realistic.

Culorile maselor de pământ sunt aplicate dintr-o textură, însă oceanul, pentru a părea mai viu și realist, este supus unui proces mai complicat. Folosesc o hartă de înălțime pentru zonele unde adâncimea apei este mică. La această înălțime, adaug cantitatea de clorofilă înregistrată în apele oceanului, iar mai apoi interpolez între culori pentru apele adânci și apele mici. Astfel, oceanul capătă nuanțe diferite și atrăgătoare. Pentru efectul de valuri, am folosit un procedeu numit *triplanar mapping*, procedeu care ia o textură și o aplică unui obiect pe fiecare axă spațială (X, Y, Z), interpolând imaginea la intersecții, pentru a crea un efect de continuitate a texturii.

Din cauza limitării de procesare, nu putem avea nu număr mare de vertices, iar efectele de lumină suferă. Lumina se calculează la fiecare punct de normală, însă o normală există la fiecare vertex. Pentru a combata această problemă, am folosit conceptul de *normal mapping*, prin care, folosindu-mă de o imagine, atribui o normală fiecărui pixel, crescând semnificativ detaliul obținut prin luminarea obiectului.

Pentru o nuanță de realism, am ales să implementez o simplă sferă care reprezintă Soarele, acesta fiind singura sursă de lumină a scenei.

# **Bibliografie**

1. UV Mapping   
   <https://en.wikipedia.org/wiki/UV_mapping>
2. Sas Luca – Modern C  
   <https://youtu.be/QpAhX-gsHMs>
3. Serrano Harold – How to Implement the Rendering System  
   <https://youtu.be/oyXRE809geY>
4. nlguillemot - Scene Structure  
   <https://nlguillemot.wordpress.com/2016/11/18/opengl-renderer-design/>
5. Niklas – ID Lookup Table  
   <http://bitsquid.blogspot.com/2011/09/managing-decoupling-part-4-id-lookup.html>
6. Microsoft – Creating a Window  
   <https://docs.microsoft.com/en-us/windows/win32/learnwin32/creating-a-window>
7. Ben Golus – Triplanar Normal Mapping  
   <https://bgolus.medium.com/normal-mapping-for-a-triplanar-shader-10bf39dca05a>
8. NASA Blue Marble – Imagini ale Pământului și Hărți <https://earthobservatory.nasa.gov/features/BlueMarble>,  
   https://earthobservatory.nasa.gov/global-maps/MY1DMM\_CHLORA
9. Echipa STB – stb\_image.h  
   <https://github.com/nothings/stb/blob/master/stb_image.h>
10. dav1dde – glad.h  
    <https://glad.dav1d.de/>
11. g-truc – Matematică  
    <https://github.com/g-truc/glm>
12. ThinMatrix – Game Tutorial Episoadele 9 & 16  
    <https://youtu.be/KMWUjNE0fYI>  
    <https://youtu.be/qslBNLeSPUc>
13. Phil Nowell – Optimizarea Topologiei Unui Cub Normalizat în Sferă  
    http://mathproofs.blogspot.com/2005/07/mapping-cube-to-sphere.html
14. Indie Game Engine Devs – Comunitate pe Discord
15. r/opengl  
    https://www.reddit.com/r/opengl/