Laboratorul 09

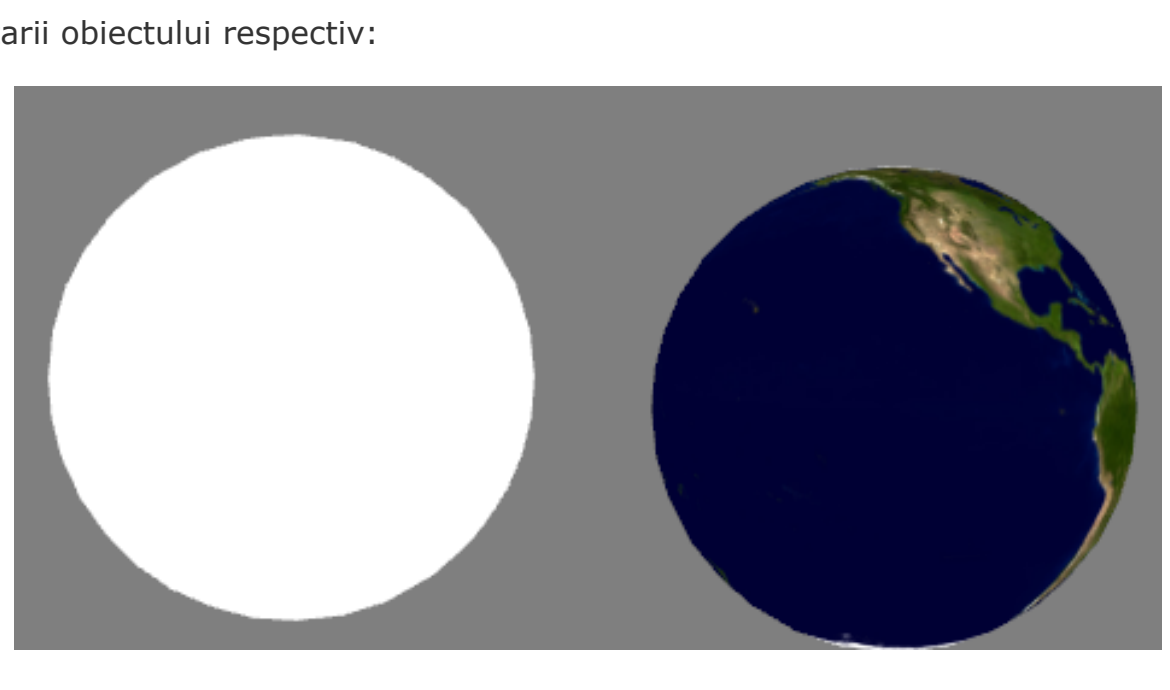
Video Laborator 9: https://youtu.be/YS262bweU2Q
Autor: Bogdan Teacă

Introducere

Am invatat in laboratoarele trecute ca, pentru a adauga detalii obiectului nostru, putem folosi culori asociate fiecarui vertex. Pentru a putea crea obiecte detaliate, asta ar insemna sa avem foarte multi vertecsi astfel incat sa putem specifica o gama cat mai variata de culori si ar ocupa foarte multa memorie. Pentru a rezolva aceasta problema se folosesc **texturi**.

Pentru scopul acestui laborator, o textura este o **imagine 2D** (exista si texturi 1D si 3D) care este folosita pentru a adauga detalii obiectului. Ganditi-va la textura ca la o bucata de hartie (cu un desen pe ea) care este impaturita peste obiectul 3D. Pentru ca putem adauga oricate detalii vrem intr-o singura imagine, putem da iluzia ca obiectul este foarte detaliat fara sa adaugam vertecsi in plus.

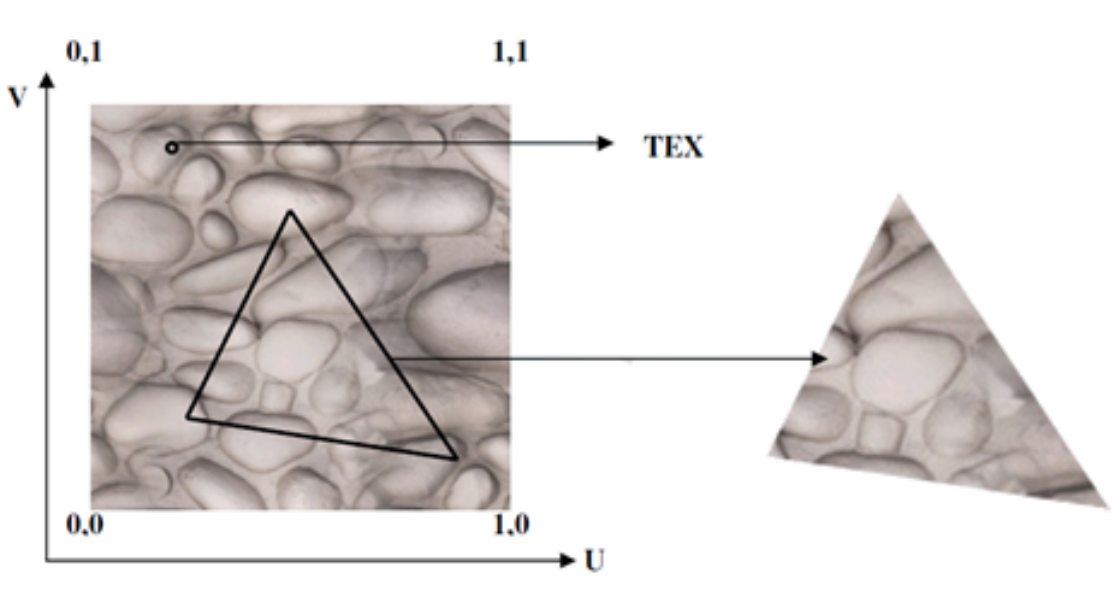
Diferenta intre un obiect texturat si acelasi obiect netexturat este remarcabil de mare din punct de vedere al acuratetei reprezentarii obiectului respectiv:



Maparea texturilor

Pentru a mapa (impacheta) o textura peste un triunghi, trebuie sa specificam in ce parte din textura corespunde fiecare vertex. Asadar, fiecare vertex ar trebui sa aiba asociat un set de coordonate de textura (2D adica `glm::vec2`) care specifica partea din textura unde isi are locul. Coordonatele de textura se afla in intervalul $[0,1]$ pentru axele x si y (in cazul 2D). Coordonatele texturii incep din punctul $(0,0)$ pentru coltul din stanga jos a imaginii pana la punctul $(1,1)$ care se afla in coltul din dreapta sus.

Un punct de pe imagine si care este in spatiul $[0,1] \times [0,1]$ se numeste **texel**, numele venind de la *texture element*. Dupa cum se poate vedea in imaginea de mai jos, in functie de coordonatele fiecarui vertex al triunghiului, partea din textura care este mapata peste triunghi poate fi diferita:



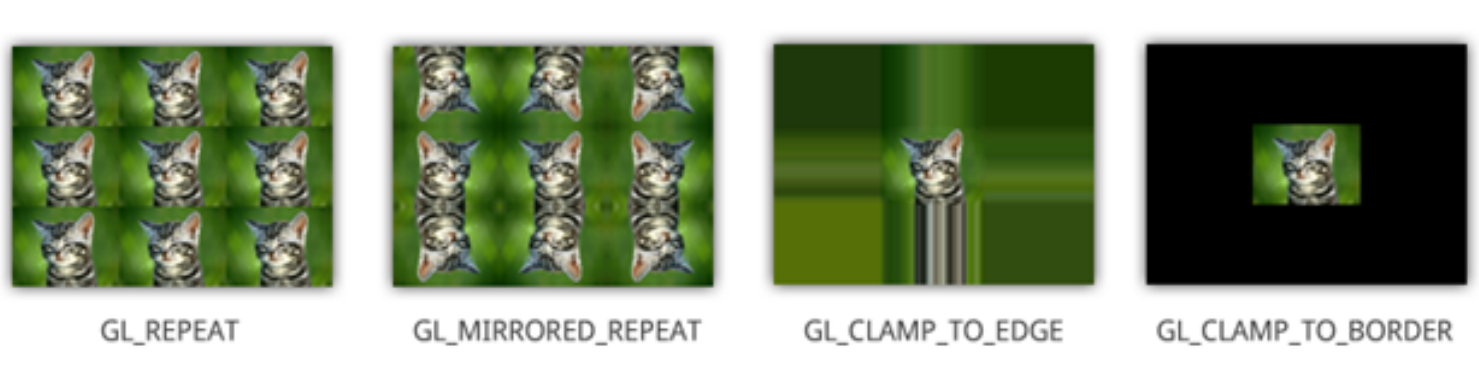
Adaugarea unei texturi

Pentru a construi o textura in OpenGL avem nevoie in primul rand de pixelii imaginii ce va fi folosita ca textura. Pixelii trebuie fie generati functional, fie incarcati dintr-o imagine, iar acest pas este independent de OpenGL. In laborator, pentru a citi texturi, se poate folosi clasa `Texture2D`:

```
Texture2D::Load2D(const char* fileName, GLenum wrapping_mode)
```

unde `wrapping_mode` poate fi:

- `GL_REPEAT`: textura se repeta pe toata suprafata obiectului
- `GL_MIRRORED_REPEAT`: textura se repeta dar va fi vazuta in oglinda pentru repetarile impare
- `GL_CLAMP_TO_EDGE`: coordonatele vor fi intre 0 si 1
- `GL_CLAMP_TO_BORDER`: asemanator cu clamp to edge, doar ca ceea ce se afla dincolo de marginea imaginii nu mai este texturat.


GL_REPEAT GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER

Pentru a seta manual modul de wrapping al texturii se pot folosi:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrapping_mode); // modul de wrapping pe orizontala
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrapping_mode); // modul de wrapping pe verticala
```

Dupa ce avem pixelii imaginii incarcate putem genera un obiect de tip textura de OpenGL folosind comanda:

```
unsigned int gl_texture_object;

glGenTextures(1, &gl_texture_object);
```

Similar cu toate celelalte procese din OpenGL, nu lucram direct cu textura ci trebuie sa o asociem unui punct de legare. Mai mult, la randul lor, punctele de legare pentru texturi sunt dependente de unitatile de texturare. O unitate de texturare e foarte similara ca si concept cu pipe-urile pe care trimitem atribute. Setam unitatea de texturare folosind comanda (o singura unitate de texturare poate fi activa):

```
glActiveTexture(GL_TEXTURE0 + nr_unitatii_de_texturare_dorite);
```

Iar pentru a lega obiectul de tip textura generat anterior la unitatea de textura activa folosim punctul de legare `GL_TEXTURE_2D`:

```
glBindTexture(GL_TEXTURE_2D, gl_texture_object);
```

Pentru a incarca datele efective in textura folosim comanda:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

- Primul argument specifica tipul de textura. Daca punem `GL_TEXTURE_2D`, inseamna ca aceasta functie va asocia obiectului de tip textura (trecut anterior prin `bind`) o textura 2D (deci daca avem legate un `GL_TEXTURE_1D` sau `GL_TEXTURE_3D`, acestea nu vor fi afectate).
- Al 2-lea argument specifica nivelul de mipmap pentru care vrem sa cream imaginea. Vom explica ce este un mipmap pe parcusul laboratorului. Pentru moment, putem sa lasam valoarea aceasta 0.
- Al 3-lea argument specifica formatul in care vrem sa fie stocata imaginea. In cazul nostru este RGB.
- Al 4-lea si al 5-lea argument seteaza marimea imaginii.
- Urmatorul argument ar trebui sa fie mereu 0 (legacy stuff)
- Argumentele 7 si 8 specifica formatul si tipul de date al imaginii sursa.
- Ultimul argument il reprezinta vectorul de date al imaginii.

Asadar: `glTextImage2D` incarca o imagine definita prin datele efective, adica un array de `unsigned char`s, pe obiectul de tip textura legat la punctul de legare `GL_TEXTURE_2D` al unitatii de texturare active la momentul curent, nivelul 0 (o sa luam aceasta constanta ca atare pentru moment), cu formatul intern `GL_RGB` cu lungimea width si cu inaltimea height, din formatul `GL_RGB`. Datele citite sunt de tip `GL_UNSIGNED_BYTE` (adica `unsigned char`) si sunt citite de la adresa data.

Pentru a va cimenta si mai mult aceste notiuni, gasiti detalii si exemple suplimentare **in aceasta postare**.

Utilizarea texturii

Pentru a folosi o textura in shader trebuie urmat acest proces:

```
glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, texture1->GetTextureID());

glUniform1i(glGetUniformLocation(shader->program, "texture_1"), 0);

glActiveTexture(GL_TEXTURE1);

glBindTexture(GL_TEXTURE_2D, texture2->GetTextureID());

glUniform1i(glGetUniformLocation(shader->program, "texture_2"), 1);
```

Unitatea de texturare este folositoare in momentul in care vrem sa atribuim textura unei variabile uniforme din shader. Scopul acestui mecanism este de a ne permite sa folosim mai mult de 1 textura in shaderele noastre. Prin folosirea unitatilor de texturare, putem face bind la multiple texturi, atat timp cat le setam ca fiind active.

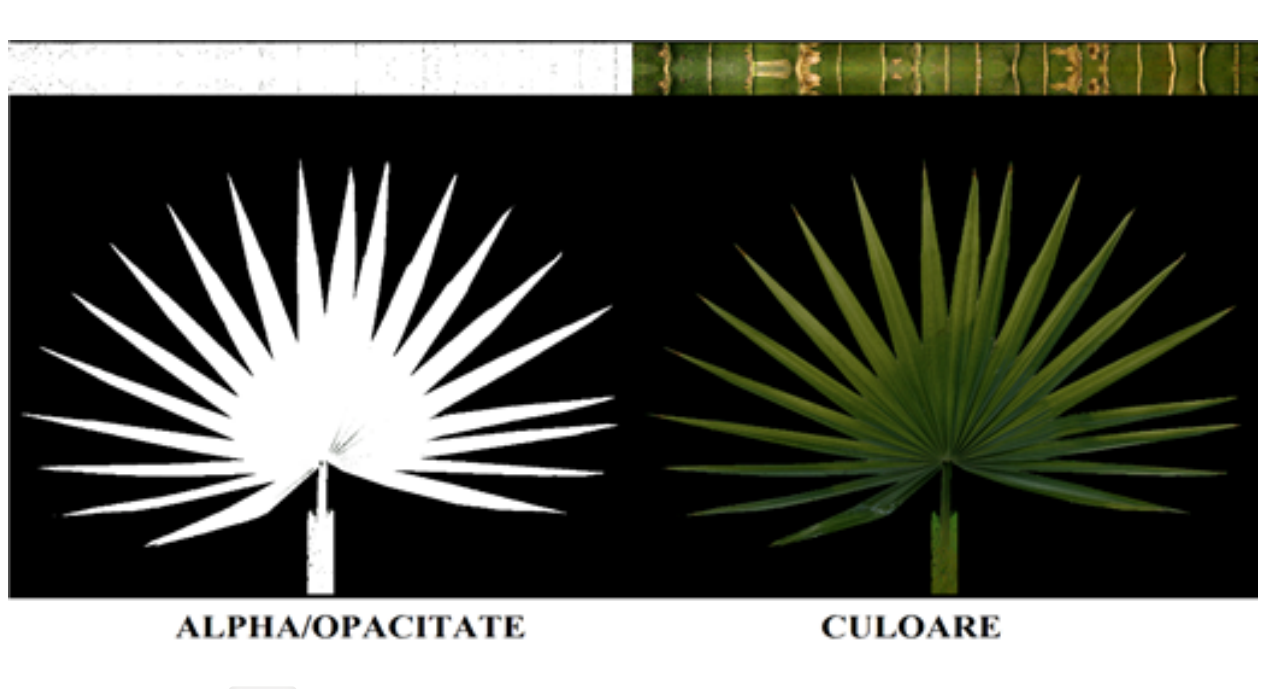
OpenGl are minim 16 unitati de texturare care pot fi activate folosind `GL_TEXTURE0` pana la `GL_TEXTURE15`. Nu este nevoie sa se specifice manual numarul, devreme ce unitatea de texturare cu numarul `X` poate fi activata folosind `GL_TEXTURE0 + X`.

Urmatorul cod este un exemplu de shader ce poate folosi legarea precedenta, unde `texcoord` reprezinta coordonatele de texturare primite ca atribute in vertex shader si apoi pasate catre rasterizer pentru interpolare:

```
#version 330

uniform sampler2D texture_1;

in vec2 texcoord;

layout(location = 0) out vec4 out_color;

void main()
{
	vec4 color = texture2D(texture_1, texcoord);
	out_color = color;
}
```

Multitexturarea este folositare in momentul in care reprezentam un obiect complex din punct de vedere topologic, de exemplu frunze, cu o textura ce are o topologie mult inferioara ca si complexitate (de exemplu un quad). Daca utilizam o asemenea reducere de complexitate, trebuie sa avem o metoda prin care sa putem elimina, la nivel de fragment, acele fragmente care nu sunt necesare (evidentiate in imaginea urmatoare).

Pentru aceasta, putem sa folosim o textura de opacitate (alpha) care ne spune care sunt fragmentele reale, vizibile, ale obiectului. Combinatia de textura de opacitate si textura de culoare este suficienta pentru definirea acestui bambus:



Elementele individuale ale unui `vec4` pot fi accesate utilizand numele componentelor:
- Pentru date geometrice: `x, y, z, w`
- Pentru date legate de culori (red, green, blue, alpha): `r, g, b, a`
- Pentru date de texturare: `s, t, p, q`

Pentru a omite desenarea fragmentelor care nu sunt vizibile se foloseste directiva de shader `discard`.

```
if (alpha == 0) {
	discard;
}
```

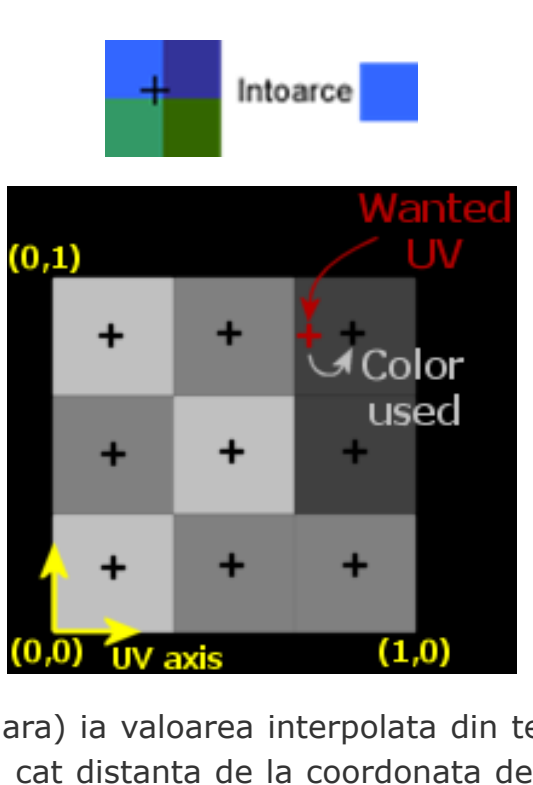
Filtrare

Coordonatele prin care se mapeaza vertecsii obiectului pe textura nu depind de rezolutia imaginii, ci sunt valori `float` in intervalul $[0,1]$, iar OpenGL trebuie sa-si dea seama ce texel (texture pixel) sa mapeze pentru coordonatele date. Pentru a rezolva aceasta problema se foloste filtrarea, care este o metoda de esantionare si reconstructie a unui semnal.

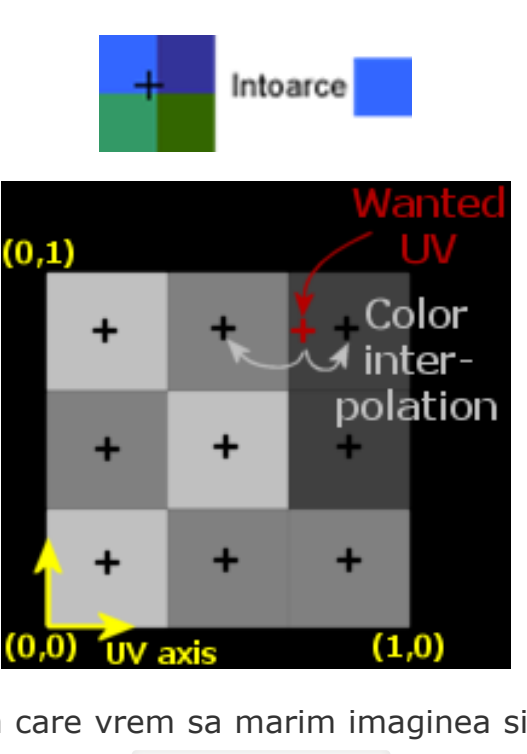
Reconstructia reprezinta procesul prin care, utilizand acesti pixeli, putem obtine valori pentru oricare din pozitiile din textura (adica nu neaparat exact la coordonatele din mijlocul pixelului, acolo unde a fost esantionata realitatea in spatiul post proiectie).

Pentru a face acest proces mai usor, OpenGL are o serie de filtre care pot fi folosite pentru a obtine maparea dorita, iar cele mai des utilizate sunt: `GL_NEAREST` si `GL_LINEAR`.

`GL_NEAREST` (care se mai numeste si *nearest neighbor filtering*) este filtrarea default pentru OpenGL. Cand este folosit acest filtru, OpenGL selecteaza pixelul al carui centru este cel mai aproape de coordonatele de texturare. Mai jos se pot vedea 4 pixeli unde crucea reprezinta exact coordonatele de texturare. Texelul din stanga sus are centrul cel mai aproape de coordonata texturii si astfel este ales:



GL_LINEAR (cunoscut drept filtrare biliniara) ia valoarea interpolata din texelii vecini ai coordonatei de texturare, aproximand astfel culoarea mai bine. Cu cat distanta de la coordonata de texturare pana la centrul texelului este mai mica, cu atat contributia culorii acelui texel este mai mare. Mai jos putem vedea cum pixelul intors



Se pot folosi filtre diferite pentru cazul in care vrem sa marim imaginea si cand vrem sa o micsoram (*upscaling* si *downscaling*). Filtrul se specifica folosind metoda `glTexParameteri`:

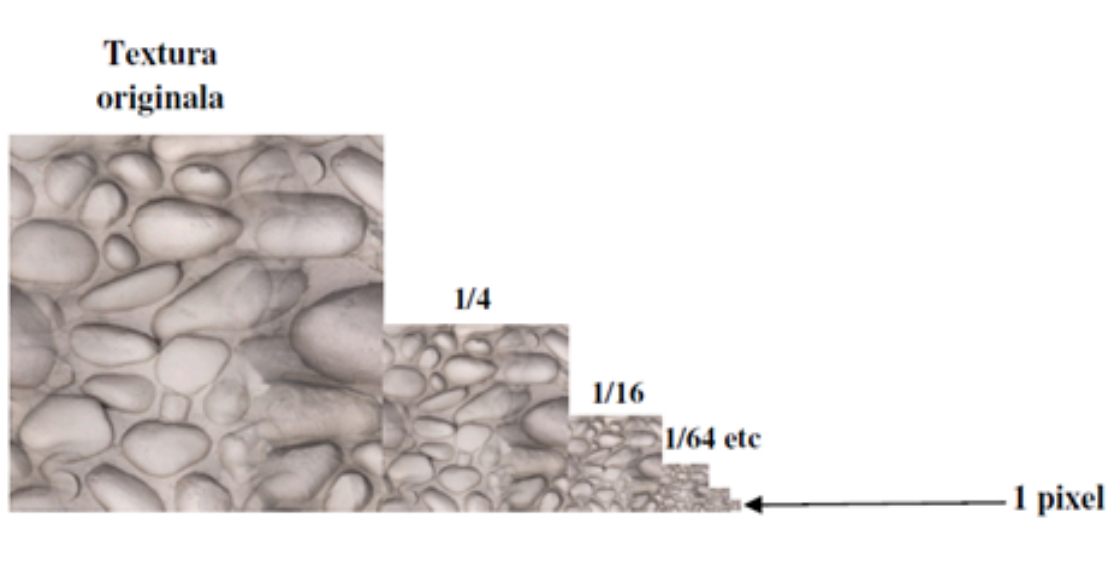
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Mipmaps

Imaginati-va cazul in care avem o camera plina cu obiecte ce folosesc aceeasi textura dar se afla in pozitii diferite. Cele care sunt mai indepartate vor aparea mai mici fata de cele care sunt mai apropiate, dar toate vor avea aceeasi textura de rezolutie mare.

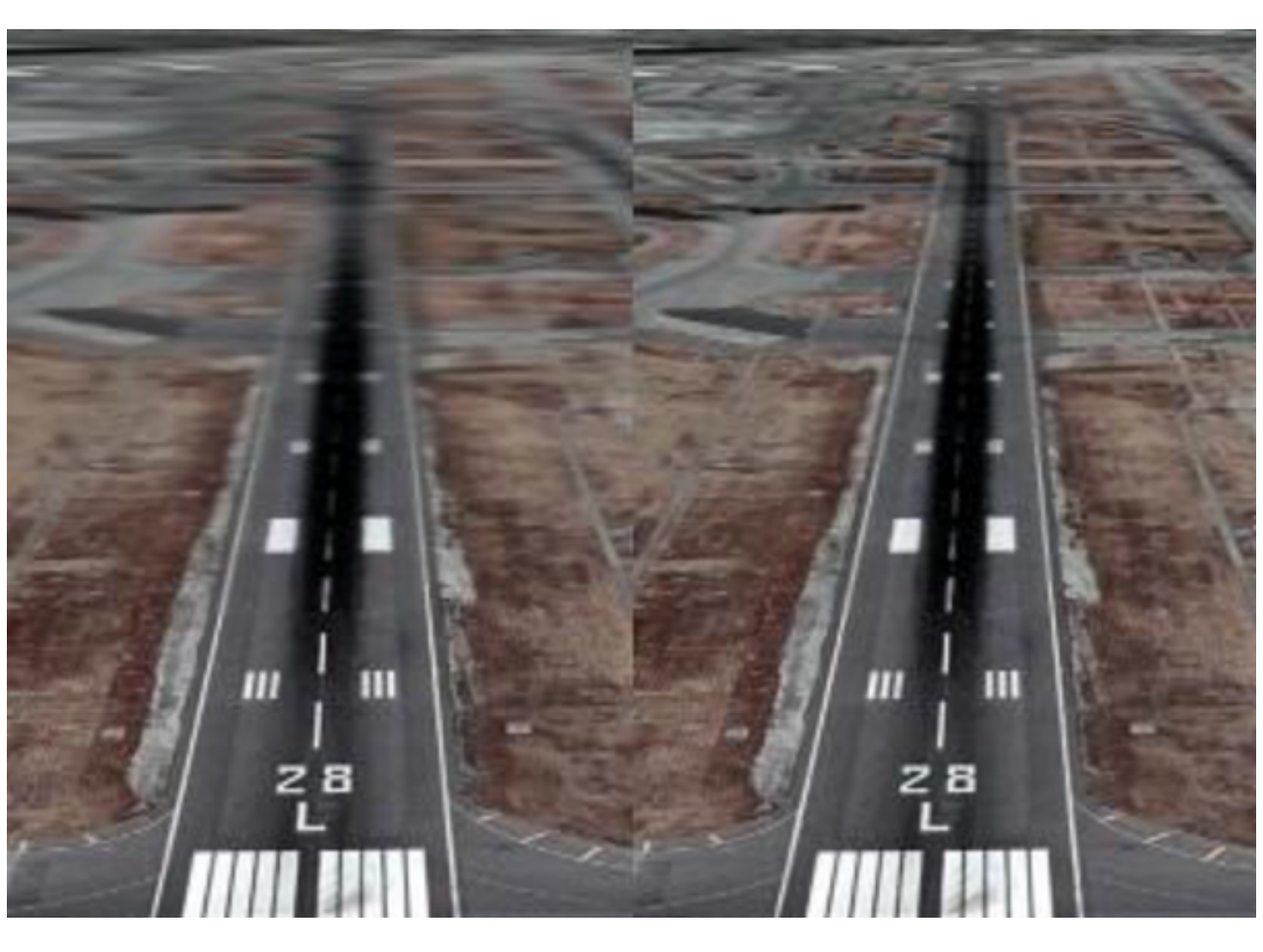
Deoarece obiectele care se afla la departare vor folosi probabil doar cateva fragmente din imaginea de baza, OpenGL va intampina dificultati in obtinerea culorii din textura de rezolutie mare deoarece trebuie sa aleaga culoarea care sa poata reprezenta o portiune foarte mare din textura. Comprimarea unei portiuni mari din textura intr-un singur frament poate duce la artefacte visible pentru obiectele mici, pe langa memoria irosita prin folosirea unei texturi mari pentru obiecte mici.

Pentru a rezolva aceasta problema, se foloseste un concept numit **mipmap**, care este de fapt o colectie de copii ale aceleiasi imagini, unde fiecare copie este de doua ori mai mica decat copia anterioara. Ideea din spatele conceptului de mipmap este destul de simpla: dupa un anumit prag de distanta, OpenGL va folosi o textura mipmap mai mica pentru acel obiect. Fiindca obiectul este la departare, faptul ca rezolutia este mai mica nu va fi observat de utilizator. O textura mipmap arata in felul urmator:



Crearea de texturi mipmaps este destul de anevoioasa de facut manual, asa ca OpenGL poate face tot acest proces in mod automat, folosind functia `glGenerateMipmap(GL_TEXTURE_2D);` dupa ce am creat textura.

Cand privim un obiect dintr-un anumit unghi, se poate ca OpenGL sa faca schimbarea intre diferite niveluri de texturi mipmap, ceea ce poate duce la artefacte asa cum se vede in imaginea de mai jos :



Exact ca filtrarea normala, este posibil sa folosim filtrare intre diferite niveluri de mipmaps folosind filtrare `NEAREST` si `LINEAR` atunci cand se produce schimbarea intre niveluri. Pentru a specifica acest tip de filtru, inlocuim filtrarea anterioara cu urmatoarele 4 optiuni :

- `GL_NEAREST_MIPMAP_NEAREST` : foloseste cea mai apropiata textura mipmap si foloseste interpolare nearest neighbor pentru a alege culoarea.
- `GL_LINEAR_MIPMAP_NEAREST` : foloseste cea mai apropiata textura mipmap si foloseste interpolare liniara pentru a obtine culoarea.
- `GL_NEAREST_MIPMAP_LINEAR` : interpoleaza liniar intre cele mai apropiate doua texturi mipmap si si foloseste interpolare nearest neighbor pentru a obtine culoarea
- `GL_LINEAR_MIPMAP_LINEAR` : interpoleaza liniar intre cele mai apropiate doua texturi mipmap si foloseste interpolare liniara pentru a obtine culoarea.

Exemplu de folosire:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

> O greseala comuna este folosirea filtrului de mipmap pentru marimea imaginii. Acest filtru nu va avea niciun efect deoarece texturile mipmap sunt folosite in principal atunci cand obiectele devin **mai mici**. Marirea texturii nu foloseste mipmap si daca dam un astfel de filtru, vom primi o eroare de tipul `GL_INVALID_ENUM`.

> Mai multe informatii si detalii despre filtrare se pot gasi pe pagina API-ului glTexParameter

Cerinte laborator

1. Descarcati framework-ul de laborator
2. Completati functia `RenderSimpleMesh` astfel incat sa trimiteti corect textura catre Shader
3. Completati coordonatele de textura pentru patrat
4. Completati shaderele astfel incat sa foloseasca coordonatele de textura
5. Sa se completeze shaderul astfel incat sa se faca alpha discard
   - In cadrul laboratorului nu se va folosi o imagine diferita pentru alpha discard ci se va testa componenta alpha din cadrul texturii
   - Ex: Faceti alpha discard daca valoarea `alpha` este mai mica de 0.5f
6. Creati si incarcati pe GPU o textura random
   - completati functia `Laborator9::CreateRandomTexture`
   - ! generati mipmaps : `glGenerateMipmap(GL_TEXTURE_2D);`
   - textura va fi folosita in cadrul randarii pe cubul din stanga
7. Modificati filterele `GL_TEXTURE_MIN_FILTER` si `GL_TEXTURE_MAG_FILTER` si observati diferentele de afisare
   - `GL_TEXTURE_MAG_FILTER` aveti doar 2 valori posibile
     - GL_LINEAR
     - GL_NEAREST
   - `GL_TEXTURE_MIN_FILTER` aveti doar 6 valori posibile
     - GL_NEAREST
     - GL_LINEAR
     - GL_NEAREST_MIPMAP_NEAREST
     - GL_LINEAR_MIPMAP_NEAREST
     - GL_NEAREST_MIPMAP_LINEAR
     - GL_LINEAR_MIPMAP_LINEAR
8. Randati un quad folosind 2 texturi. Folositi functia mix() in fragment shader pentru a obtine o combinatie intre cele doua texturi.

```
vec3 color = mix(color1, color2, 0.5f); // ultimul parametru reprezinta factorul de interpolare intre ce
```

Bonus:

1. Sa se trimita timpul aplicatiei `Engine::GetElapsedTime()` catre fragment shader si sa se utilizeze pentru a cicla prin textura de pe globul pamantesc (pe coordonata OX) (doar pentru acel obiect, deci e nevoie de si de o variabila uniform pentru a testa obiectul randat)
2. Sa se roteasca spre directia camerei (doar pe OY) quadul cu textura de iarba astfel incat sa fie orientat tot timpul catre camera.

Search
Info curs
- Elemente de Grafică pe Calculator
- Infographie

Cataloage EGC
- TBA

Laboratoare
- Laboratorul 01
- Laboratorul 02
- Laboratorul 03
- Laboratorul 04
- Laboratorul 05
- Laboratorul 06
- Laboratorul 07
- Prezentare Tema 1
- Laboratorul 08
- Laboratorul 09
- Vacanță
- Prezentare Tema 2
- Recuperări laborator
- Prezentare Tema 3
- Resurse: Redare text

Teme
- Regulament General
- Tema 1 - Bow and Arrow
- Tema 2 - Skyroads
- Tema 3 - Stylised Runner

Resurse
- Resurse Utile
- Notare

Table of Contents

- Laboratorul 09
  - Introducere
  - Maparea texturilor
  - Adaugarea unei texturi
  - Utilizarea texturii
  - Filtrare
  - Mipmaps
  - Cerinte laborator