# Culori, vârfuri, primitive grafice. Program principal și *shader*-e

Mihai-Sorin Stupariu

Sem. I, 2024 - 2025

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:
    - dimensiunea $n$ a spațiului de culori în care lucrăm, $n = 3$ (RGB) sau $n = 4$ (RGBA); A=factorul "alpha", legat de opacitate/transparență.

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:
    - dimensiunea $n$ a spațiului de culori în care lucrăm, $n = 3$ (RGB) sau $n = 4$ (RGBA); A=factorul "alpha", legat de opacitate/transparență.
    - tipul de date utilizat, care poate fi:
      `i` (integer) ($\texttt{i} \in \{0, 1, \ldots, 255\}$)
      `f` (float)
      `d` (double) ($\texttt{f}, \texttt{d} \in [0.0, 1.0]$)

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:
    - dimensiunea $n$ a spațiului de culori în care lucrăm, $n = 3$ (RGB) sau $n = 4$ (RGBA); A=factorul "alpha", legat de opacitate/transparență.
    - tipul de date utilizat, care poate fi:
      `i` (integer) ($\texttt{i} \in \{0, 1, \ldots, 255\}$)
      `f` (float)
      `d` (double) ($\texttt{f}, \texttt{d} \in [0.0, 1.0]$)
    - (opțional) posibila formă vectorială, indicată prin sufixul `v`.

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:
    - dimensiunea $n$ a spațiului de culori în care lucrăm, $n = 3$ (RGB) sau $n = 4$ (RGBA); A=factorul "alpha", legat de opacitate/transparență.
    - tipul de date utilizat, care poate fi:
      `i` (integer) ($\texttt{i} \in \{0, 1, \ldots, 255\}$)
      `f` (float)
      `d` (double) ($\texttt{f}, \texttt{d} \in [0.0, 1.0]$)
    - (opțional) posibila formă vectorială, indicată prin sufixul `v`.
  - în OpenGL "nou": culorile sunt manevrate într-un mod asemănător vârfurilor (folosind VBO), odată cu acestea (atribute ale vârfurilor).

# Spațiul culorilor

- Culorile sunt obținute combinând intensitățile de pe trei canale: R (red); G (green); B (blue) – Cubul RGB.
- În OpenGL o culoare este indicată prin:
  - în OpenGL "vechi" folosind funcția

    `glColor* ( );` – "deprecated"

    Sufixul * poate indica:
    - dimensiunea $n$ a spațiului de culori în care lucrăm, $n = 3$ (RGB) sau $n = 4$ (RGBA); A=factorul "alpha", legat de opacitate/transparență.
    - tipul de date utilizat, care poate fi:
      `i` (integer) (`i` $\in \{0, 1, \ldots, 255\}$)
      `f` (float)
      `d` (double) (`f`, `d` $\in [0.0, 1.0]$)
    - (opțional) posibila formă vectorială, indicată prin sufixul `v`.
  - în OpenGL "nou": culorile sunt manevrate într-un mod asemănător vârfurilor (folosind VBO), odată cu acestea (atribute ale vârfurilor).
  - elementul comun este faptul că, în ambele cazuri, culorile conțin informații legate de canalele R, G, B, precum și de canalul A (factorul $\alpha$, legat de opacitate).

# Vârfuri - funcții pentru indicare

Primitivele grafice sunt trasate cu ajutorul **vârfurilor** (*entități abstracte, a nu fi confundate cu punctele!*). În OpenGL un vârf este definit:

# Vârfuri - funcții pentru indicare

Primitivele grafice sunt trasate cu ajutorul **vârfurilor** (*entități abstracte, a nu fi confundate cu punctele!*). În OpenGL un vârf este definit:

- în OpenGL "vechi" cu ajutorul funcției

```
glVertex* (  );
```

– "deprecated"

# Vârfuri - funcții pentru indicare

Primitivele grafice sunt trasate cu ajutorul **vârfurilor** (*entități abstracte, a nu fi confundate cu punctele!*). În OpenGL un vârf este definit:

- în OpenGL "vechi" cu ajutorul funcției

`glVertex* ( );` – "deprecated"

- în OpenGL "nou":
  - vârfurile sunt stocate în matrice;
  - împachetate în *VAO (Vertex Array Objects)*;
  - trimise plăcii grafice sub formă de *VBO (Vertex Buffer Objects)*.

  O serie întreagă de funcții asociate (exemple mai jos, v. și slide-uri):

```
//  Se creeaza un buffer pentru VARFURI;
glGenBuffers(1, &VboId);
glBindBuffer(GL_ARRAY_BUFFER, VboId);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
//  Se asociaza atributul (0 = coordonate) pentru shader;
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),
  - o normală (legată de funcții de iluminare),

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),
  - o normală (legată de funcții de iluminare),
  - coordonate de texturare

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),
  - o normală (legată de funcții de iluminare),
  - coordonate de texturare
- În OpenGL "vechi": pentru o anumită caracteristică, este considerată valoarea *curentă* a respectivei caracteristici. Altfel spus, ea trebuie indicată în codul sursă *înaintea vârfului*.

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),
  - o normală (legată de funcții de iluminare),
  - coordonate de texturare
- În OpenGL "vechi": pentru o anumită caracteristică, este considerată valoarea *curentă* a respectivei caracteristici. Altfel spus, ea trebuie indicată în codul sursă *înaintea* *vârfului*.
- În OpenGL "modern": pentru o anumită caracteristică, este generat un *buffer object*, la fel ca pentru coordonatele vârfurilor. Alternativ, caracteristica respectivă poate fi indicată în aceeași matrice cu coordonatele vârfurilor.

# Caracteristici ale vârfurilor

- Unui vârf îi sunt asociate:
  - coordonate (fac parte din definiție),
  - o culoare (v. secțiunea Culori...),
  - o normală (legată de funcții de iluminare),
  - coordonate de texturare
- În OpenGL "vechi": pentru o anumită caracteristică, este considerată valoarea *curentă* a respectivei caracteristici. Altfel spus, ea trebuie indicată în codul sursă *înaintea vârfului*.
- În OpenGL "modern": pentru o anumită caracteristică, este generat un *buffer object*, la fel ca pentru coordonatele vârfurilor. Alternativ, caracteristica respectivă poate fi indicată în aceeași matrice cu coordonatele vârfurilor.
- Caracteristicile vârfurilor, indicate în programul principal, se regăsesc (sub un format adecvat) și în *shader*-ul de vârfuri.

# Funcții pentru primitive

Vârfurile sunt utilizate pentru trasarea *primitivelor grafice*. Funcțiile folosite sunt diferite pentru cele două moduri de randare:

- în OpenGL "vechi", o funcție de tipul `glVertex ( )` poate fi apelată într-un cadru de tip

```
glBegin (*);

glEnd;
```

– "deprecated"

(unde * reprezintă tipul de primitivă generat);

# Funcții pentru primitive

Vârfurile sunt utilizate pentru trasarea *primitivelor grafice*. Funcțiile folosite sunt diferite pentru cele două moduri de randare:

- în OpenGL "vechi", o funcție de tipul `glVertex ( )` poate fi apelată într-un cadru de tip

```
glBegin (*);

glEnd;
```

– "deprecated"

(unde * reprezintă tipul de primitivă generat);

- în OpenGL "nou" este utilizată o funcție de tipul

```
glDrawArrays (GLenum mode, GLint first, GLint count);
```

unde

`mode`: tipul primitivei;
`first`: primul vârf;
`count`: câte vârfuri se iau în considerare.

# Tipuri de primitive (corespund lui `GLenum mode`)

- Puncte: `GL_POINTS`

# Tipuri de primitive (corespund lui `GLenum mode`)

- Puncte: `GL_POINTS`
- Segmente de dreaptă: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`

# Tipuri de primitive (corespund lui `GLenum mode`)

- Puncte: `GL_POINTS`
- Segmente de dreaptă: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
- Triunghiuri: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`

# Tipuri de primitive (corespund lui `GLenum mode`)

- Puncte: `GL_POINTS`
- Segmente de dreaptă: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
- Triunghiuri: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- Dreptunghiuri: `GL_QUADS`, `GL_QUAD_STRIP`

# Tipuri de primitive (corespund lui `GLenum mode`)

- Puncte: `GL_POINTS`
- Segmente de dreaptă: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
- Triunghiuri: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- Dreptunghiuri: `GL_QUADS`, `GL_QUAD_STRIP`
- Poligoane (convexe!): `GL_POLYGON`

# Despre GLSL și obiecte-program

- GLSL este un limbaj de nivel înalt, cu sintaxă asemănătoare limbajului C: detalii în specificațiile GLSL - sunt prezentate funcțiile standard utilizate.

# Despre GLSL și obiecte-program

- GLSL este un limbaj de nivel înalt, cu sintaxă asemănătoare limbajului C: detalii în specificațiile GLSL - sunt prezentate funcțiile standard utilizate.
- Avantaje: utilizare pe platforme multiple, compatibilitate cu diverse tipuri de plăci grafice.
- Motivație: de a controla banda grafică (*graphics pipeline*). Finalitate: obținerea unui obiect-program (*Program Object*) care să fie rulat pe placa grafică.

# Despre GLSL și obiecte-program

- GLSL este un limbaj de nivel înalt, cu sintaxă asemănătoare limbajului C: detalii în specificațiile GLSL - sunt prezentate funcțiile standard utilizate.
- Avantaje: utilizare pe platforme multiple, compatibilitate cu diverse tipuri de plăci grafice.
- Motivație: de a controla banda grafică (*graphics pipeline*). Finalitate: obținerea unui obiect-program (*Program Object*) care să fie rulat pe placa grafică.
- Pentru a genera obiectul-program, trebuie parcurse o serie de etape prin intermediul programului principal - aplicația instalează codul GLSL pe placa grafică. Practic: v. *slide*-ul următor.

# Despre GLSL și obiecte-program

- GLSL este un limbaj de nivel înalt, cu sintaxă asemănătoare limbajului C: detalii în specificațiile GLSL - sunt prezentate funcțiile standard utilizate.
- Avantaje: utilizare pe platforme multiple, compatibilitate cu diverse tipuri de plăci grafice.
- Motivație: de a controla banda grafică (*graphics pipeline*). Finalitate: obținerea unui obiect-program (*Program Object*) care să fie rulat pe placa grafică.
- Pentru a genera obiectul-program, trebuie parcurse o serie de etape prin intermediul programului principal - aplicația instalează codul GLSL pe placa grafică. Practic: v. *slide*-ul următor.
- Codul este efectiv rulat la apelarea `glDrawArrays()` sau a unei funcții de desenare similare.

# Despre *shader*-e

- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

```
GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
```

din fișierul `LoadShaders.cpp`
Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.

# Despre *shader*-e

- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

  ```
  GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
  ```

  din fișierul `LoadShaders.cpp`
  Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.
- Fluxul:

# Despre *shader*-e

- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

  ```
  GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
  ```

  din fișierul `LoadShaders.cpp`
  Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.
- Fluxul:
  - (dacă este cazul) obținerea codului din fișier și transformarea în *string*

# Despre *shader*-e

- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

```
GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
```

din fișierul `LoadShaders.cpp`
Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.

- Fluxul:
  - (dacă este cazul) obținerea codului din fișier și transformarea în *string*
  - crearea unui obiect *shader* `glCreateShader()`, încărcarea *string*-ului în obiectul *shader* `glShaderSource()` și compilarea `glCompileShader()` - pas realizat separat pentru fiecare tip de *shader*

# Despre *shader*-e

- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

```
GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
```

din fișierul `LoadShaders.cpp`
Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.

- Fluxul:
  - (dacă este cazul) obținerea codului din fișier și transformarea în *string*
  - crearea unui obiect *shader* `glCreateShader()`, încărcarea *string*-ului în obiectul *shader* `glShaderSource()` și compilarea `glCompileShader()` - pas realizat separat pentru fiecare tip de *shader*
  - crearea unui obiect-program `glCreateProgram()`, atașarea obiectelor *shader compilate* `glAttachShader()` și legarea programului `glLinkProgram()`

# Despre *shader*-e

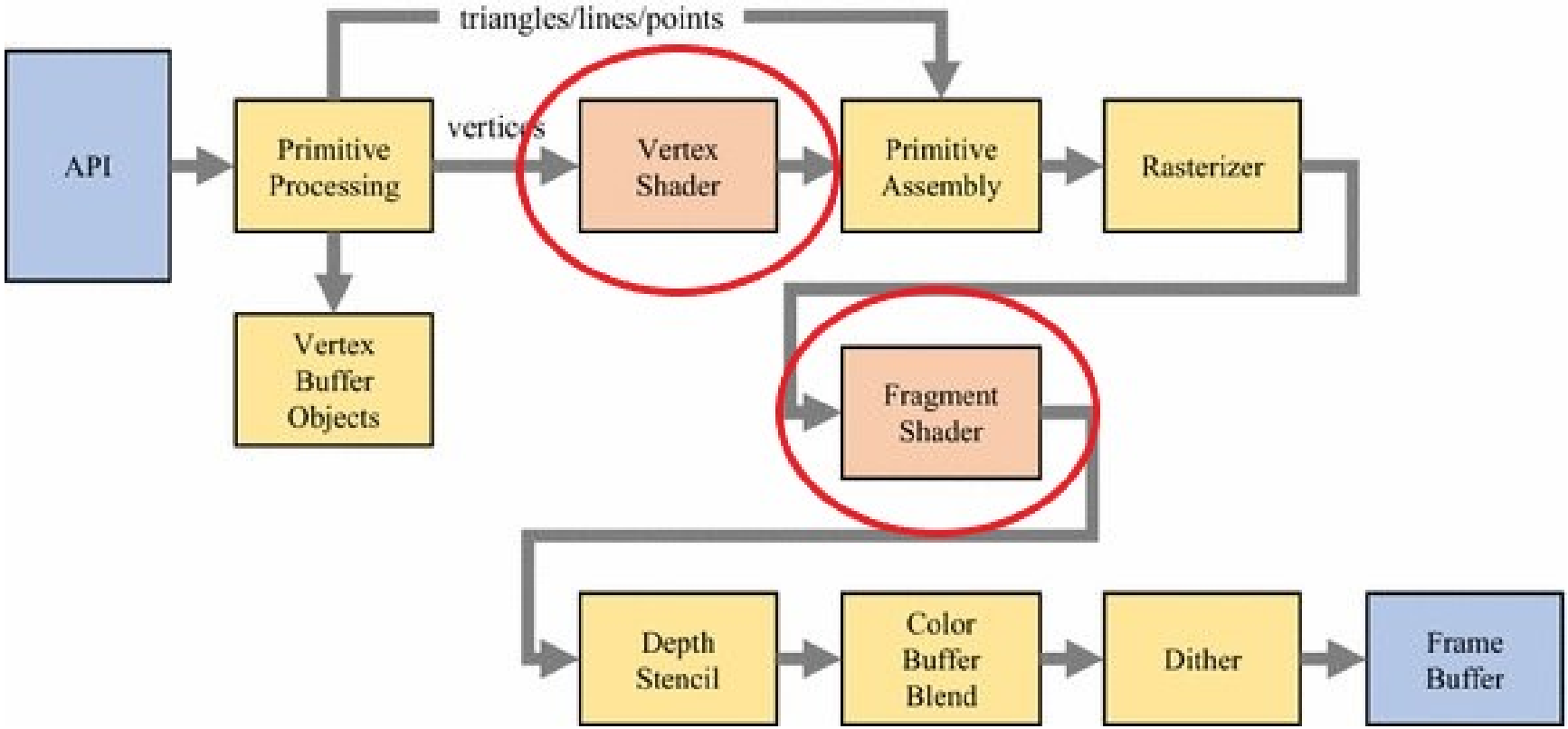
- Manevrare: în *template*-ul pus la dispoziție, etapele specifice sunt incluse în funcția

```
GLuint LoadShaders(const char *vertex_file_path,const char *fragment_file_path)
```

din fișierul `LoadShaders.cpp`
Ca date de intrare: calea către *shader*-ele de vârfuri, respectiv fragment. Funcția returnează un identificator al obiectului-program generat.

- Fluxul:
  - (dacă este cazul) obținerea codului din fișier și transformarea în *string*
  - crearea unui obiect *shader* `glCreateShader()`, încărcarea *string*-ului în obiectul *shader* `glShaderSource()` și compilarea `glCompileShader()` - pas realizat separat pentru fiecare tip de *shader*
  - crearea unui obiect-program `glCreateProgram()`, atașarea obiectelor *shader compilate* `glAttachShader()` și legarea programului `glLinkProgram()`
  - utilizarea obiectului-program `glUseProgram()` - poate fi apelată inclusiv în funcția de desenare.

# “Programmable pipeline”




Sursa: Baek and Kim, 2019

# Shader-ul de vârfuri (*vertex shader*)

- Rol: procesează vârfurile (și informațiile despre acestea) și le trimite mai departe în banda grafică.

# Shader-ul de vârfuri (*vertex shader*)

- Rol: procesează vârfurile (și informațiile despre acestea) și le trimite mai departe în banda grafică.
- La apelarea unei funcții de desenare (de ex. `glDrawArrays()`) *shader*-ul de vârfuri procesează, unul câte unul (eventual în paralel) vârfurile din *VBO*.

# Shader-ul de vârfuri (*vertex shader*)

- Rol: procesează vârfurile (și informațiile despre acestea) și le trimite mai departe în banda grafică.
- La apelarea unei funcții de desenare (de ex. `glDrawArrays()`) *shader*-ul de vârfuri procesează, unul câte unul (eventual în paralel) vârfurile din *VBO*.
- Date de intrare / ieșire: informații despre vârfuri (poziție, culoare, etc.) și o serie de alte date (de ex. matrice ale transformărilor). În particular, în *shader*-ul de vârfuri pot fi aplicate diverse transformări (inclusiv pentru mișcare) și poate fi controlată geometria scenei. De asemenea, mai există variabilele uniforme, preluate din programul principal.

# Shader-ul de vârfuri (*vertex shader*)

- Rol: procesează vârfurile (și informațiile despre acestea) și le trimite mai departe în banda grafică.
- La apelarea unei funcții de desenare (de ex. `glDrawArrays()`) *shader*-ul de vârfuri procesează, unul câte unul (eventual în paralel) vârfurile din *VBO*.
- Date de intrare / ieșire: informații despre vârfuri (poziție, culoare, etc.) și o serie de alte date (de ex. matrice ale transformărilor). În particular, în *shader*-ul de vârfuri pot fi aplicate diverse transformări (inclusiv pentru mișcare) și poate fi controlată geometria scenei. De asemenea, mai există variabilele uniforme, preluate din programul principal.
- Datele furnizate de *shader*-ul de vârfuri trec printr-un proces de asamblare a primitivelor și de rasterizare, fiind transformate în fragmente (locații ale pixelilor) - proces realizat de API.

# Shader-ul de vârfuri (*vertex shader*)

- Rol: procesează vârfurile (și informațiile despre acestea) și le trimite mai departe în banda grafică.
- La apelarea unei funcții de desenare (de ex. `glDrawArrays()`) *shader*-ul de vârfuri procesează, unul câte unul (eventual în paralel) vârfurile din *VBO*.
- Date de intrare / ieșire: informații despre vârfuri (poziție, culoare, etc.) și o serie de alte date (de ex. matrice ale transformărilor). În particular, în *shader*-ul de vârfuri pot fi aplicate diverse transformări (inclusiv pentru mișcare) și poate fi controlată geometria scenei. De asemenea, mai există variabilele uniforme, preluate din programul principal.
- Datele furnizate de *shader*-ul de vârfuri trec printr-un proces de asamblare a primitivelor și de rasterizare, fiind transformate în fragmente (locații ale pixelilor) - proces realizat de API.
- Mai multe detalii: ulterior (de ex. texturare, modele de iluminare, etc.)

# Shader-ul de fragment (*fragment shader*)

- Rol: procesarea fragmentelor și stabilirea culorii pentru fiecare pixel din buffer-ul de cadru. Sunt realizate operații la nivel de fragment (de ex. amestecarea culorilor, combinarea cu texturile, etc.)

# Shader-ul de fragment (*fragment shader*)

- Rol: procesarea fragmentelor și stabilirea culorii pentru fiecare pixel din buffer-ul de cadru. Sunt realizate operații la nivel de fragment (de ex. amestecarea culorilor, combinarea cu texturile, etc.)
- Date de intrare: provin din *shader*-ul de vârfuri. Date de ieșire: legate de culoarea fragmentelor. De asemenea, mai există variabilele uniforme, preluate din programul principal.

# Shader-ul de fragment (*fragment shader*)

- Rol: procesarea fragmentelor și stabilirea culorii pentru fiecare pixel din buffer-ul de cadru. Sunt realizate operații la nivel de fragment (de ex. amestecarea culorilor, combinarea cu texturile, etc.)
- Date de intrare: provin din *shader*-ul de vârfuri. Date de ieșire: legate de culoarea fragmentelor. De asemenea, mai există variabilele uniforme, preluate din programul principal.
- Mai multe detalii: ulterior (de ex. texturare, modele de iluminare, etc.)

# Elemente de principiu - comunicarea dintre progr. princ. și *shader*-e

Două moduri de comunicare

- Prin intermediul *buffer*-elor (vârfuri și caracteristici asociate) - C++ și al *vertex attributes* - *shader*.
- Folosind variabile uniforme.

# Pentru vârfuri și caracteristici ale acestora - prog. princ.

- Pași realizați:

# Pentru vârfuri și caracteristici ale acestora - prog. princ.

- Pași realizați:
  - creare vectori cu vârfuri, caracteristici (culoare, etc.) - pot fi indicate separat sau în aceeași matrice, indici;
  - generare nume ptr. buffer-objects: `glGenBuffers( )`;
  - activare/"legare" buffer `glBindBuffer( )` și transfer de date în buffer: `glBufferData( )`.

# Pentru vârfuri și caracteristici ale acestora - prog. princ.

- Pași realizați:
  - creare vectori cu vârfuri, caracteristici (culoare, etc.) - pot fi indicate separat sau în aceeași matrice, indici;
  - generare nume ptr. buffer-objects: `glGenBuffers( )`;
  - activare/"legare" buffer `glBindBuffer( )` și transfer de date în buffer: `glBufferData( )`.
  - asociere cu un *vertex attribute* și indicarea locațiilor (ptr. *shader!*): glVertexAttribPointer( ) - parametrul `GLuint index` trebuie să se regăsească, cu aceeași valoare, și în *shader*,
  - activare a *vertex attribute*: glEnableVertexAttribArray( ) - parametrii `GLuint index` și `GLint size` trebuie să se regăsească, cu aceeași valoare, și în *shader*,

- Structura specifică pentru *buffer*: *VBO (Vertex Buffer Object)*. Pot fi utilizate mai multe *VBO* (inclusiv pentru vârfuri).

# Pentru vârfuri și caracteristici ale acestora - prog. princ.

- Pași realizați:
  - creare vectori cu vârfuri, caracteristici (culoare, etc.) - pot fi indicate separat sau în aceeași matrice, indici;
  - generare nume ptr. buffer-objects: `glGenBuffers( )`;
  - activare/"legare" buffer `glBindBuffer( )` și transfer de date în buffer: `glBufferData( )`.
  - asociere cu un *vertex attribute* și indicarea locațiilor (ptr. *shader!*): glVertexAttribPointer( ) - parametrul `GLuint index` trebuie să se regăsească, cu aceeași valoare, și în *shader*,
  - activare a *vertex attribute*: glEnableVertexAttribArray( ) - parametrii `GLuint index` și `GLint size` trebuie să se regăsească, cu aceeași valoare, și în *shader*,
  - apelare funcție de desenare.
- Structura specifică pentru *buffer*: *VBO (Vertex Buffer Object)*. Pot fi utilizate mai multe *VBO* (inclusiv pentru vârfuri).
- O structură asociată: *VAO (Vertex Array Object)* - are rolul de a permite gestionarea eficientă și organizarea *VBO* - necesar cel puțin un *VAO*. Funcții: `glGenVertexArrays`, `glBindVertexArray( )`.

# Pentru vârfuri și caracteristici ale acestora - *shader*

- În *shader*-ul de vârfuri:

# Pentru vârfuri și caracteristici ale acestora - *shader*

- În *shader*-ul de vârfuri:
  - Informațiile despre vârfuri sunt transferate în *shader* ca *vertex attributes*, aici devenind valori de intrare (*input*). Parametrii `index` și `size` din funcția `glEnableVertexAttribArray( )` se regăsesc (cu aceleași valori) apelați în `location = index`, respectiv `vec*`, unde `*=size`. Exemplu:

    ```
    layout (location = 0) in vec4 in_Position;
    layout (location = 1) in vec3 in_Color;
    ```

    Indexul `0` corespunde unui vector cu 4 componente (coordonate).
    Indexul `1` corespunde unui vector cu 3 componente (culori).

# Pentru vârfuri și caracteristici ale acestora - *shader*

- În *shader*-ul de vârfuri:
  - Informațiile despre vârfuri sunt transferate în *shader* ca *vertex attributes*, aici devenind valori de intrare (*input*). Parametrii `index` și `size` din funcția `glEnableVertexAttribArray( )` se regăsesc (cu aceleași valori) apelați în `location = index`, respectiv `vec*`, unde `*=size`. Exemplu:

    ```
    layout (location = 0) in vec4 in_Position;
    layout (location = 1) in vec3 in_Color;
    ```

    Indexul `0` corespunde unui vector cu 4 componente (coordonate).
    Indexul `1` corespunde unui vector cu 3 componente (culori).
  - Trebuie să existe o corespondență perfectă între valorile din programul principal și cele din *shader*, altminteri efectul nu va fi cel dorit.

# Variabile uniforme

- În programul principal:

# Variabile uniforme

- În programul principal:
  - obținerea unei referințe către variabila uniformă
    `glGetUniformLocation(GLuint program, const GLchar *name)`
    Variabila `program` și string-ul cu numele variabilei trebuie modificate.
    Exemplu:
    ```
    codColLocation = glGetUniformLocation(ProgramId,
    "codColShader");
    ```
    Această funcție poate fi apelată și în `init`

# Variabile uniforme

- În programul principal:
  - obținerea unei referințe către variabila uniformă
    `glGetUniformLocation(GLuint program, const GLchar *name)`
    Variabila `program` și string-ul cu numele variabilei trebuie modificate.
    Exemplu:
    ```
    codColLocation = glGetUniformLocation(ProgramId,
    "codColShader");
    ```
    Această funcție poate fi apelată și în `init`
  - specificarea valorii variabilei `glUniform*()` Sufixul * indică informații despre variabilă. De asemenea, trebuie indicată locația utilizată.
    Această funcție trebuie apelată la fiecare modificare a valorii variabilei.
    Exemplu:
    ```
    int codCol = 0;
    glUniform1i(codColLocation, codCol);
    ```

# Variabile uniforme

- În programul principal:
  - obținerea unei referințe către variabila uniformă
    `glGetUniformLocation(GLuint program, const GLchar *name)`
    Variabila `program` și string-ul cu numele variabilei trebuie modificate.
    Exemplu:
    ```
    codColLocation = glGetUniformLocation(ProgramId,
    "codColShader");
    ```
    Această funcție poate fi apelată și în `init`
  - specificarea valorii variabilei `glUniform*()` Sufixul * indică informații despre variabilă. De asemenea, trebuie indicată locația utilizată.
    Această funcție trebuie apelată la fiecare modificare a valorii variabilei.
    Exemplu:
    ```
    int codCol = 0;
    glUniform1i(codColLocation, codCol);
    ```
- În *shader*:

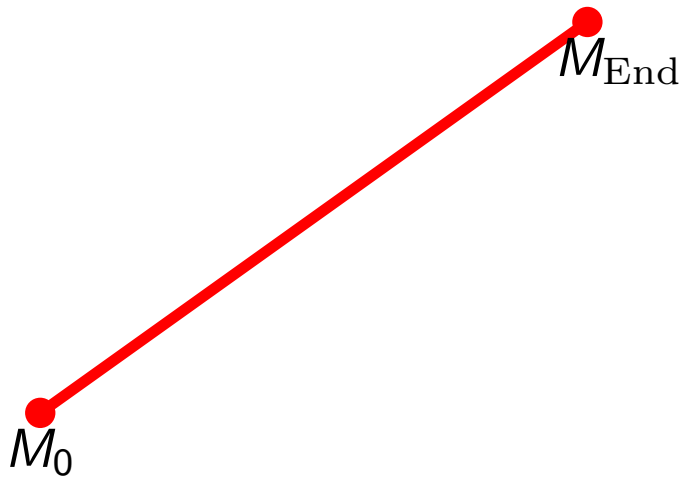
# Variabile uniforme

- În programul principal:
  - obținerea unei referințe către variabila uniformă
    `glGetUniformLocation(GLuint program, const GLchar *name)`
    Variabila `program` și string-ul cu numele variabilei trebuie modificate.
    Exemplu:
    ```
    codColLocation = glGetUniformLocation(ProgramId,
    "codColShader");
    ```
    Această funcție poate fi apelată și în `init`
  - specificarea valorii variabilei `glUniform*()` Sufixul * indică informații despre variabilă. De asemenea, trebuie indicată locația utilizată.
    Această funcție trebuie apelată la fiecare modificare a valorii variabilei.
    Exemplu:
    ```
    int codCol = 0;
    glUniform1i(codColLocation, codCol);
    ```
- În *shader*:
  - declararea variabilei uniforme și a tipului acesteia.
    Exemplu:
    ```
    uniform int codColShader
    ```

# Motivație și problematizare

"Continuu"

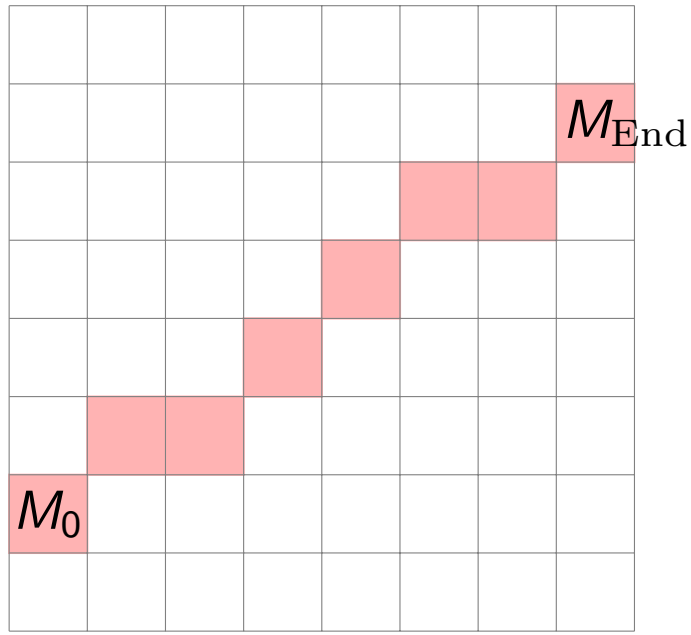
"Grafica vectorială"




$$M_0 = (x_0, y_0), M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$$
$$x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{R}$$

"Discret"

"Grafica rasterială"

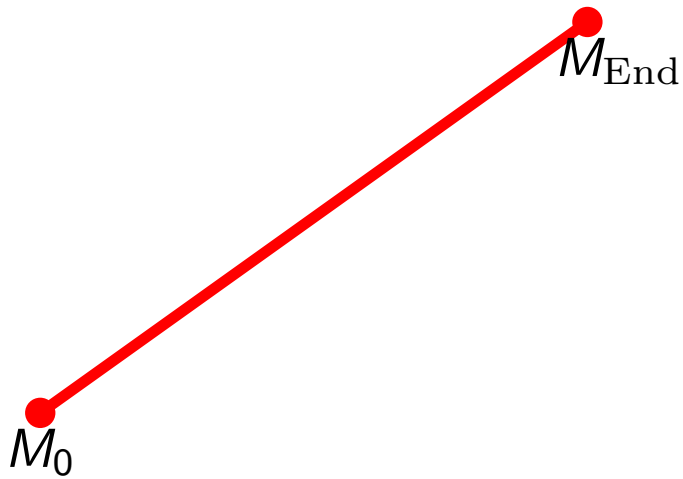



$$M_0 = (x_0, y_0), M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$$
$$x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{N}(\mathbb{Z})$$

# Motivație și problematizare

"Continuu"

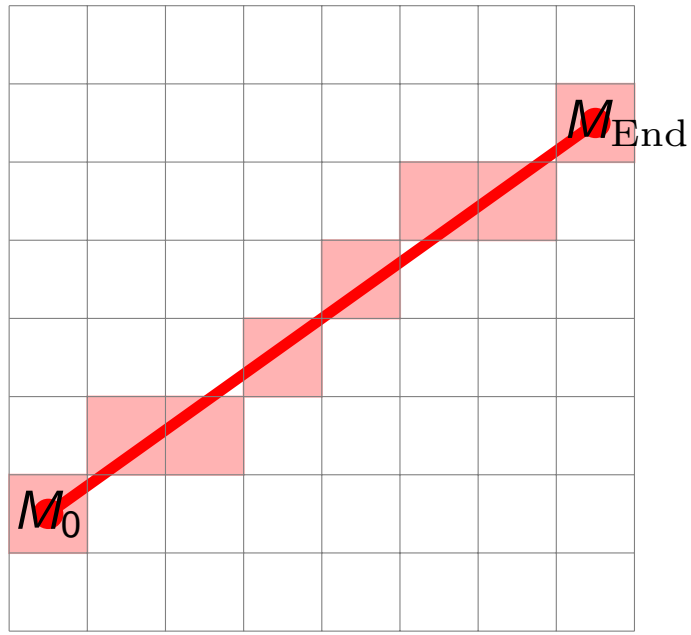
"Grafica vectorială"




$$M_0 = (x_0, y_0), M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$$
$$x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{R}$$

"Discret"

"Grafica rasterială"




$$M_0 = (x_0, y_0), M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$$
$$x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{N}(\mathbb{Z})$$

# Ce este un algoritm de rasterizare?

Un algoritm de rasterizare are ca efect reprezentarea grafică a unei primitive într-un sistem de reprezentare de tip grilă (monitor), care este format dintr-o structură discretă de pixeli. Pentru un segment, un algoritm de rasterizare are ca:

> **Input:** Coordonatele $x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{N}\ (\in \mathbb{Z})$ ale extremităților segmentului care urmează să fie reprezentat - altfel spus, pixelii inițial $M_0 = (x_0, y_0)$, respectiv final $M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$.

# Ce este un algoritm de rasterizare?

Un algoritm de rasterizare are ca efect reprezentarea grafică a unei primitive într-un sistem de reprezentare de tip grilă (monitor), care este format dintr-o structură discretă de pixeli. Pentru un segment, un algoritm de rasterizare are ca:

**Input:** Coordonatele $x_0, y_0, x_{\text{End}}, y_{\text{End}} \in \mathbb{N}$ $(\in \mathbb{Z})$ ale extremităților segmentului care urmează să fie reprezentat - altfel spus, pixelii inițial $M_0 = (x_0, y_0)$, respectiv final $M_{\text{End}} = (x_{\text{End}}, y_{\text{End}})$.

**Output:** Pixelii selectați pentru a trasa segmentul de la $M_0$ la $M_{\text{End}}$

# Ipoteze / restricții

Raționamentele se pot adapta la restul situațiilor/cazurilor. Ipoteze (simplificatoare) făcute:

Ip. 1 Intuitiv: "deplasarea se face înspre dreapta/sus"

$$x_{\text{End}} > x_0, \quad y_{\text{End}} > y_0 \Leftrightarrow$$

$$\Leftrightarrow \Delta x > 0, \quad \Delta y > 0.$$

# Ipoteze / restricții

Raționamentele se pot adapta la restul situațiilor/cazurilor. Ipoteze (simplificatoare) făcute:

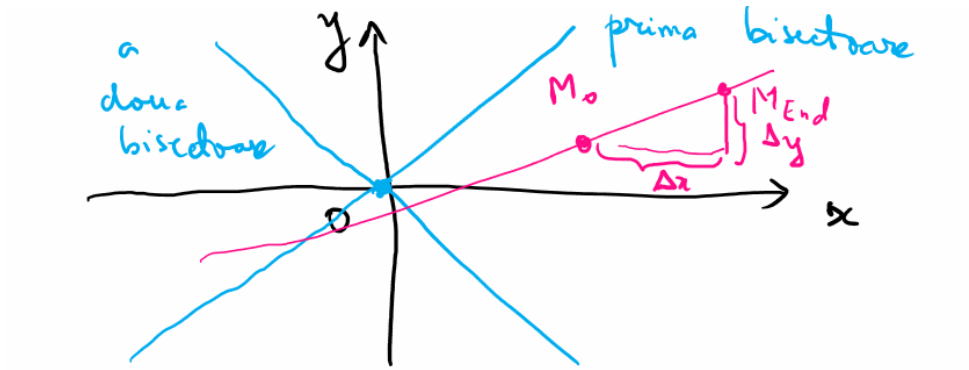
Ip. 1 Intuitiv: "deplasarea se face înspre dreapta/sus"

$$x_{\text{End}} > x_0, \quad y_{\text{End}} > y_0 \Leftrightarrow$$

$$\Leftrightarrow \Delta x > 0, \quad \Delta y > 0.$$

Ip. 2 Intuitiv: "dreapta trasată prin origine și paralelă cu dreapta suport este situată sub prima bisectoare"

$$\Leftrightarrow \Delta x > \Delta y > 0.$$



# Ecuația dreptei

Ecuația dreptei care unește punctele $M_0$ și $M_{\text{End}}$

$$y = mx + n. \tag{1}$$

# Ecuația dreptei

Ecuația dreptei care unește punctele $M_0$ și $M_{\text{End}}$

$$y = mx + n. \tag{1}$$

Elemente importante: panta $m$ și coeficientul $n$, care poate fi exprimat în funcție de pantă și de coordonatele lui $M_0$

$$m = \frac{\Delta y}{\Delta x} \quad \text{(panta)}$$

$$y_0 = mx_0 + n \implies n = y_0 - mx_0 \implies n = y_0 - \frac{\Delta y}{\Delta x} x_0$$

(pct $M_0(x_0, y_0)$ aparține dreptei)

# Ecuația dreptei

Ecuația dreptei care unește punctele $M_0$ și $M_{\text{End}}$

$$y = mx + n. \tag{1}$$

Elemente importante: panta $m$ și coeficientul $n$, care poate fi exprimat în funcție de pantă și de coordonatele lui $M_0$

$$m = \frac{\Delta y}{\Delta x} \quad \text{(panta)}$$

$$y_0 = mx_0 + n \Rightarrow n = y_0 - mx_0 \Rightarrow n = y_0 - \frac{\Delta y}{\Delta x} x_0$$

(pct $M_0(x_0, y_0)$ aparține dreptei)

**Concluzie:** În cazul continuu avem

$$y = mx + n \overset{NOT}{=} f(x).$$

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- `Iniţializare` $x_0, y_0 = f(x_0), m, n$

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m, n$
- Pasul $k \rightarrow k + 1 \ (k \geq 0)$

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m, n$
- Pasul $k \to k+1 \ (k \geq 0)$

  $x_{k+1} \leftarrow x_k + 1$

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m, n$
- Pasul $k \to k+1 \ (k \geq 0)$

  $x_{k+1} \leftarrow x_k + 1$

  $f(x_{k+1}) \leftarrow m \cdot x_{k+1} + n$ // formula (1)

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m, n$
- Pasul $k \to k+1 \ (k \geq 0)$

  $x_{k+1} \leftarrow x_k + 1$
  $f(x_{k+1}) \leftarrow m \cdot x_{k+1} + n$ // formula (1)
  $y_{k+1} \leftarrow \text{round}(f(x_{k+1}))$

# Varianta 1 - înlocuire în ecuația dreptei

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m, n$
- Pasul $k \to k+1$ $(k \geq 0)$

  $x_{k+1} \leftarrow x_k + 1$

  $f(x_{k+1}) \leftarrow m \cdot x_{k+1} + n$ // formula (1)

  $y_{k+1} \leftarrow \text{round}(f(x_{k+1}))$

# Calcule...

pentru $f(x_{k+1})$

# Calcule...

pentru $f(x_{k+1})$

$$f(x_{k+1}) = mx_{k+1} + n \overset{x_{k+1}=x_k+1}{=} m(x_k + 1) + n =$$

$$= mx_k + m + n = mx_k + n + m = f(x_k) + m.$$

# Varianta 2 - algoritmul Digital Differential Analyzer (DDA)

**Algoritm**

- Iniţializare $x_0, y_0 = f(x_0), m$
- Pasul $k \to k+1$ $(k \geq 0)$

  $x_{k+1} \leftarrow x_k + 1$

  $f(x_{k+1}) \leftarrow f(x_k) + m$ // formula (1), calculele anterioare

  $y_{k+1} \leftarrow \text{round}(f(x_{k+1}))$

# Exemplu

$$M_0 = (10, 20), M_{\text{End}} = (20, 28)$$

# Exemplu

$M_0 = (10, 20), M_{\text{End}} = (20, 28)$

$$\Delta x = 10,\ \Delta y = 8,\ m = \frac{\Delta y}{\Delta x} = \frac{8}{10} = 0.8.$$

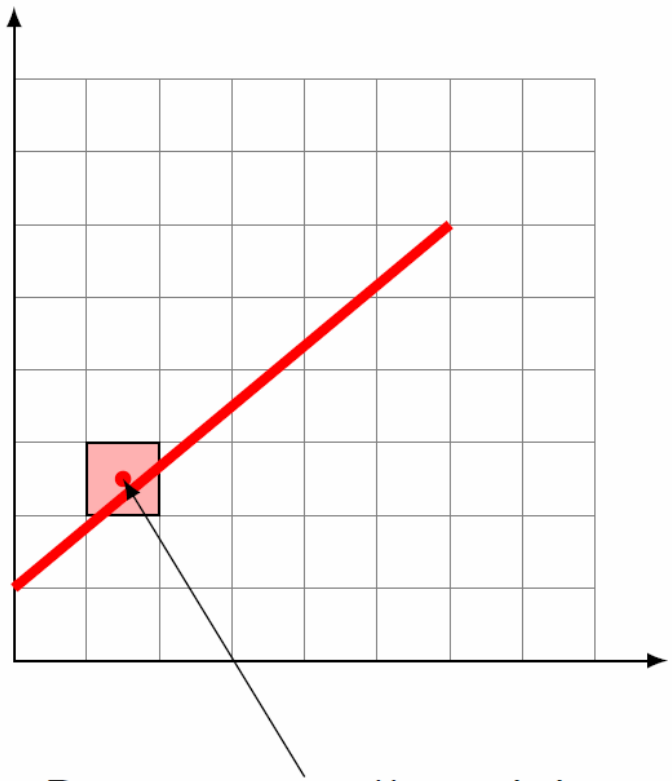
# Exemplu

$M_0 = (10, 20), M_{\text{End}} = (20, 28)$

$$\Delta x = 10,\ \Delta y = 8,\ m = \frac{\Delta y}{\Delta x} = \frac{8}{10} = 0.8.$$

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_k$ | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| $f(x_k)$ | 20 | 20.8 | 21.6 | 22.4 | 23.2 | 24 | 24.8 | 25.6 | 26.4 | 27.2 | 28 |
| $y_k$ | 20 | 21 | 22 | 22 | 23 | 24 | 25 | 26 | 26 | 27 | 28 |

# Varianta 3 - algoritmul lui Bresenham. Idei fundamentale



Presupunem că pixelul $(x_k, y_k)$ a fost selectat. Care este pixelul următor?

Doi "candidați" pt. pixelul următor:

$j: (x_k + 1, y_k)$

$s: (x_k + 1, y_k + 1)$

(⚠ ipoteza privind panta dreptei)

Ideea: se calculează distanța de la punctul de pe dreaptă corresp. abscisei $x_k + 1$ la centrele pixelilor candidați:

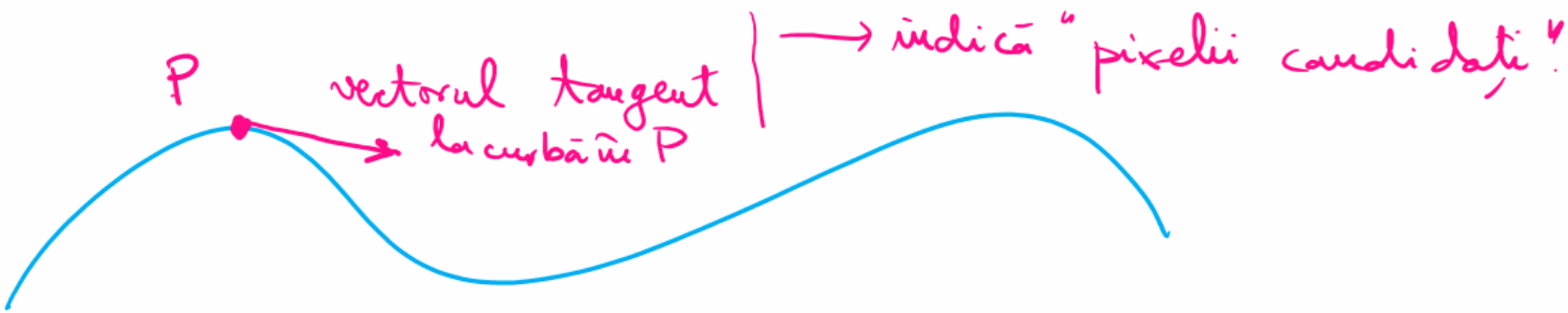
$d_s = y_k + 1 - f(x_k + 1)$

$d_j = f(x_k + 1) - y_k$

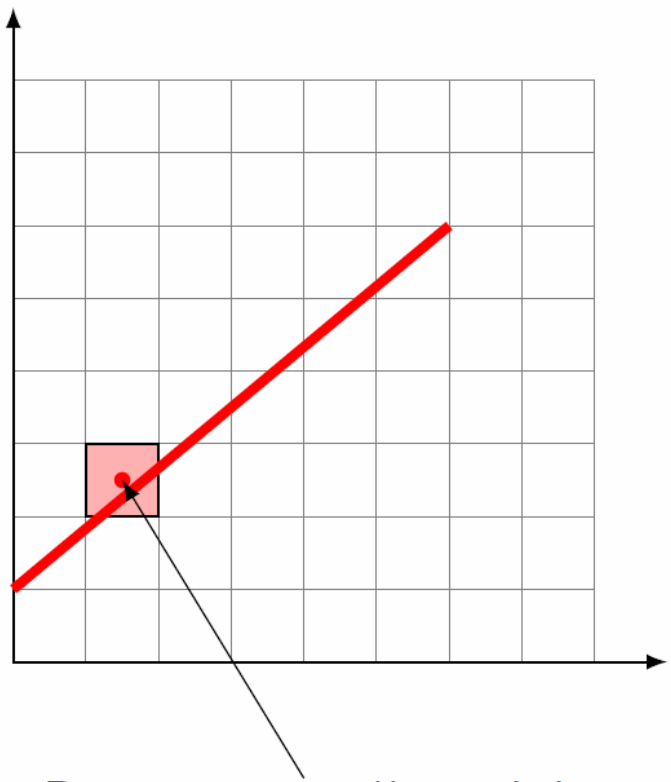
(sunt diferențe de ordonate, pct. au aceeași abscisă)

# Algoritmul lui Bresenham. Observație

Varianta originală (1965) avea în vedere în special segmentele de dreaptă. Folosind conceptul de vector tangent, algoritmul poate fi adaptat și aplicat și în cazul altor curbe.



# Algoritmul lui Bresenham. Factorul de decizie



Presupunem că pixelul $(x_k, y_k)$ a fost selectat.
Care este pixelul următor?

$d_s = y_k + 1 - f(x_k + 1)$

ordonata pixelului de sus — ordonata pct. de pe dreaptă

$d_j = f(x_k + 1) - y_k$

$d_s = y_k + 1 - m x_k - m - n$

$d_j = m x_k + m + n - y_k$

$d_s \gtrless d_j$ ?

# Algoritmul lui Bresenham. Semnul factorului de decizie

Ne interesează semnul numărului $d_j - d_s \in \mathbb{Q}$

$$d_j - d_s = \underbrace{2m}_{\frac{\Delta y}{\Delta x}}(x_k + 1) - 2y_k + 2m - 1 =$$

$$= \frac{2\Delta y(x_k + 1) - 2y_k \cdot \Delta x + 2m\Delta x - \Delta x}{\Delta x} \qquad \Delta x > 0$$

$$\boxed{f = 2\Delta y + 2m\Delta x - \Delta x} \quad \text{(notație)}$$

Deci: semnul lui $d_j - d_s$ este semnul $\boxed{p_k \overset{\text{DEF}}{=} 2\Delta y \cdot x_k - 2y_k \cdot \Delta x + f}$

# Algoritmul lui Bresenham. Parametrul de decizie

Semnul lui $d_j - d_s$ este semnul **parametrului de decizie**

$$p_k \overset{DEF}{=} 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + \gamma$$

- $p_k < 0 \Leftrightarrow d_j - d_s < 0 \Leftrightarrow d_j < d_s \Rightarrow$ se alege pixelul $(x_k + 1, y_k)$
- $p_k \geq 0 \Leftrightarrow d_j - d_s \geq 0 \Leftrightarrow d_j \geq d_s \Rightarrow$ se alege pixelul $(x_k + 1, y_k + 1)$

# Algoritmul lui Bresenham. Parametrul de decizie - rescriere

Evaluăm $p_{k+1} - p_k =$

$= 2\Delta y\, x_{k+1} - 2\Delta x\, y_{k+1} + \gamma - 2\Delta y\, x_k + 2\Delta x\, y_k - \gamma =$

$= 2\Delta y\,(x_{k+1} - x_k = 1) - 2\Delta x\,(y_{k+1} - y_k) =$

$y_{k+1} - y_k = \begin{cases} 0, & p_k < 0 \\ 1, & p_k \geq 0 \end{cases}$

$= \begin{cases} 2\Delta y, & \text{dacă } p_k < 0 \text{ (adică } y_{k+1} = y_k) \\ 2\Delta y - 2\Delta x, & \text{dacă } p_k \geq 0 \text{ (adică } y_{k+1} = y_k + 1) \end{cases}$

# Algoritmul lui Bresenham. Parametrul de decizie - valoarea lui $p_0$

Evaluăm

$$p_0 = 2\Delta y \cdot x_0 - 2\Delta x \cdot y_0 + \gamma =$$

$$= 2\Delta y \cdot x_0 - 2\Delta x \cdot (m \cdot x_0 + n) + 2\Delta y + 2\Delta x \cdot n - \Delta x \Rightarrow$$

$$p_0 = 2\Delta y - \Delta x.$$

# Algoritmul lui Bresenham

**Algoritm**

- `Calcul preliminar` $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$

# Algoritmul lui Bresenham

**Algoritm**

- Calcul preliminar $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- Pasul $k \to k+1$ $(k \geq 0)$. // Pp. că avem $x_k, y_k, p_k \in \mathbb{Z}$

# Algoritmul lui Bresenham

**Algoritm**

- Calcul preliminar $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- Pasul $k \to k+1$ $(k \geq 0)$. // Pp. că avem $x_k, y_k, p_k \in \mathbb{Z}$

  Dacă $p_k < 0$ (‘‘stagnare pe orizontală’’)

# Algoritmul lui Bresenham

**Algoritm**

- Calcul preliminar $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- Pasul $k \to k+1$ $(k \geq 0)$. // Pp. că avem $x_k, y_k, p_k \in \mathbb{Z}$

  Dacă $p_k < 0$ (‘‘stagnare pe orizontală’’)

  $x_{k+1} \leftarrow x_k + 1$

  $y_{k+1} \leftarrow y_k$

  $p_{k+1} \leftarrow p_k + 2\Delta y$

# Algoritmul lui Bresenham

**Algoritm**

- Calcul preliminar $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- Pasul $k \to k+1$ $(k \geq 0)$. // Pp. că avem $x_k, y_k, p_k \in \mathbb{Z}$

  Dacă $p_k < 0$ (‘‘stagnare pe orizontală’’)

  $x_{k+1} \leftarrow x_k + 1$

  $y_{k+1} \leftarrow y_k$

  $p_{k+1} \leftarrow p_k + 2\Delta y$

  Dacă $p_k \geq 0$ (‘‘în sus’’)

# Algoritmul lui Bresenham

**Algoritm**

- Calcul preliminar $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- Pasul $k \to k+1$ $(k \geq 0)$. // Pp. că avem $x_k, y_k, p_k \in \mathbb{Z}$

  Dacă $p_k < 0$ (‘‘stagnare pe orizontală’’)

  $x_{k+1} \leftarrow x_k + 1$
  $y_{k+1} \leftarrow y_k$
  $p_{k+1} \leftarrow p_k + 2\Delta y$

  Dacă $p_k \geq 0$ (‘‘în sus’’)

  $x_{k+1} \leftarrow x_k + 1$
  $y_{k+1} \leftarrow y_k + 1$
  $p_{k+1} \leftarrow p_k + 2\Delta y - 2\Delta x$

# Algoritmul lui Bresenham. Exemplu

$M_0 = (10, 20), M_{\text{End}} = (20, 28)$

$x_0 = 10,\ y_0 = 20,\ x_{\text{End}} = 20,\ y_{\text{End}} = 28.$

$\Delta x = 10$ , $\Delta y = 8$

$p_0 = 2\Delta y - \Delta x = 6$

$2\Delta y = 16$

$2\Delta y - 2\Delta x = -4$

| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_k$ | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| $y_k$ | 20 | 21 | 22 | 22 | 23 | 24 | 25 | 26 | 26 | 27 | 28 |
| $p_k$ | 6 | 2 | −2 | 14 | 10 | 6 | 2 | −2 | 14 | 10 | 6 |

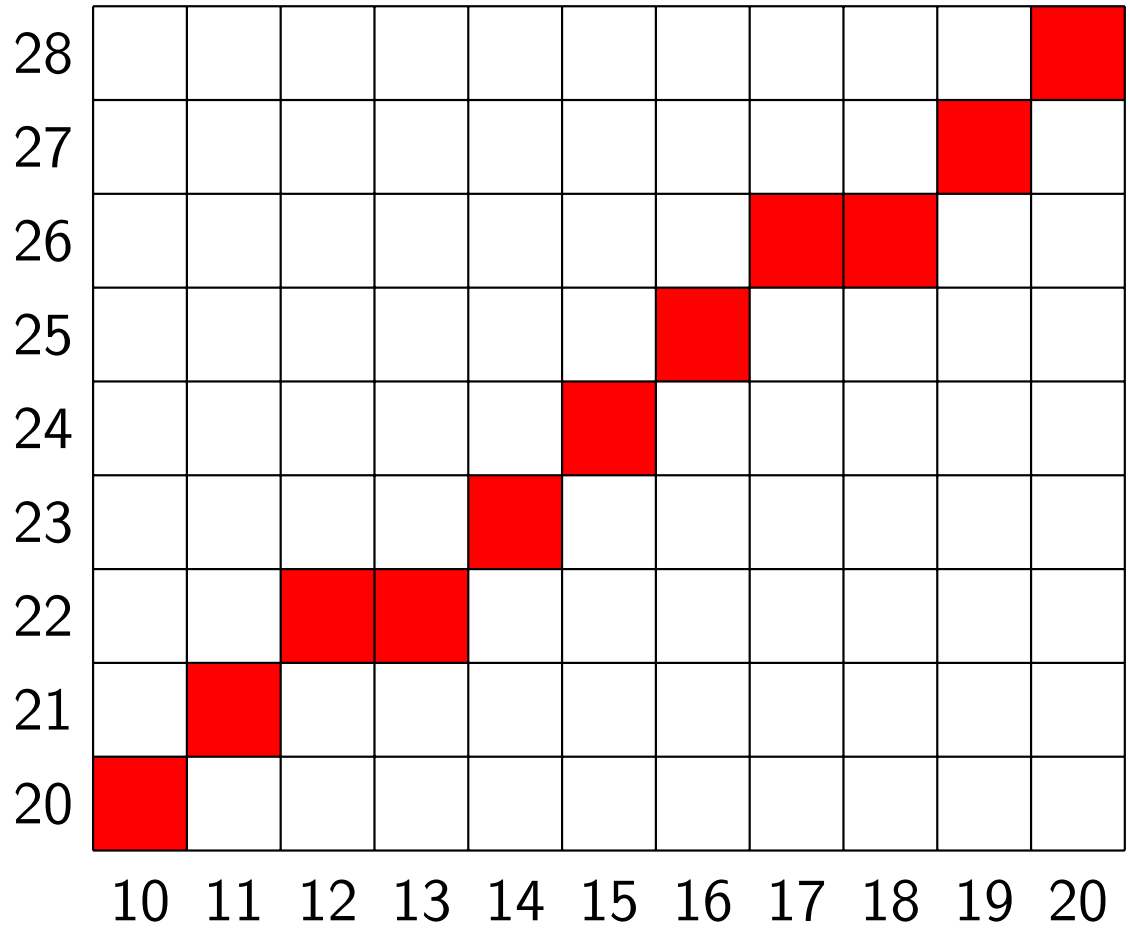



Figura: **Algoritmul DDA / algoritmul lui Bresenham.** Pixelii selectați pentru a uni punctele $(10, 20)$ și $(20, 28)$ sunt colorați cu roșu.